Exhibit A
(Part 2of 2)

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Frakes discloses an information storage and retrieval system. For example, Frakes discloses "hashing, an information storage and retrieval technique useful for implementing many … other structures." (Frakes at 293).
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Frakes discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Frakes also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Frakes discloses "hashing, an information storage and retrieval technique useful for implementing many … other structures." (Frakes at 293). Frakes discloses that *hashing* is "a ubiquitous information retrieval strategy for providing efficient access to information based on a key." *Id.* Frakes further discloses "chained hashing. It is so named because each bucket stores a linked list—that is, a chain—of key-information pairs, rather than a single one." (Frakes at 298).
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Frakes discloses a record search means utilizing a search key to access the linked list. Frakes also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, Frakes discloses that "[t]he goal (of hashing) is to avoid *collisions*. A collision occurs when two or more keys map to the same location. If no keys collide, then locating the information associated with a key is simply the process of determining the key's location. Whenever a

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		collision occurs, some extra computation is necessary to further determine a unique location for a key." (Frakes at 294).
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	Frakes discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. Frakes also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, Frakes discloses that a "hash table with *m* buckets may therefore store more than *m* keys. However, performance will degrade as the number of keys increases. Computing the bucket in which a key resides is still fast—a matter of evaluating the hash function—but locating it within that bucket (or simply determining its presence, which is necessary in all operations) requires traversing the linked list." (Frakes at 299). Additionally, Frakes discloses that "performance will degrade as the number of keys increases." *Id.* Thus, Frakes suggests removing at least some of the automatically expired records from the linked list when the linked list is accessed. (*See id.*)
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Frakes discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Frakes also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, Frakes discloses several operations that are usually provided by an implementation of hashing:

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		1. Initialization: indicate that the hash table contains no elements. 2. Insertion: insert information, indexed by a key *k*, into a hash table. If the has table already contains *k*, then it cannot be inserted. (Some implementations do allow such insertion, to permit replacing existing information.) 3. Retrieval: given a key *k*, retrieve the information associated with it. 4. Deletion: remove the information associated with key *k* from a hash table, if any exists. New information indexed by *k* may subsequently be placed in the table. (Frakes at 297). Frakes further discloses that "[m]ost of the code from the routines `Insert`, `Delete`, `Clear` (for `Initialize`), and `Member` (for `Retrieve`) can be used directly." (Frakes at 299). Additionally, Frakes discloses that "performance will degrade as the number of keys increases." *Id.* Thus Frakes suggests removing at least some of the automatically expired records from the linked list when the linked list is accessed. (*See id.*)
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Frakes to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Frakes with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		number of potential problems. For example, the removal of expired records described in Frakes can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Frakes is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Frakes combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) (“Frakes”) alone and in combination**
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT C-12

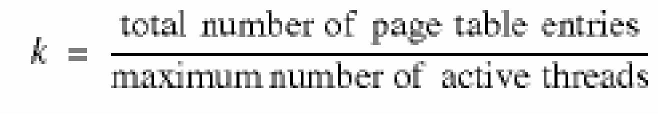
Asserted Claims From **U.S. Pat. No. 5,893,120**		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) (“Frakes”) alone and in combination**
		 $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Frakes and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Frakes. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Frakes nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Frakes and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Frakes with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Frakes with Thatte would be nothing more than the

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove. Further, one of ordinary skill in the art would be motivated to combine Frakes with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Frakes can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Frakes with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Frakes with Thatte. Alternatively, it would also be obvious to combine Frakeswith the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) (“Frakes”) alone and in combination**
		to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-12

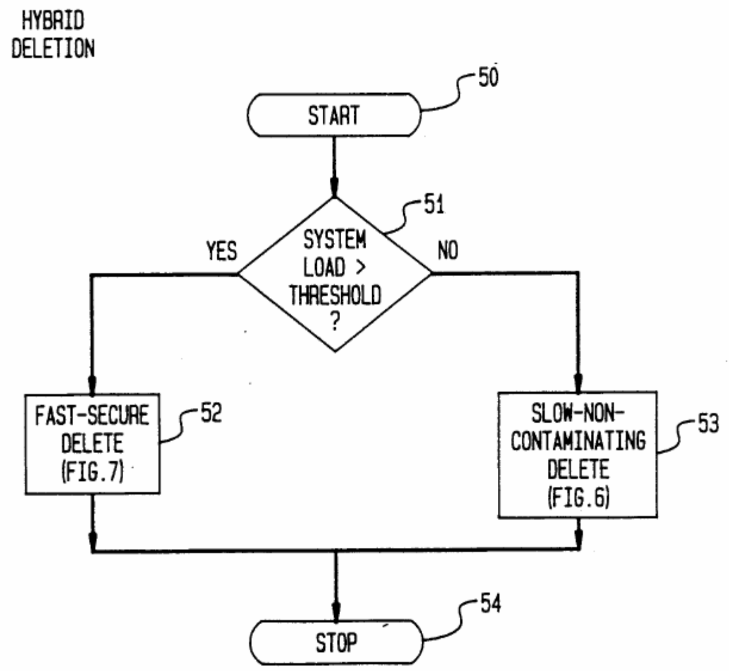
Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Frakes and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Frakes. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Frakes would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) (“Frakes”) alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Frakes and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Frakes with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) (“Frakes”) alone and in combination
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Frakes and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Frakes. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Frakes would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Frakes and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Frakes to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Frakes with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Frakescan be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Frakes in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Frakes. For example, both Linux 2.0.1 and Frakes describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		the function rt_cache_add automatically increments and decrements the variable rt_cache_size, the variable rt_cache_size is determined dynamically. Furthermore, Linux 2.0.1 includes the function rt_garbage_collect_1. The function rt_garbage_collect_1 loops through each of the linked lists in the ip_rt_hash_table global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function rt_garbage_collect_1 accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than the maximum number of records that the function rt_cache_add can remove from a linked list.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, Frakes discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Frakes also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Frakes discloses "hashing, an information storage and retrieval technique useful for implementing many … other structures." (Frakes at 293). Frakes also discloses "hashing, an information storage and retrieval technique useful for implementing many … other structures." (Frakes at 293). Frakes discloses that *hashing* is "a ubiquitous information retrieval strategy for providing efficient access to information based on a key." *Id.* Frakes further discloses "chained hashing. It is so named because each bucket stores a linked list—that is, a chain—of key-information pairs, rather than a single one." (Frakes at 298).
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Frakes discloses accessing a linked list of records. Frakes also discloses accessing a linked list of records having same hash address. For example, Frakes discloses that "[t]he goal (of hashing) is to avoid *collisions*. A collision occurs when two or more keys map to the same location. If no keys collide, then locating the information associated with a key is simply the process of determining the key's location. Whenever a collision occurs, some extra computation is necessary to further determine a unique location for a key." (Frakes at 294).
[3b] identifying at least	[7b] identifying at least	Frakes discloses identifying at least some of the automatically expired ones of

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
some of the automatically expired ones of the records, and	some of the automatically expired ones of the records,	the records. Frakes also discloses identifying at least some of the automatically expired ones of the records. For example, Frakes discloses that a "hash table with *m* buckets may therefore store more than *m* keys. However, performance will degrade as the number of keys increases. Computing the bucket in which a key resides is still fast—a matter of evaluating the hash function—but locating it within that bucket (or simply determining its presence, which is necessary in all operations) requires traversing the linked list." (Frakes at 299).
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Frakes discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. Frakes also discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, Frakes discloses that "performance will degrade as the number of keys increases." (Frakes at 299). Thus, Frakes suggests removing at least some of the automatically expired records from the linked list when the linked list is accessed. (*See id.*)
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	Frakes discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, Frakes discloses several operations that "are usually provided by an implementation of hashing: 1. Initialization: indicate that the hash table contains no elements. 2. Insertion: insert information, indexed by a key *k*, into a hash table. If the has table already contains *k*, then it cannot be inserted. (Some implementations do allow such insertion, to permit replacing existing information.)

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		3. Retrieval: given a key *k*, retrieve the information associated with it. 4. Deletion: remove the information associated with key *k* from a hash table, if any exists. New information indexed by *k* may subsequently be placed in the table." (Frakes at 297). Frakes further discloses that "[m]ost of the code from the routines `Insert`, `Delete`, `Clear` (for `Initialize`), and `Member` (for `Retrieve`) can be used directly." (Frakes at 299).
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Frakes to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Frakes with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Frakes can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Frakes is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) (“Frakes”) alone and in combination**
		“[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Frakes combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in the chart of Dirks, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Frakes and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as that described Frakes. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Frakes would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Frakes and would have

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps. Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Frakes with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Frakes with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove. Further, one of ordinary skill in the art would be motivated to combine Frakes with Thatte and recognized the benefits of doing so. For example, the removal of expired records described in Frakescan be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Frakes with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Frakes with Thatte. Alternatively, it would also be obvious to combine Frakes with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-12

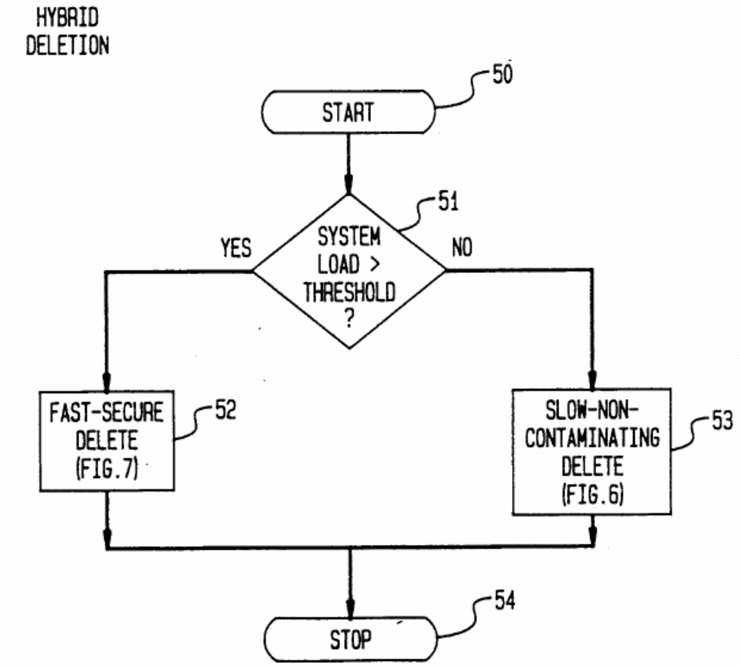
Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		FIG.5  HYBRID DELETION 50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Frakes and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as Frakes. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Frakes would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Frakes and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Frakes with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Frakes and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Frakes. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Frakes would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Frakes and would have seen the benefits of doing so. One such benefit, for example, is that the system would only perform deletions when the system was not already too overloaded, thus preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Frakes to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Frakes with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Frakes can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Frakes in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Frakes. For example, both Linux 2.0.1 and Frakes describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function

EXHIBIT C-12

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		`rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the

Asserted Claims From U.S. Pat. No. 5,893,120		**William B. Frakes & Ricardo Baeza-Yates, *Information Retrieval: Data Structures & Algorithms* (Prentice-Hall, Inc. 1992) ("Frakes") alone and in combination**
		predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Brown discloses an information storage and retrieval system. For example, Brown discloses an information storage and retrieval system made up of a hash table of linked lists. *See*, *e.g.*, Brown at 60-62, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Brown discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Brown also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. S*ee, e.g.*, Brown at 33-34, 60-62, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5. Brown discloses in part: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection." *See* Brown at 33-34.
[1b] a record search means	[5b] a record search means	Brown discloses a record search means utilizing a search key to access the

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
utilizing a search key to access the linked list,	utilizing a search key to access a linked list of records having the same hash address,	linked list. Brown also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. *See, e.g.*, Brown at 60-62, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	Brown discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. Brown also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. *See, e.g.*, Brown at 33, 60-62, 66-68, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5. Brown further discloses: "The value array and the key heap grow towards each other, such that the maximum number of entries in a bucket is variable. The array and heap entries are paired-up from inside out, eliminating the need for string heap offsets in the value array entries and minimizing the amount of space required by the key/value pairs (compression techniques excluded). The

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		tradeoff is a more complex bucket and search algorithm. To find a key/value pair in a bucket, we must scan the bucket's key heap from left to right . . . ." *See* Brown at 61. Brown further discloses: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection." *See* Brown at 33-34.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Brown discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Brown also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. *See, e.g.*, Brown at 33-34, 60-62, 66-68, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5. In addition, Brown discloses: "The value array and the key heap grow towards each other, such that the maximum number of entries in a bucket is variable. The array and heap entries are paired-up from inside out, eliminating the need for string heap offsets in the value array entries and minimizing the amount of space required by the key/value pairs (compression techniques excluded). The tradeoff is a more complex bucket and search algorithm. To find a key/value

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		pair in a bucket, we must scan the bucket's key heap from left to right . . . ." *See* Brown at 61. Brown further discloses: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to further document addition operations." *See* Brown at 33-34. Brown further discloses: "In the third approach, all of the inverted lists in the inverted file are scanned and entries for the deleted document are removed from inverted lists as they are found . . . . The scan of the inverted file is driven at the object level and is supported by Mneme's object scanning facility. This facility allows an object pool to iterate through its objects in order of object identifier." *See* Brown at 67.
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed	It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Brown to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Brown with the fundamental concept of dynamically determining the maximum number of

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
linked list of records.	linked list of records.	expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Brown can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Brown is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Brown combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Brown and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Brown. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

EXHIBIT C-13

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995)** **(hereinafter "Brown") alone and in combination**
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Brown nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Brown and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Brown with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Brown with Thatte would be nothing more than the

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove. Further, one of ordinary skill in the art would be motivated to combine Brown with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Brown can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Brown with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Brown with Thatte. Alternatively, it would also be obvious to combine Brown with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record

<u>EXHIBIT C-13</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-13

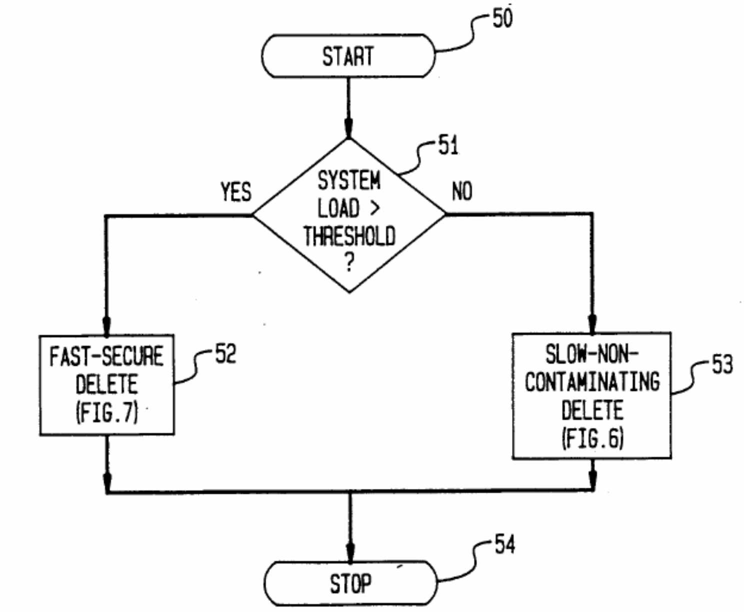
Asserted Claims From **U.S. Pat. No. 5,893,120**		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33,

EXHIBIT C-13

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995)** **(hereinafter “Brown”) alone and in combination**
		Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Brown and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Brown. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Brown would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Brown and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Brown with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, Brown discloses in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Brown and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Brown. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Brown would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Brown and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Brown to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Brown with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Browncan be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Brown in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Brown. For example, both Linux 2.0.1 and Brown describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter “Brown”) alone and in combination**
		dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access	7. A method for storing and retrieving information records using a hashing technique to provide access	To the extent the preamble is a limitation, Brown discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Brown also discloses a method for storing and retrieving information

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
to the records, at least some of the records automatically expiring, the method comprising the steps of:	to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. *See*, *e.g.*, Brown at 33-34, 60-62, 66-68, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5. Brown further discloses: "The value array and the key heap grow towards each other, such that the maximum number of entries in a bucket is variable. The array and heap entries are paired-up from inside out, eliminating the need for string heap offsets in the value array entries and minimizing the amount of space required by the key/value pairs (compression techniques excluded). The tradeoff is a more complex bucket and search algorithm. To find a key/value pair in a bucket, we must scan the bucket's key heap from left to right . . . ." *See* Brown at 61. In addition, Brown discloses: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		further document addition operations." *See* Brown at 33-34.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Brown discloses accessing a linked list of records. Brown also discloses accessing a linked list of records having same hash address. *See, e.g.,* Brown at 33-34, 60-62, 66-68, Fig. 3.5. For example, Brown discloses in part: "We have implemented a Mneme-based hash table for our inverted File Manager using the overall structure shown in Figure 3.5 . . . . Each slot points to a linked list of buckets, which contain the key/value pairs for the keys that hash to that slot." *See* Brown at 60-62, Fig. 3.5. Brown further discloses: "The value array and the key heap grow towards each other, such that the maximum number of entries in a bucket is variable. The array and heap entries are paired-up from inside out, eliminating the need for string heap offsets in the value array entries and minimizing the amount of space required by the key/value pairs (compression techniques excluded). The tradeoff is a more complex bucket and search algorithm. To find a key/value pair in a bucket, we must scan the bucket's key heap from left to right . . . ." Brown also discloses for example: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		further document addition operations." *See* Brown at 33-34. Brown further discloses: "In the third approach, all of the inverted lists in the inverted file are scanned and entries for the deleted document are removed from inverted lists as they are found . . . . The scan of the inverted file is driven at the object level and is supported by Mneme's object scanning facility. This facility allows an object pool to iterate through its objects in order of object identifier." *See* Brown at 67.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Brown discloses identifying at least some of the automatically expired ones of the records. *See, e.g.*, Brown at 33-34, 60-62, 66-68, Fig. 3.5. For example, Brown discloses in part: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to further document addition operations." *See* Brown at 33-34. Brown further discloses: "In the third approach, all of the inverted lists in the inverted file are scanned and entries for the deleted document are removed from inverted lists as they are found . . . . The scan of the inverted file is driven at the object level and is supported by Mneme's object scanning facility. This facility allows an object pool to iterate through its objects in order of object

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		identifier." *See* Brown at 67.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Brown discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. *See, e.g.*, Brown at 33-34, 60-62, 66-68, Fig. 3.5. For example, Brown discloses in part: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to further document addition operations." *See* Brown at 33-34. Brown further discloses: "In the third approach, all of the inverted lists in the inverted file are scanned and entries for the deleted document are removed from inverted lists as they are found . . . . The scan of the inverted file is driven at the object level and is supported by Mneme's object scanning facility. This facility allows an object pool to iterate through its objects in order of object identifier." *See* Brown at 67.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of	Brown discloses inserting, retrieving or deleting one of the records from the system following the step of removing. *See, e.g.*, Brown at 33-34, 60-62, 66-68, Fig. 3.5.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
	removing.	For example, Brown discloses in part: "The ability to modify an existing document collection is a natural requirement for any information retrieval system . . . . Additionally, old news articles will eventually expire and must be deleted from the current events document collection. Articles may expire either because their content is relevant only for a certain period of time, or because the size of the current events collection must be held below some threshold due to performance requirements or capacity limitations. Expired articles will either be discarded or archived in a larger secondary document collection, leading to further document addition operations." *See* Brown at 33-34. Brown further discloses: "In the third approach, all of the inverted lists in the inverted file are scanned and entries for the deleted document are removed from inverted lists as they are found . . . . The scan of the inverted file is driven at the object level and is supported by Mneme's object scanning facility. This facility allows an object pool to iterate through its objects in order of object identifier." *See* Brown at 67. In addition, Brown discloses: "Should all of the document entries be deleted from an inverted list, the list's object can be freed and the corresponding term can be deleted from the term hash table . . . . The long inverted lists are processed next. The page-object pool that contains the link list object is scanned, giving us the first object of each long inverted list." *See* Brown at 67-68.
4. The method according to claim 3 further including the step of dynamically	8. The method according to claim 7 further including the step of dynamically	It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Brown to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
determining maximum number of expired ones of the records to remove when the linked list is accessed.	determining maximum number of expired ones of the records to remove when the linked list is accessed.	fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Brown with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Brown can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Brown is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Brown combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more

EXHIBIT C-13

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in the chart of Dirks, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are

EXHIBIT C-13

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter “Brown”) alone and in combination**
		transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT$_i$ on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Brown and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		implementations such as that described Brown. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Brown would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Brown and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps. Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Brown with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter “Brown”) alone and in combination**
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Brown with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove. Further, one of ordinary skill in the art would be motivated to combine Brown with Thatte and recognized the benefits of doing so. For example, the removal of expired records described in Brown can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Brown with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Brown with Thatte. Alternatively, it would also be obvious to combine Brown with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination
		excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-13

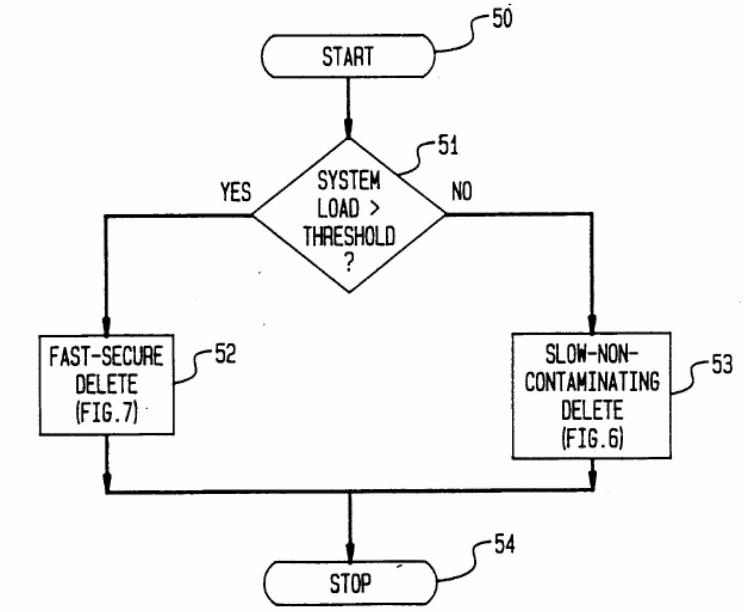
Asserted Claims From **U.S. Pat. No. 5,893,120**		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33,

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Brown and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as Brown. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Brown would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter “Brown”) alone and in combination**
		based on a systems load as taught by the ’663 patent and with Brown and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Brown with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, Brown discloses in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Brown and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Brown. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Brown would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Brown and would have seen the benefits of doing so. One such benefit, for example, is that the system would only perform deletions when the system was not already too overloaded, thus preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Brown to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Brown with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Brown can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Brown in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Brown. For example, both Linux 2.0.1 and Brown describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter “Brown”) alone and in combination**
		the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the

EXHIBIT C-13

Asserted Claims From U.S. Pat. No. 5,893,120		**Eric W. Brown, *Execution Performance Issues in Full Text Information Retrieval*, University of Massachusetts Amherst (October 1995) (hereinafter "Brown") alone and in combination**
		function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Costello discloses an information storage and retrieval system. For example, Costello describes a method and system for storing and retrieving callouts. *See, e.g.*, Costello, page 3-4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello describes in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a `callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello also describes: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Costello discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Costello also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Costello describes storing outstanding callouts in a linked list. *See, e.g.*, Costello, page 3, 7. The entries are removed when the callout is expired. *Id.* *See also*, Costello Presentation, page 3, 15-18, and 26.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		For example, Costello describes storing the callouts in both a circular array of linked lists and a hash table of linked lists. *See, e.g.*, *Costello*, page 3. *Id.* at 4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a `callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4. Costello discloses in part: "At first, we used a closed-chaining hash table." *See*, Costello, page 4. Costello further discloses: "We wanted both sorts of calls to have equal costs in the new implementation as well, so we switched to an open-chaining hash table, in which only one bucket needs to be searched in any case." *See*, Costello, page 7.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same	Costello discloses a record search means utilizing a search key to access the linked list. Costello also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
	hash address,	For example, Costello describes using both a circular array and a hash table. *See, e.g.*, *Costello*, page 3, 7. In either case, a search key is utilized to access the linked list of callouts. *Id.* at 4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a `callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4. Costello further recites: "At first, we used a closed-chaining hash table." *See*, Costello, page 4. Costello further recites: "We wanted both sorts of calls to have equal costs in the new implementation as well, so we switched to an open-chaining hash table, in which only one bucket needs to be searched in any case." *See*, Costello, page 7.
[1c] the record search means including a means for identifying and removing at least some of	[5c] the record search means including means for identifying and removing at least some expired ones	Costello discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. Costello also discloses the record search means including means for identifying and removing at least some expired

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
the expired ones of the records from the linked list when the linked list is accessed, and	of the records from the linked list of records when the linked list is accessed, and	ones of the records from the linked list of records when the linked list is accessed. For example, Costello describes identifying the expired callouts and removing them from the linked list when it is accessed. *See, e.g.*, Costello, page 3-4, 7. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a `callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses that: "If the first callout has expired, a software clock interrupt is generated. Its handler, `softclock()`, repeatedly checks the callout at the head of the list, and if it is expired (`c_time` member equal to zero), removes it and calls its function." *See*, Costello, page 3. In addition, Costello discloses: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4. Costello further discloses: "At first, we used a closed-chaining hash table." *See*, Costello, page 4. Costello further discloses: "We wanted both sorts of calls to have equal costs in the new implementation as well, so we switched to an open-chaining hash table,

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		in which only one bucket needs to be searched in any case." *See*, Costello, page 7.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Costello discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Costello also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, Costello describes using the record search means to access the linked list and retrieving and removing expired callouts from the linked list at the same time. *See, e.g.*, Costello, page 3-4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a `callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "If the first callout has expired, a software clock interrupt is generated. Its handler, `softclock()`, repeatedly checks the callout at the head of the list, and if it is expired (`c_time` member equal to zero), removes it and calls its function." *See,* Costello, page 3.
2. The information storage	6. The information storage	Costello discloses an information storage and retrieval system further including

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. For example, Costello describes limiting the number of records removed from the linked list during a particular sequence using a max_softclock_steps variable. *See, e.g.*, Costello, page 8. For example, Costello discloses in part: "`softclock()` keeps track of the number of steps it has taken since it last enabled interrupts, and whenever the count reaches `MAX_SOFTCLOCK_STEPS`, it briefly enables them. Therefore, `softclock()` never disables interrupts for more than a constant amount of time." *See, e.g., Costello*, page 8. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Costello to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Costello with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Costello can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Costello is avoiding these problems.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Costello, Costello combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Costello and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Costello. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Costello nothing more than

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Costello and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Costello with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, as both Costello and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Costello with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		Further, one of ordinary skill in the art would be motivated to combine Costello with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Costello can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Costello with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Costello with Thatte. Alternatively, it would also be obvious to combine Costello with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent").

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-14

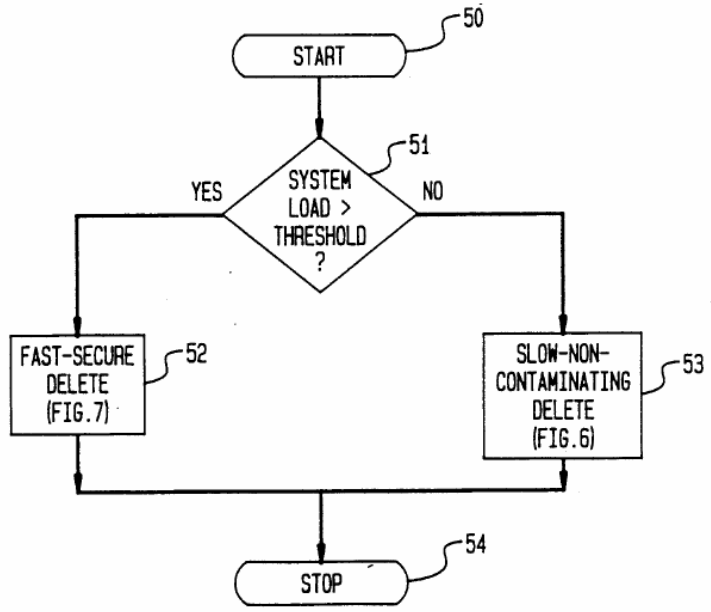
Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33,

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Costello and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Costello. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Costello would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Costello and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Costello with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Costello and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Costello. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Costello would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Costello and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Costello to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Costello with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Costello can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Costello in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Costello. For example, both Linux 2.0.1 and Costello describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an	To the extent the preamble is a limitation, Costello discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Costello also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
some of the records automatically expiring, the method comprising the steps of:	external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Costello describes a method and system for storing and retrieving callouts which expire automatically based on timers. *See, e.g.*, Costello, page 3-4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a `callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Costello discloses accessing a linked list of records. Costello also discloses accessing a linked list of records having same hash address. For example, Costello describes using both a circular array and a hash table. *See, e.g.*, Costello, page 3-4, 7. In either case, a search key is utilized to access the linked list of callouts. *Id.* at 4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		`callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4. Costello further discloses: "At first, we used a closed-chaining hash table." *See*, Costello, page 4. Costello further discloses: "We wanted both sorts of calls to have equal costs in the new implementation as well, so we switched to an open-chaining hash table, in which only one bucket needs to be searched in any case." *See*, Costello, page 7.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Costello discloses identifying at least some of the automatically expired ones of the records. For example, the entries are removed when the callout is expired. *See, e.g.*, Costello, page 3-4, 7. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "If the first callout has expired, a software clock

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		interrupt is generated. Its handler, `softclock()`, repeatedly checks the callout at the head of the list, and if it is expired (`c_time` member equal to zero), removes it and calls its function." *See*, Costello, page 3.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Costello discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, Costello describes identifying the expired callouts and removing them from the linked list when it is accessed. *See, e.g.*, Costello, page 3-4. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "If the first callout has expired, a software clock interrupt is generated. Its handler, `softclock()`, repeatedly checks the callout at the head of the list, and if it is expired (`c_time` member equal to zero), removes it and calls its function." *See*, Costello, page 3.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	Costello discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, Costello describes using the record search means to access the linked list and insert, retrieving, or delete expired callouts from the linked list following the step of removing expired callouts *See, e.g.*, Costello, page 3-4, 7. *See also*, Costello Presentation, page 3, 15-18, and 26. For example, Costello discloses in part: "Instead of a single sorted list of callout structures, we use a circular array of unsorted lists. The array, called a

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		`callwheel` (see Figure 2), contains `callwheelsize` entries. All callouts scheduled to expire at time `t` appear in the list `callwheel[t% callwheelsize]`, and their `c_time` members are set to `t/callwheelsize`." *See*, Costello, page 3. Costello further discloses: "If the first callout has expired, a software clock interrupt is generated. Its handler, `softclock()`, repeatedly checks the callout at the head of the list, and if it is expired (`c_time` member equal to zero), removes it and calls its function." *See*, Costell, page 3. Costello further discloses: "We could try to find a way, given a function pointer and argument pointer, to produce a pointer to the matching callout in constant time. A hash table is the obvious mechanism." *See*, Costello, page 4. In addition, Costello discloses: "At first, we used a closed-chaining hash table." *See*, Costello, page 4. Costello further discloses: "We wanted both sorts of calls to have equal costs in the new implementation as well, so we switched to an open-chaining hash table, in which only one bucket needs to be searched in any case." *See*, Costello, page 7.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is	Costello discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, Costello describes limiting the number of records removed from the linked list during a particular sequence using a max_softclock_steps variable. *See, e.g.*, Costello, page 8.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
accessed.	accessed.	For example, Costello discloses in part: "`softclock()` keeps track of the number of steps it has taken since it last enabled interrupts, and whenever the count reaches `MAX_SOFTCLOCK_STEPS`, it briefly enables them. Therefore, `softclock()` never disables interrupts for more than a constant amount of time." *See*, Costello, page 8. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Costello to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Costello with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Costello can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Costello is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Costello, Costello combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Costello and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Costello. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with Costello nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Costello and would have seen the benefits of doing so. One possible benefit, for example, is

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Costello with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, as both Costello and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Costello with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Costello with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Costello can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Costello with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Costello with Thatte. Alternatively, it would also be obvious to combine Costello with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of

EXHIBIT C-14

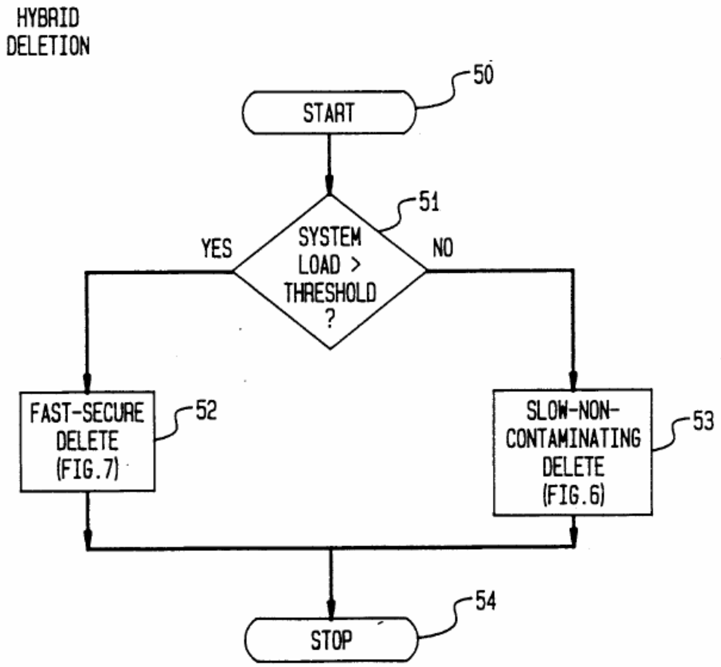
Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		Id. at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Costello and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Costello. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Costello would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Costello and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Costello with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Costello and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Costello. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Costello would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Costello and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Costello to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").
		disclosed in Costello with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Costello can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Costello in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Costello. For example, both Linux 2.0.1 and Costello describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result.

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 ("Costello"). *See also*, Costello, Adam, *et al*., Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 ("Costello Presentation").**
		the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record’s last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record’s reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function

EXHIBIT C-14

Asserted Claims From U.S. Pat. No. 5,893,120		**Costello, Adam, *et al.*, *Redesigning the BSD - Callout and Time Facilities*, WUSC 95-23, November 2, 1995 (“Costello”). *See also*, Costello, Adam, *et al.*, Presentation - *Redesigning the BSD Unix Callout and Time Facilities*, January 10, 1996 (“Costello Presentation”).**
		`rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Foster discloses an information storage and retrieval system. *See, e.g.*, Foster at 4-12, 24-26, 33-40. For example, Foster discloses in part: "A very important use of the vector of lists is in one of the methods for doing 'hash coding'. The problem is to find something which has been associated with an object, on being presented with the object itself." *See* Foster at 25. Foster further discloses: "When a name is read, the appropriate list is selected and searched. If 26 is a suitable value for n and the unevenness of distribution of initial letters is not thought to matter, then these could serve as the index for the lists." *See* Foster at 25.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Foster discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Foster also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. *See, e.g.*, Foster at 4-12, 33-40. For example, Foster discloses in part: "A very important use of the vector of lists is in one of the methods for doing 'hash coding'. The problem is to find something which has been associated with an object, on being presented with the object itself." *See* Foster at 25. Foster further discloses: "When a name is read, the appropriate list is selected and searched. If 26 is a suitable value for *n* and the unevenness of distribution of initial letters is not thought to matter, then these could serve as the index for the lists." *See* Foster at 25.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		In addition, Foster discloses for example: "But most list processing problems will require more store than can be made available in this straightforward manner, and something has to be done to enable the re-use of stores of which the contents are no longer needed. . . . All list processing languages provide, either explicitly in the language or implicitly in the system, some method of reclaiming the store." *See* Foster at 33.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Foster discloses a record search means utilizing a search key to access the linked list. Foster also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. *See, e.g.*, Foster at 24-26, 33-38. For example, Foster discloses in part: "A very important use of the vector of lists is in one of the methods for doing 'hash coding'. The problem is to find something which has been associated with an object, on being presented with the object itself." *See* Foster at 25. Foster further discloses: "When a name is read, the appropriate list is selected and searched. If 26 is a suitable value for *n* and the unevenness of distribution of initial letters is not thought to matter, then these could serve as the index for the lists." *See* Foster at 25.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	Foster discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. Foster also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. *See, e.g.*, Foster at 35-38. For example, Foster discloses in part: "A very important use of the vector of

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination
		lists is in one of the methods for doing ‘hash coding’. The problem is to find something which has been associated with an object, on being presented with the object itself.” *See* Foster at 25. Foster further discloses in part: “When a name is read, the appropriate list is selected and searched. If 26 is a suitable value for *n* and the unevenness of distribution of initial letters is not thought to matter, then these could serve as the index for the lists.” *See* Foster at 25. For example, Foster also discloses: “But most list processing problems will require more store than can be made available in this straightforward manner, and something has to be done to enable the re-use of stores of which the contents are no longer needed. . . . All list processing languages provide, either explicitly in the language or implicitly in the system, some method of reclaiming the store.” *See* Foster at 33. In addition, Foster discloses: “A convention, which has been adopted in order to make the memory of the wanted cells easier to the user, is to say that a list is owned in one place only. If it appears in other places it is being borrowed. The list that is the owner is responsible for declaring that the cells are not wanted.” *See* Foster at 35.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the	Foster discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Foster also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. *See, e.g.*, Foster at 33-40.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. Foster, List Processing (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
	records in the accessed linked list of records.	For example, Foster discloses in part: "But most list processing problems will require more store than can be made available in this straightforward manner, and something has to be done to enable the re-use of stores of which the contents are no longer needed. . . . All list processing languages provide, either explicitly in the language or implicitly in the system, some method of reclaiming the store." *See* Foster at 33. Foster further discloses: "A convention, which has been adopted in order to make the memory of the wanted cells easier to the user, is to say that a list is owned in one place only. If it appears in other places it is being borrowed. The list that is the owner is responsible for declaring that the cells are not wanted." *See* Foster at 35. Foster also discloses, for example: "Hence the process of garbage collection has to proceed in two stages, the first of which goes through all of the lists dependent on the list heads and marks them as wanted. The second then scans the whole area allotted to lists and returns the unmarked cells to the free list, simultaneously unmarking the marked cells ready for next time." *See* Foster at 35.
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Foster to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Foster with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. Foster, List Processing (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		number of potential problems. For example, the removal of expired records described in Foster can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Foster is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Foster combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks,

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination**
		$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Foster and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Foster. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. Foster, List Processing (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Foster nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Foster and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Foster with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Foster and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Foster with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number

EXHIBIT C-15

Asserted Claims From **U.S. Pat. No. 5,893,120**		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination**
		for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Foster with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Foster can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Foster with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Foster with Thatte. Alternatively, it would also be obvious to combine Foster with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”).

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-15

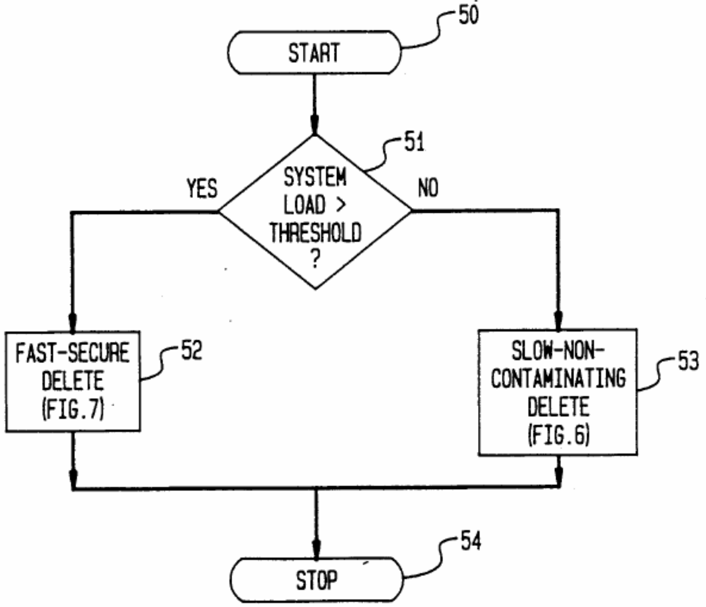
Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Foster and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Foster. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Foster would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Foster and

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination**
		would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Foster with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. Foster, List Processing (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination**
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Foster and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Foster. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with Foster would be nothing more than the predictable use of prior

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Foster and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Foster to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Foster with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Foster can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Foster in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Foster. For example, both Linux 2.0.1 and Foster describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1`

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination**
		function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record’s last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record’s reference count is zero. *See* Linux

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, Foster discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Foster also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. *See, e.g.,* Foster at 4-12, 24-26, and 33-40. For example, Foster discloses in part: "A very important use of the vector of lists is in one of the methods for doing 'hash coding'. The problem is to find something which has been associated with an object, on being presented with the object itself." *See* Foster at 25.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. Foster, List Processing (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		Foster further discloses, for example: "When a name is read, the appropriate list is selected and searched. If 26 is a suitable value for n and the unevenness of distribution of initial letters is not thought to matter, then these could serve as the index for the lists." *See* Foster at 25. Additional, Foster discloses in part: "But most list processing problems will require more store than can be made available in this straightforward manner, and something has to be done to enable the re-use of stores of which the contents are no longer needed. . . . All list processing languages provide, either explicitly in the language or implicitly in the system, some method of reclaiming the store." *See* Foster at 33. Foster further discloses: "A convention, which has been adopted in order to make the memory of the wanted cells easier to the user, is to say that a list is owned in one place only. If it appears in other places it is being borrowed. The list that is the owner is responsible for declaring that the cells are not wanted." *See* Foster at 35.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Foster discloses accessing a linked list of records. Foster also discloses accessing a linked list of records having same hash address. *See, e.g.*, Foster at 4-12, 24-26. For example, Foster discloses in part: "A very important use of the vector of lists is in one of the methods for doing 'hash coding'. The problem is to find something which has been associated with an object, on being presented with the object itself." *See* Foster at 25. Foster further discloses, for example: "When a name is read, the appropriate

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		list is selected and searched. If 26 is a suitable value for *n* and the unevenness of distribution of initial letters is not thought to matter, then these could serve as the index for the lists." *See* Foster at 25.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Foster discloses identifying at least some of the automatically expired ones of the records. *See* Foster at 35-38. For example, Foster discloses in part: "But most list processing problems will require more store than can be made available in this straightforward manner, and something has to be done to enable the re-use of stores of which the contents are no longer needed. . . . All list processing languages provide, either explicitly in the language or implicitly in the system, some method of reclaiming the store." *See* Foster at 33. Foster also discloses for example: "A convention, which has been adopted in order to make the memory of the wanted cells easier to the user, is to say that a list is owned in one place only. If it appears in other places it is being borrowed. The list that is the owner is responsible for declaring that the cells are not wanted." *See* Foster at 35. Foster also discloses: "Hence the process of garbage collection has to proceed in two stages, the first of which goes through all of the lists dependent on the list heads and marks them as wanted. The second then scans the whole area allotted to lists and returns the unmarked cells to the free list, simultaneously unmarking the marked cells ready for next time." *See* Foster at 35.
[3c] removing at least some of the automatically expired records from the	[7c] removing at least some of the automatically expired records from the	Foster discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. *See* Foster at 35-38.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
linked list when the linked list is accessed.	linked list when the linked list is accessed, and	For example, Foster discloses in part: "Hence the process of garbage collection has to proceed in two stages, the first of which goes through all of the lists dependent on the list heads and marks them as wanted. The second then scans the whole area allotted to lists and returns the unmarked cells to the free list, simultaneously unmarking the marked cells ready for next time." *See* Foster at 35.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	Foster discloses inserting, retrieving or deleting one of the records from the system following the step of removing. *See* Foster at 4-12, 22-29, 33-40. For example, Foster discloses in part: "Hence the process of garbage collection has to proceed in two stages, the first of which goes through all of the lists dependent on the list heads and marks them as wanted. The second then scans the whole area allotted to lists and returns the unmarked cells to the free list, simultaneously unmarking the marked cells ready for next time." *See* Foster at 35. It would be obvious to a person of skill in the art to perform known functions such as an insertion, deletion, or retrieval on the linked list after the step of removing is performed.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Foster to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Foster with the fundamental concept of dynamically determining the maximum

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination
		number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Foster can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Foster is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Foster combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks,

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		$$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Foster and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Foster. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Foster nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Foster and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Foster with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Foster and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Foster with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number

EXHIBIT C-15

Asserted Claims From **U.S. Pat. No. 5,893,120**		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination**
		for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Foster with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Foster can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Foster with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Foster with Thatte. Alternatively, it would also be obvious to combine Foster with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”).

EXHIBIT C-15

Asserted Claims From **U.S. Pat. No. 5,893,120**		**J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination**
		In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-15

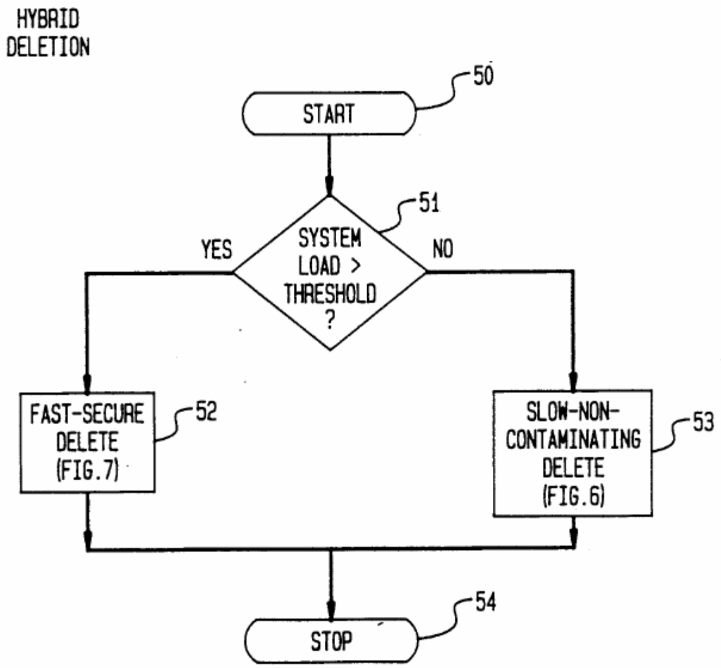
Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter “Foster”) alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Foster and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Foster. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Foster would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Foster and

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Foster with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Foster and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Foster. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Foster would be nothing more than the predictable use of prior

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Foster and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Foster to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Foster with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Foster can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Foster in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Foster. For example, both Linux 2.0.1 and Foster describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1`

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux

EXHIBIT C-15

Asserted Claims From U.S. Pat. No. 5,893,120		J.M. FOSTER, LIST PROCESSING (Macdonald & Co. 1967) (hereinafter "Foster") alone and in combination
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation Keshav discloses an information storage and retrieval system. For example, Keshav discloses in part: “The algorithm is implemented at the server that schedules packets on the output trunk of a router or switch in a store-and-forward network.” Srinivasan Keshav, *On the Efficient Implementation of Fair Queuing* (hereinafter “Keshav”) at 2.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Keshav discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Keshav also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Keshav discloses in part: “Buffering Alternatives We considered four buffering schemes: an ordered linked list (LINK), a binary tree (TREE), a double heap (HEAP), and a combination of per-conversation queuing and heaps (PERC). We expect that the reader is familiar with details of the list, tree and heap data structures. They are also described in standard texts such as References [10, 11]. Ordered List Tag values usually increase with time, since bid numbers are strictly monotonic within each conversation. This suggests that packets should be

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		buffered in a ordered linked list, inserting incoming packets by linearly scanning from the largest tag value.” Keshav at 8. “If all the buffers are full, the server drops the packet with the largest bid number (unlike the algorithm in Reference [1], this buffer allocation policy accounts for differences in packet lengths). The abstract data structure required for packet buffering is a bounded heap. A bounded heap is named by its root, and contains a set of packets that are tagged by their bid number.” Keshav at 7. “The conversation ID is used to access a data structure for storing state. Since IDs could span large address spaces, the standard solution is to hash the ID onto a index, and the technology for this is well known [9]. Recently, a simple and efficient hashing scheme that ignores hash collections has been proposed [5]. In this approach, some conversations could share the same state, leading to unfair service, since these conversations are served first-come-first-served. However, this is attenuated by occasionally perturbing the hash function.” Keshav at 5. “Assume for the moment that data from each source destination pair (a conversation) can be distinguished, and is stored in a logically distinct per-conversation queue.” Keshav at 2.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Keshav discloses a record search means utilizing a search key to access the linked list. Keshav also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, Keshav discloses in part:

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		"Assume for the moment that data from each source destination pair (a conversation) can be distinguished, and is stored in a logically distinct per-conversation queue." Keshav at 2. The paper discloses in detail how to generate this key in a unique and coherent manner. *See* page 4: "The choice of the conversation ID depends on the entity to whom fair service is granted (see the discussion in Reference [1]), and the naming space of the network. For example, if the unit is a transport connection in the IP Internet, one such unique identifier is the tuple (source address, destination address, source port number, destination port number, protocol type)." Keshav at 4. "The conversation ID is used to access a data structure for storing state. Since IDs could span large address spaces, the standard solution is to hash the ID onto a index, and the technology for this is well known [9]. Recently, a simple and efficient hashing scheme that ignores hash collections has been proposed [5]. In this approach, some conversations could share the same state, leading to unfair service, since these conversations are served first-come-first-served. However, this is attenuated by occasionally perturbing the hash function." Keshav at 5.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	Keshav discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. Keshav also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, Keshav discloses in part:

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		"Assume for the moment that data from each source destination pair (a conversation) can be distinguished, and is stored in a logically distinct per-conversation queue." Keshav at 2. "The choice of the conversation ID depends on the entity to whom fair service is granted (see the discussion in Reference [1]), and the naming space of the network. For example, if the unit is a transport connection in the IP Internet, one such unique identifier is the tuple (source address, destination address, source port number, destination port number, protocol type)." Keshav at 4. "The conversation ID is used to access a data structure for storing state." Keshav at 5. "insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to handle this case if the item is already in the heap. To allow this, we always keep enough free apace in the buffer to accommodate a maximum sized packet." Keshav at 7.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Keshav discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Keshav also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, Keshav discloses in part:

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		"insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to handle this case if the item is already in the heap. To allow this, we always keep enough free [s]pace in the buffer to accommodate a maximum sized packet." Keshav at 7.
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	Keshav discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. For example, Keshav discloses in part: "insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to hand this case if the item is already in the heap. To allow this, we always keep enough free [s]pace in the buffer to accommodate a maximum sized packet, get_min ( ) returns a pointer to the item with the smallest tag value and deletes it." Keshav at 7. It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Keshav to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Keshav with the fundamental concept of dynamically determining the maximum number of

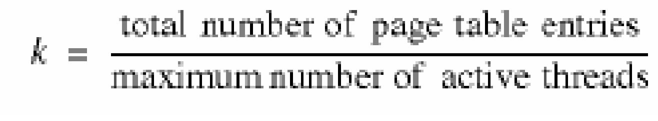
EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Keshav can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Keshav is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Keshav combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		 $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Keshav and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Keshav. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

<u>EXHIBIT C-16</u>

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with Keshav nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Keshav and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Keshav with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Keshav and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Keshav with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Keshav with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Keshav can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Keshav with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Keshav with Thatte. Alternatively, it would also be obvious to combine Keshav with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent

<u>EXHIBIT C-16</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-16

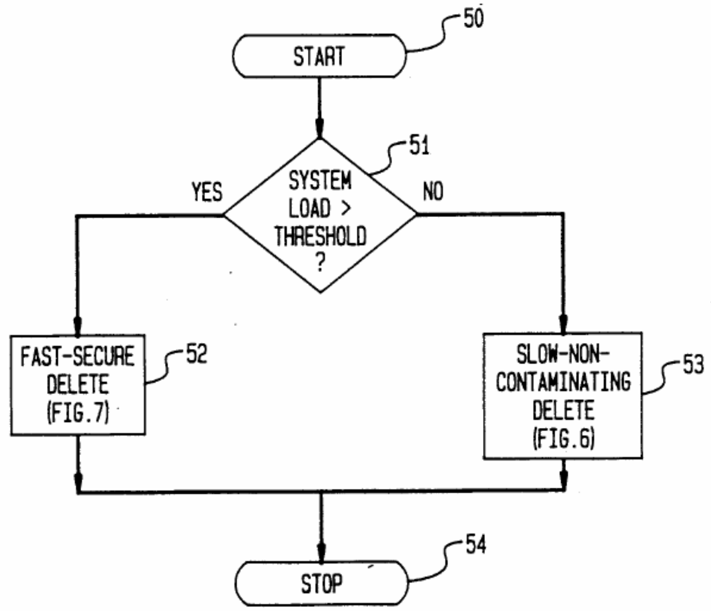
Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination
		FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33,

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Keshav and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Keshav. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Keshav would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		based on a systems load as taught by the '663 patent and with Keshav and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Keshav with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Keshav and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Keshav. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision

EXHIBIT C-16

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		procedure with Keshav would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Keshav and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Keshav to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Keshav with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Keshav can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Keshav in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Keshav. For example, both Linux 2.0.1 and Keshav describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the	To the extent the preamble is a limitation, Keshav discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Keshav also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Keshav discloses in part:

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
	method comprising the steps of:	"The algorithm is implemented at the server that schedules packets on the output trunk of a router or switch in a store-and-forward network." Keshav at 2. "Buffering Alternatives We considered four buffering schemes: an ordered linked list (LINK), a binary tree (TREE), a double heap (HEAP), and a combination of per-conversation queuing and heaps (PERC). We expect that the reader is familiar with details of the list, tree and heap data structures. They are also described in standard texts such as References [10, 11]. Ordered List Tag values usually increase with time, since bid numbers are strictly monotonic within each conversation. This suggests that packets should be buffered in a ordered linked list, inserting incoming packets by linearly scanning from the largest tag value." Keshav at 8. "If all the buffers are full, the server drops the packet with the largest bid number (unlike the algorithm in Reference [1], this buffer allocation policy accounts for differences in packet lengths). The abstract data structure required for packet buffering is a bounded heap. A bounded heap is named by its root, and contains a set of packets that are tagged by their bid number." Keshav at 7. "The conversation ID is used to access a data structure for storing state. Since IDs could span large address spaces, the standard solution is to hash the ID onto a index, and the technology for this is well known [9]. Recently, a simple and efficient hashing scheme that ignores hash collections has been proposed [5]. In this approach, some conversations could share the same state, leading

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		to unfair service, since these conversations are served first-come-first-served. However, this is attenuated by occasionally perturbing the hash function." Keshav at 5. "Assume for the moment that data from each source destination pair (a conversation) can be distinguished, and is stored in a logically distinct per-conversation queue." Keshav at 2. "The choice of the conversation ID depends on the entity to whom fair service is granted (see the discussion in Reference [1]), and the naming space of the network. For example, if the unit is a transport connection in the IP Internet, one such unique identifier is the tuple (source address, destination address, source port number, destination port number, protocol type)." Keshav at 4.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Keshav discloses accessing a linked list of records. Keshav also discloses accessing a linked list of records having same hash address. For example, Keshav discloses in part: "Assume for the moment that data from each source destination pair (a conversation) can be distinguished, and is stored in a logically distinct per-conversation queue." Keshav at 2. "The choice of the conversation ID depends on the entity to whom fair service is granted (see the discussion in Reference [1]), and the naming space of the network. For example, if the unit is a transport connection in the IP Internet, one such unique identifier is the tuple (source address, destination address, source port number, destination port number, protocol type)." Keshav at 4.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		The conversation ID is used as hash key to get to the conversation data records (*i.e.* having the same hash address). *See* page 5, line 9: "The conversation ID is used to access a data structure for storing state. Since IDs could span large address spaces, the standard solution is to hash the ID onto a index, and the technology for this is well known [9]. Recently, a simple and efficient hashing scheme that ignores hash collections has been proposed [5]. In this approach, some conversations could share the same state, leading to unfair service, since these conversations are served first-come-first-served. However, this is attenuated by occasionally perturbing the hash function." Keshav at 5. "Buffering Alternatives We considered four buffering schemes: an ordered linked list (LINK), a binary tree (TREE), a double heap (HEAP), and a combination of per-conversation queuing and heaps (PERC). We expect that the reader is familiar with details of the list, tree and heap data structures. They are also described in standard texts such as References [10, 11]. Ordered List Tag values usually increase with time, since bid numbers are strictly monotonic within each conversation. This suggests that packets should be buffered in a ordered linked list, inserting incoming packets by linearly scanning from the largest tag value." Keshav at 8.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Keshav discloses identifying at least some of the automatically expired ones of the records. Keshav also discloses identifying at least some of the automatically expired ones of the records. For example, Keshav discloses in part:

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination
		"insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to handle this case if the item is already in the heap. To allow this, we always keep enough free [s]pace in the buffer to accommodate a maximum sized packet." Keshav at 7.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Keshav discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. Keshav also discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, Keshav discloses in part: "The abstract data structure required for packet buffering is a bounded heap. A bounded heap is named by its root, and contains a set of packets that are tagged by their bid number. It is associated with two operations, insert (root, item, conversation_ID) and get_min(root), and a parameter, MAX, which is the maximum size of the heap. "insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to handle this case if the item is already in the heap. To allow this, we always keep enough free space in the buffer to accommodate a maximum sized packet." Keshav at 7.
	[7d] inserting, retrieving or deleting one of the records from the system	Keshav discloses inserting, retrieving or deleting one of the records from the system following the step of removing.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
	following the step of removing.	For example, Keshav discloses in part: "insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to handle this case if the item is already in the heap. To allow this, we always keep enough free apace in the buffer to accommodate a maximum sized packet get_min () returns a pointer to the item with the smallest tag value and deletes." Keshav at 7.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Keshav discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, "insert () first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to hand this case if the item is already in the heap. To allow this, we always keep enough free [s]pace in the buffer to accommodate a maximum sized packet, get_min ( ) returns a pointer to the item with the smallest tag value and deletes it." Keshav at 7. It would have been obvious to one of ordinary skill in the art to modify the system disclosed in Keshav to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Keshav with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Keshav can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Keshav is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Keshav combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries

EXHIBIT C-16

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Keshav and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Keshav. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with Keshav nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Keshav and would

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Keshav with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Keshav and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Keshav with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Keshav with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Keshav can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Keshav with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load.

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Keshav with Thatte. Alternatively, it would also be obvious to combine Keshav with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of

EXHIBIT C-16

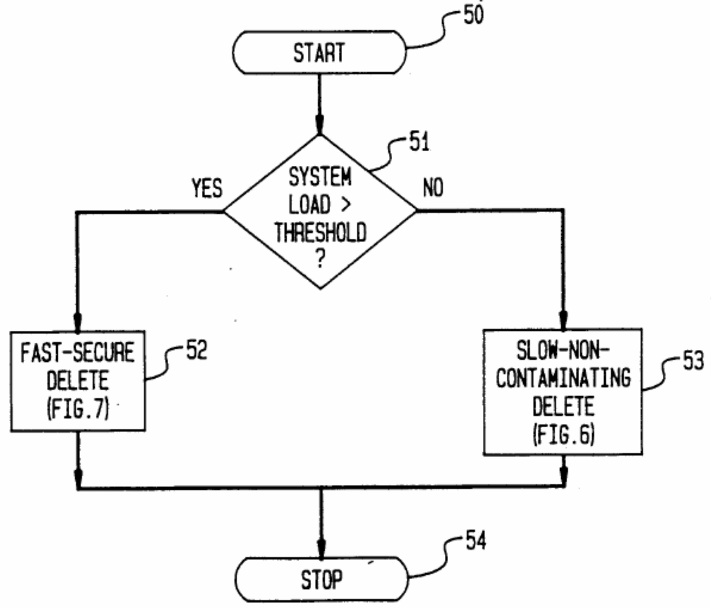
Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination
		automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		Id. at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Keshav and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Keshav. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Keshav would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Keshav and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Keshav with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the

<u>EXHIBIT C-16</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Keshav and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Keshav. Moreover, one of ordinary skill in the art

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with Keshav would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Keshav and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Keshav to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Keshav with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Keshav can be burdensome on the system,

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Keshav in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Keshav. For example, both Linux 2.0.1 and Keshav describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination**
		2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 (“Keshav”) alone and in combination
		line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the

EXHIBIT C-16

Asserted Claims From U.S. Pat. No. 5,893,120		**Sirinivasan Keshav, *On the Efficient Implementation of Fair Queueing*, Journal of Internetworking: Research and Experience, 1991 ("Keshav") alone and in combination**
		predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than the maximum number of records that the function rt_cache_add can remove from a linked list.

EXHIBIT C-16

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Varghese and Lauck discloses an information storage and retrieval system. *See, e.g.*, George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck") at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30. "For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Varghese and Lauck discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Varghese and Lauck also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. *See, e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		For example, Varghese and Lauck disclose in part: "6.1.1 Scheme 5: Hash Table with Sorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "6.1.2 Scheme 6: Hash Table with Unsorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30. "The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30. "PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		always *O*(*n*) (all timers expire at the same time), we believe that the choice of hash function for Scheme 6 is insignificant." *See* Varghese and Lauck at p. 31.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Varghese and Lauck discloses a record search means utilizing a search key to access the linked list. Varghese and Lauck also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. See, e.g., Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "6.1.1 Scheme 5: Hash Table with Sorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "6.1.2 Scheme 6: Hash Table with Unsorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30. "The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30.
[1c] the record search means including a means	[5c] the record search means including means for	Varghese and Lauck discloses the record search means including a means for identifying and removing at least some of the expired ones of the records

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	from the linked list when the linked list is accessed. Varghese and Lauck also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. *See, e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "6.1.1 Scheme 5: Hash Table with Sorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "6.1.2 Scheme 6: Hash Table with Unsorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30. "The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		"PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is always *O*(*n*) (all timers expire at the same time), we believe that the choice of hash function for Scheme 6 is insignificant." *See* Varghese and Lauck at p. 31.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Varghese and Lauck discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Varghese and Lauck also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. *See, e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "6.1.1 Scheme 5: Hash Table with Sorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "6.1.2 Scheme 6: Hash Table with Unsorted Lists in each Bucket" *See* Varghese and Lauck at p. 30.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		"For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30. "The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30. "PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is always *O*(*n*) (all timers expire at the same time), we believe that the choice of hash function for Scheme 6 is insignificant." *See* Varghese and Lauck at p. 31.
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining	Varghese and Lauck discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. *See, e.g.*, Varghese and Lauck at p. 28.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
maximum number for the record search means to remove in the accessed linked list of records.	maximum number for the record search means to remove in the accessed linked list of records.	For example, Varghese and Lauck disclose in part: "The simulation proceeds by processing the earliest event, which in turn may schedule further events. The simulation continues until the event list is empty or some condition (e.g. clock > MAX-SIMULATION-TIME} holds." *See,* Varghese and Lauck at p. 28. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Varghese and Lauck to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Varghese and Lauck with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Varghese and Lauck can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Varghese and Lauck is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired

EXHIBIT C-17

Asserted Claims From **U.S. Pat. No. 5,893,120**		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Varghese and Lauck, Varghese and Lauck combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries

EXHIBIT C-17

Asserted Claims From **U.S. Pat. No. 5,893,120**		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks:

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Varghese and Lauck and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Varghese and Lauck. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Varghese and Lauck nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Varghese and Lauck and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Varghese and Lauck with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, as both Varghese and Lauck and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Varghese and Lauck with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		Further, one of ordinary skill in the art would be motivated to combine Varghese and Lauck with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Varghese and Lauck can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Varghese and Lauck with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Varghese and Lauck with Thatte. Alternatively, it would also be obvious to combine Varghese and Lauck with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-17

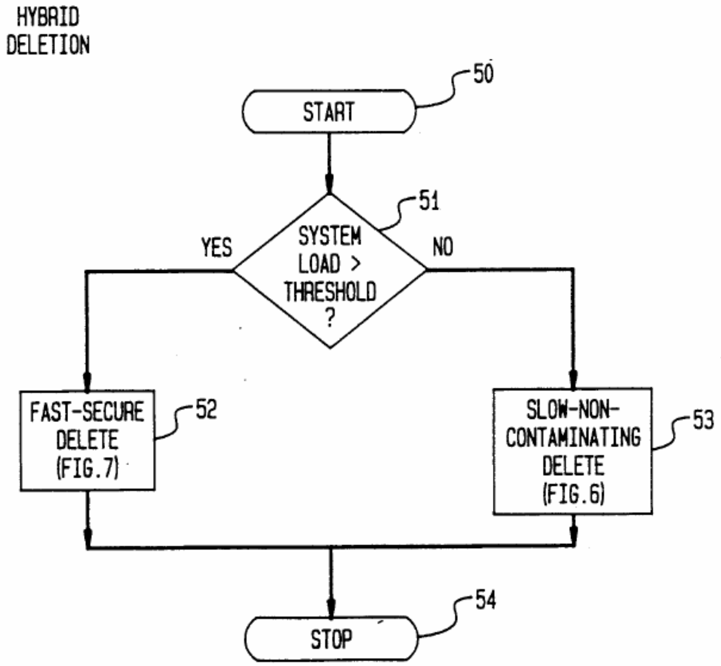
Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Varghese and Lauck and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Varghese and Lauck. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Varghese and Lauck would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Varghese and Lauck and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Varghese and Lauck with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Varghese and Lauck and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Varghese and Lauck. Moreover, one of ordinary skill in the art would recognize that it would improve similar

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Varghese and Lauck would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Varghese and Lauck and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Varghese and Lauck to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Varghese and Lauck with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed

EXHIBIT C-17

Asserted Claims From **U.S. Pat. No. 5,893,120**		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		linked list of records to solve a number of potential problems. For example, the removal of expired records described in Varghese and Lauck can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Varghese and Lauck in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Varghese and Lauck. For example, both Linux 2.0.1 and Varghese and Lauck describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds

EXHIBIT C-17

Asserted Claims From **U.S. Pat. No. 5,893,120**		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		`RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, Varghese and Lauck discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Varghese and Lauck also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. *See*, *e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "6.1.1 Scheme 5: Hash Table with Sorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "6.1.2 Scheme 6: Hash Table with Unsorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		"The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30. "PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is always *O*(*n*) (all timers expire at the same time), we believe that the choice of hash function for Scheme 6 is insignificant." *See* Varghese and Lauck at p. 31.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Varghese and Lauck discloses accessing a linked list of records. Varghese and Lauck also discloses accessing a linked list of records having same hash address. *See, e.g.*, Varghese and Lauck at p. 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "6.1.1 Scheme 5: Hash Table with Sorted Lists in each Bucket" *See* Varghese and Lauck at p. 30. "6.1.2 Scheme 6: Hash Table with Unsorted Lists in each Bucket" *See*

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		Varghese and Lauck at p. 30. "For example, if the table size is a power of 2, an arbitrary size timer can easily be divided by the table size; the remainder (low order bits) is added to the current time pointer to yield the index within the array. The result of the division (high order bits) is stored in a list pointed to by the index." *See* Varghese and Lauck at p. 30. "The previous scheme has an obvious analogy to inserting an element in an array using the element value as an index. If there is insufficient memory we can hash the element value to yield an index." *See* Varghese and Lauck at p. 30.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Varghese and Lauck discloses identifying at least some of the automatically expired ones of the records. *See, e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is always *O*(*n*) (all timers expire at the same time), we believe that the choice of

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		hash function for Scheme 6 is insignificant." *See* Varghese and Lauck at p. 31.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Varghese and Lauck discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. *See, e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is always *O*(*n*) (all timers expire at the same time), we believe that the choice of hash function for Scheme 6 is insignificant. " *See* Varghese and Lauck at p. 31.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	Varghese and Lauck discloses inserting, retrieving or deleting one of the records from the system following the step of removing. *See, e.g.*, Varghese and Lauck at p. 25-27, 29-31, and Figs. 8-9. For example, Varghese and Lauck disclose in part: "PER_TICK_BOOKKEEPING increments the current time pointer. If the value stored in the array element being pointed to is zero, there is no more work. Otherwise, as in Scheme 2, the top of the list is decremented. If it

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		expires, EXPIRY_PROCESSING is called and the top list element is deleted." *See* Varghese and Lauck at p. 30. "Thus the hash distribution in Scheme 6 only controls the "burstiness" (variance) of the latency of PER_TICK_BOOKEEPING, and not the average latency. Since the worst-case latency of PER_TICK_BOOKEEPING is always *O*(*n*) (all timers expire at the same time), we believe that the choice of hash function for Scheme 6 is insignificant. " *See* Varghese and Lauck at p. 31. It would be obvious to one of skill in the art to perform a known function such as inserteing, retrieving, or deleting, following the step of calling EXPIRY_PROCESSING to remove an element from the list.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Varghese and Lauck discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. *See, e.g.*, Varghese and Lauck at p. 28. "The simulation proceeds by processing the earliest event, which in turn may schedule further events. The simulation continues until the event list is empty or some condition (e.g. clock > MAX-SIMULATION-TIME} holds." *See,* Varghese and Lauck at p. 28. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Varghese and Lauck to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		ordinary skill in the art would have been motivated to combine the system disclosed in Varghese and Lauck with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Varghese and Lauck can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in Varghese and Lauck is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Varghese and Lauck, Varghese and Lauck combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT$_i$ on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Varghese and Lauck and Dirks relate to deletion of aged records upon

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Varghese and Lauck. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Varghese and Lauck nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Varghese and Lauck and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Varghese and Lauck with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, as both Varghese and Lauck and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Varghese and Lauck with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Varghese and Lauck with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Varghese and Lauck can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Varghese and Lauck with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Varghese and Lauck with Thatte. Alternatively, it would also be obvious to combine Varghese and Lauck with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The ’663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating

EXHIBIT C-17

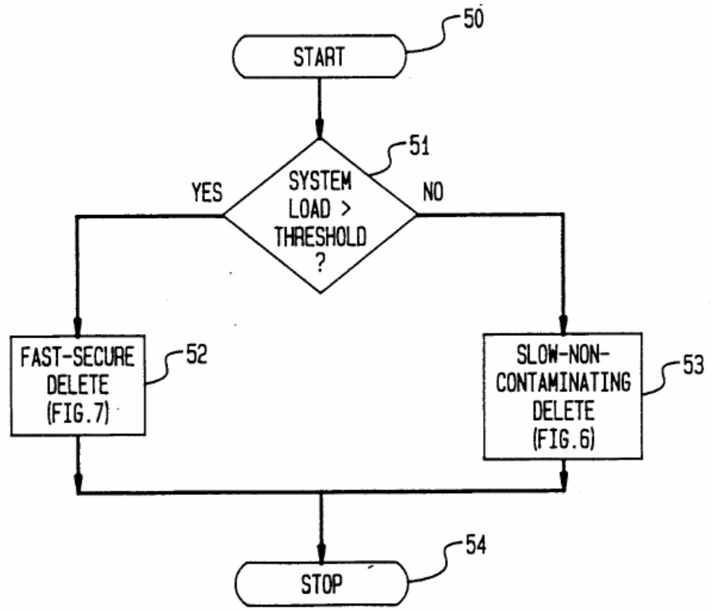
<table>
<tr>
<th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th>
<th>George Varghese and Tony Lauck, <em>Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility</em>, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter “Varghese and Lauck”)</th>
</tr>
<tr>
<td></td>
<td></td>
<td>

> deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46.

This hybrid deletion is shown in Figure 5.

FIG.5

HYBRID DELETION



50
START
51
YES
SYSTEM LOAD > THRESHOLD ?
NO
FAST-SECURE DELETE (FIG.7)
52
SLOW-NON-CONTAMINATING DELETE (FIG.6)
53
54
STOP

Id. at Figure 5.

</td>
</tr>
</table>

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Varghese and Lauck and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Varghese and Lauck. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Varghese and Lauck would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Varghese and Lauck and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Varghese and Lauck with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter “Varghese and Lauck”)**
		scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Varghese and Lauck and the Opportunistic Garbage Collection

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Varghese and Lauck. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Varghese and Lauck would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Varghese and Lauck and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Varghese and Lauck to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")
		art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Varghese and Lauck with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Varghese and Lauck can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Varghese and Lauck in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		with Varghese and Lauck. For example, both Linux 2.0.1 and Varghese and Lauck describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110.

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS OPERATING SYSTEMS REVIEW, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function

EXHIBIT C-17

Asserted Claims From U.S. Pat. No. 5,893,120		**George Varghese and Tony Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, ACM SIGOPS Operating Systems Review, Vol. 21, Issue 5, p. 25-38 (November 1987) (hereinafter "Varghese and Lauck")**
		`rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT C-18

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, *Kruse* discloses an information storage and retrieval system. For example, *Kruse* discloses "[w]hen writing a program, we have had to decide on the maximum amount of memory that would be needed for our arrays and set this aside in the declarations." *Kruse* at 105. *Kruse* also discloses "First, and array must be declared that will hold the hash table. … To insert a record into the hash table, the hash function for the key is first calculated. … To retrieve the record with a given key is entirely similar." *Kruse* at 200.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	*Kruse* discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. *Kruse* also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, *Kruse* discloses that "[t]he idea we use is that of a pointer. A *pointer*, also called a *link* or a *reference*, is defined to be a variable that gives the location of some other variable, typically of a record containing data that we wish to use. If we use pointers to locate all the records in which we are interested, then we need not be concerned about where the records themselves are actually stored, since by using a pointer, we can let the computer system itself locate the record when required." *Kruse* at 105. *Kruse* also discloses that the "idea of a linked list is, for every record in the list, to put a pointer into the record giving the location of the next record in the list." *Kruse* at 106.

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		Additionally, *Kruse* states that "when an item is no longer needed, its space can be returned to the system, which can then assign it to another user." *Kruse* at 107. "If our program is one that continually sets up new nodes and disposes of others, then we shall often find it necessary to set up our own procedures to keep track of nodes that are no longer needed, and to reuse the space when new nodes are later required." *Kruse* at 117. Moreover, *Kruse* discloses "[w]e have already decided to represent our sparse array of cells as a hash table, but we have not yet decided between open addressing and chaining. … Do we need to make deletions, and, if so, when? We could keep track of all cells until the memory is full, and then delete those that are not needed. But this would require rehashing the full array, which would be slow and painful. With chaining we can easily dispose of cells as soon as they are not needed, and thereby reduce the number of cells in the hash table as much as possible." *Kruse* at 216. *Kruse* also discloses, "[i]f we use chaining, then we can add a cell to a list either by inserting the cell itself or a pointer to it, rather than by inserting its coordinates as before. In this way we can locate the cell directly with no need for any search." … "For reasons both of flexibility and time saving, therefore, let us decide to use dynamic memory allocation, a chained hash table, and linked lists." *Kruse* at 217. Finally, *Kruse* discloses that "[i]n using a hash table, let the nature of the data and the required operations help you decide between chaining and open addressing. Chaining is generally preferable if deletions are required, if the records are relatively large, or if overflow might be a problem." *Kruse* at 223.
[1b] a record search means utilizing a search key to	[5b] a record search means utilizing a search key to	*Kruse* discloses a record search means utilizing a search key to access the linked list. *Kruse* also discloses a record search means utilizing a search key to

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
access the linked list,	access a linked list of records having the same hash address,	access a linked list of records having the same hash address. For example, *Kruse* discloses "[t]he idea of a hash table (such as the one shown in Figure 6.10) is to allow many of the different possible keys that might occur to be mapped to the same location in an array under the action of the index function." *Kruse* at 199. *Kruse* also discloses "[t]he task of the procedure is first to look in the hash table for the cell with the given coordinates. If the search is successful, then the procedure returns a pointer to the cell; otherwise, it must create a new cell, assign it the given coordinates, initialize its other fields to the default values, and put it in the hash table as well as return a pointer to it." *Kruse* at 220-21.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	*Kruse* discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. *Kruse* also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, *Kruse* discloses "[t]he task of the procedure `Vivify` is to traverse the list `live`, determine whether each cell on it satisfies the conditions to become alive, and vivify it if so, else delete it from the list. The usual way to facilitate deletion from a linked list is to keep two pointers in lock step, one position apart, while traversing the list." … "Let us take advantage of the indirect linkage of our lists, and when we wish to delete an entry form the list, let us leave the node in place, but set its entry field to **nil**. In this way, the node will be flagged as empty when it is again encountered in the procedure `AddNeighbors`." *Kruse* at 219.

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		Kruse also discloses an exercise where the reader "rewrite(s) the procedure Vivify to use two pointers in traversing the list `live`, and dispose of redundant nodes when they are encountered. Also make the accompanying simplifications in the procedures `AddNeighbors` and `SubtractNeighbors`." *Kruse* at 222. By way of further example, one of ordinary skill in the art would have combined linked lists as taught by this reference and one of ordinary skill in the art to the system disclosed in the admitted prior and would have seen the benefits of doing so. One such benefit, for example, is hash table collision resolution.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	*Kruse* discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. *Kruse* also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, *Kruse* discloses "[t]he task of the procedure `Vivify` is to traverse the list `live`, determine whether each cell on it satisfies the conditions to become alive, and vivify it if so, else delete it from the list. The usual way to facilitate deletion from a linked list is to keep two pointers in lock step, one position apart, while traversing the list." … "Let us take advantage of the indirect linkage of our lists, and when we wish to delete an entry form the list, let us leave the node in place, but set its entry field to **nil**. In this way, the node will be flagged as empty when it is again encountered in the procedure

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		`AddNeighbors`.” *Kruse* at 219. *Kruse* also discloses an exercise where the reader “rewrite(s) the procedure Vivify to use two pointers in traversing the list `live`, and dispose of redundant nodes when they are encountered. Also make the accompanying simplifications in the procedures `AddNeighbors` and `SubtractNeighbors`.” *Kruse* at 222. Finally, *Kruse* discloses inserting and retrieving records from the system:

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination

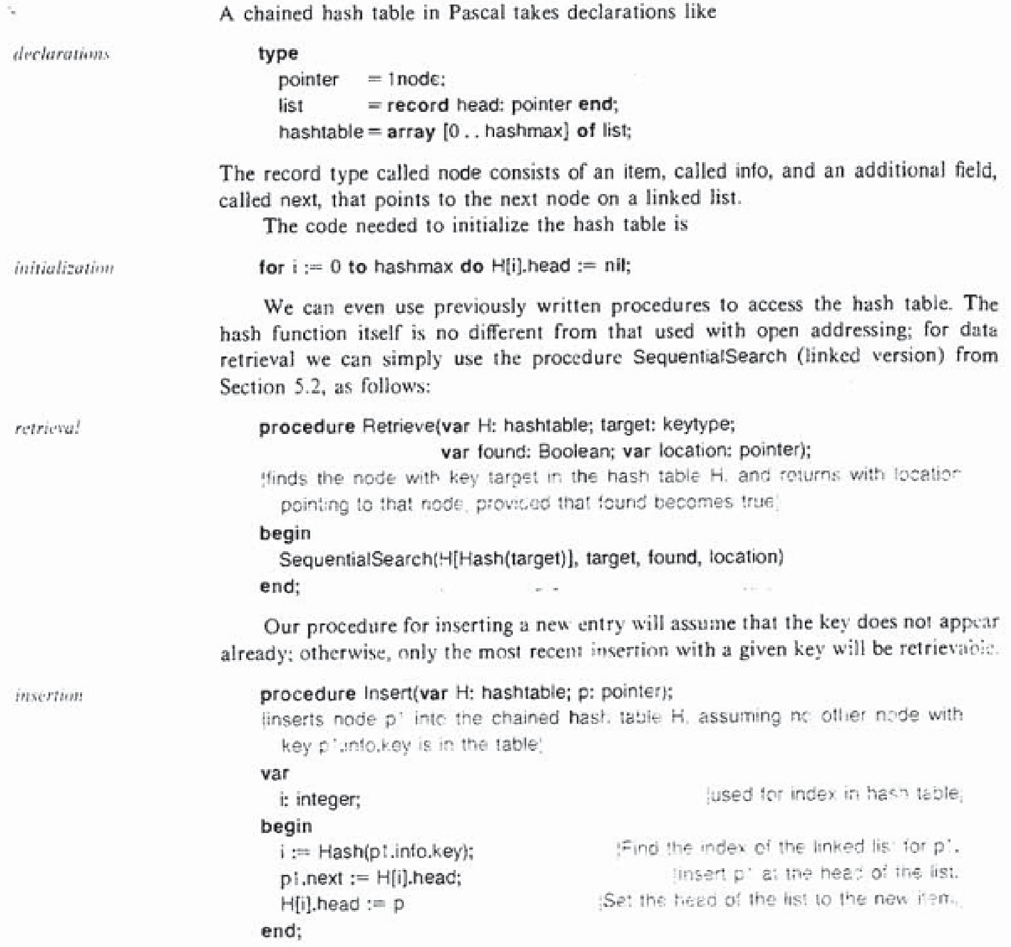
		
		A chained hash table in Pascal takes declarations like
		declarations
		`type`
		`  pointer   = ↑node;`
		`  list      = record head: pointer end;`
		`  hashtable = array [0 . . hashmax] of list;`
		The record type called node consists of an item, called info, and an additional field, called next, that points to the next node on a linked list.
		The code needed to initialize the hash table is
		initialization
		`for i := 0 to hashmax do H[i].head := nil;`
		We can even use previously written procedures to access the hash table. The hash function itself is no different from that used with open addressing; for data retrieval we can simply use the procedure SequentialSearch (linked version) from Section 5.2, as follows:
		retrieval
		`procedure Retrieve(var H: hashtable; target: keytype;`
		`                   var found: Boolean; var location: pointer);`
		`{finds the node with key target in the hash table H, and returns with location`
		`  pointing to that node, provided that found becomes true}`
		`begin`
		`  SequentialSearch(H[Hash(target)], target, found, location)`
		`end;`
		Our procedure for inserting a new entry will assume that the key does not appear already; otherwise, only the most recent insertion with a given key will be retrievable.
		insertion
		`procedure Insert(var H: hashtable; p: pointer);`
		`{inserts node p↑ into the chained hash table H, assuming no other node with`
		`  key p↑.info.key is in the table}`
		`var`
		`  i: integer;                          {used for index in hash table}`
		`begin`
		`  i := Hash(p↑.info.key);              {Find the index of the linked list for p↑.}`
		`  p↑.next := H[i].head;                {Insert p↑ at the head of the list.}`
		`  H[i].head := p                       {Set the head of the list to the new item.}`
		`end;`
		Kruse at 208.

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		By way of further example, one of ordinary skill in the art would have combined linked lists as taught by this reference and one of ordinary skill in the art to the system disclosed in the admitted prior and would have seen the benefits of doing so. One such benefit, for example, is hash table collision resolution.
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	*Kruse* discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in *Kruse* to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in *Kruse* with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in *Kruse* can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in *Kruse* is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Kruse combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination
		will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT$_i$ on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks:

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Kruse and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Kruse. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Kruse nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
		combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Kruse and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Kruse with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Kruse and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Kruse with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Kruse with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Kruse can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		ordinary skill in the art would recognize that combining Kruse with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Kruse with Thatte. Alternatively, it would also be obvious to combine Kruse with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The ’663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations

<u>EXHIBIT C-18</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-18

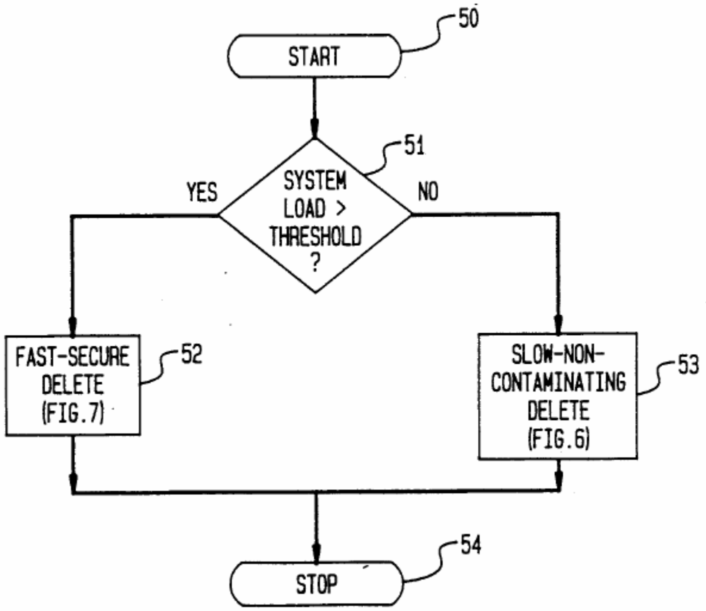
Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT C-18

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Kruse and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Kruse. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Kruse would be nothing more than the predictable use of prior art elements according to their established functions.

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Kruse and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Kruse with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT C-18

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Kruse and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Kruse. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Kruse would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Kruse and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Kruse to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Kruse with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Kruse can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Thus, the ’120 patent provides motivations to combine *Kruse* with Thatte, Dirks, the '663 patent, and/or the Opportunistic Garbage Collection Articles in addition to motivations within the text of *Kruse*, such as “[s]imilarly, when an item is no longer needed, its space can be returned to the system, which can then assign it to another user. In this way a program can start small and grow only as necessary, so that when it is small, it can run more efficiently, and when necessary, it can grow to the limits of the computer system.” *Kruse* at 107. *Kruse* further provides motivations within the text by posing the question “[d]o we need to make deletions, and, if so, when? We could keep track of all cells until the memory is full, and then delete those that are not needed.” *Kruse* at 216. To the extent that dynamically determining a maximum number of expired records is not disclosed by Kruse in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		would have been obvious to combine Linux 2.0.1 with Kruse. For example, both Linux 2.0.1 and Kruse describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function rt_cache_add automatically increments an integer variable rt_cache_size. *See* Linux 2.0.1, route.c at line 1359. When the function rt_cache_add removes an expired record, the function rt_cache_add decrements the variable rt_cache_size. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable rt_cache_size indicates the number of records in the hash table (i.e., ip_rt_hash_table). Because the function rt_cache_add automatically increments and decrements the variable rt_cache_size, the variable rt_cache_size is determined dynamically. Furthermore, Linux 2.0.1 includes the function rt_garbage_collect_1. The function rt_garbage_collect_1 loops through each of the linked lists in the ip_rt_hash_table global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function rt_garbage_collect_1 accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, *Kruse* discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. *Kruse* also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, *Kruse* discloses "[w]hen writing a program, we have had to decide on the maximum amount of memory that would be needed for our arrays and set this aside in the declarations." *Kruse* at 105. *Kruse* also discloses "First, and array must be declared that will hold the hash table. … To insert a record into the hash table, the hash function for the key is first calculated. … To retrieve the record with a given key is entirely similar." *Kruse* at 200. Furthermore, *Kruse* discloses that "[t]he idea we use is that of a pointer. A

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		pointer, also called a *link* or a *reference*, is defined to be a variable that gives the location of some other variable, typically of a record containing data that we wish to use. If we use pointers to locate all the records in which we are interested, then we need not be concerned about where the records themselves are actually stored, since by using a pointer, we can let the computer system itself locate the record when required." *Kruse* at 105. *Kruse* also discloses that the "idea of a linked list is, for every record in the list, to put a pointer into the record giving the location of the next record in the list." *Kruse* at 106. Additionally, *Kruse* states that "when an item is no longer needed, its space can be returned to the system, which can then assign it to another user." *Kruse* at 107. "If our program is one that continually sets up new nodes and disposes of others, then we shall often find it necessary to set up our own procedures to keep track of nodes that are no longer needed, and to reuse the space when new nodes are later required." *Kruse* at 117. Moreover, *Kruse* discloses "[w]e have already decided to represent our sparse array of cells as a hash table, but we have not yet decided between open addressing and chaining. … Do we need to make deletions, and, if so, when? We could keep track of all cells until the memory is full, and then delete those that are not needed. But this would require rehashing the full array, which would be slow and painful. With chaining we can easily dispose of cells as soon as they are not needed, and thereby reduce the number of cells in the hash table as much as possible." *Kruse* at 216. *Kruse* also discloses, "[i]f we use chaining, then we can add a cell to a list either by inserting the cell itself or a pointer to it, rather than by inserting its coordinates as before. In this way we can locate the cell directly with no need for any search." … "For reasons both of flexibility and time saving, therefore, let us decide to use dynamic memory

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
		allocation, a chained hash table, and linked lists." *Kruse* at 217. Finally, *Kruse* discloses that "[i]n using a hash table, let the nature of the data and the required operations help you decide between chaining and open addressing. Chaining is generally preferable if deletions are required, if the records are relatively large, or if overflow might be a problem." *Kruse* at 223.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	*Kruse* discloses accessing a linked list of records. *Kruse* also discloses accessing a linked list of records having same hash address. For example, *Kruse* discloses "[t]he idea of a hash table (such as the one shown in Figure 6.10) is to allow many of the different possible keys that might occur to be mapped to the same location in an array under the action of the index function." *Kruse* at 199. *Kruse* also discloses "[t]he task of the procedure is first to look in the hash table for the cell with the given coordinates. If the search is successful, then the procedure returns a pointer to the cell; otherwise, it must create a new cell, assign it the given coordinates, initialize its other fields to the default values, and put it in the hash table as well as return a pointer to it." *Kruse* at 220-21.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	*Kruse* discloses identifying at least some of the automatically expired ones of the records. *Kruse* also discloses identifying at least some of the automatically expired ones of the records. For example, *Kruse* discloses "[t]he task of the procedure `Vivify` is to traverse the list `live`, determine whether each cell on it satisfies the conditions to become alive, and vivify it if so, else delete it from the list. The usual way to facilitate deletion from a linked list is to keep two pointers in lock step, one position apart, while traversing the list." … "Let us take advantage of the

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		indirect linkage of our lists, and when we wish to delete an entry form the list, let us leave the node in place, but set its entry field to **nil**. In this way, the node will be flagged as empty when it is again encountered in the procedure `AddNeighbors`.” *Kruse* at 219. *Kruse* also discloses an exercise where the reader “rewrite(s) the procedure Vivify to use two pointers in traversing the list `live`, and dispose of redundant nodes when they are encountered. Also make the accompanying simplifications in the procedures `AddNeighbors` and `SubtractNeighbors`.” *Kruse* at 222.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	*Kruse* discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. *Kruse* also discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, *Kruse* discloses “[t]he task of the procedure `Vivify` is to traverse the list `live`, determine whether each cell on it satisfies the conditions to become alive, and vivify it if so, else delete it from the list. The usual way to facilitate deletion from a linked list is to keep two pointers in lock step, one position apart, while traversing the list.” … “Let us take advantage of the indirect linkage of our lists, and when we wish to delete an entry form the list, let us leave the node in place, but set its entry field to **nil**. In this way, the node will be flagged as empty when it is again encountered in the procedure `AddNeighbors`.” *Kruse* at 219. *Kruse* also discloses an exercise where the reader “rewrite(s) the procedure Vivify to use two pointers in traversing the list `live`, and dispose of redundant

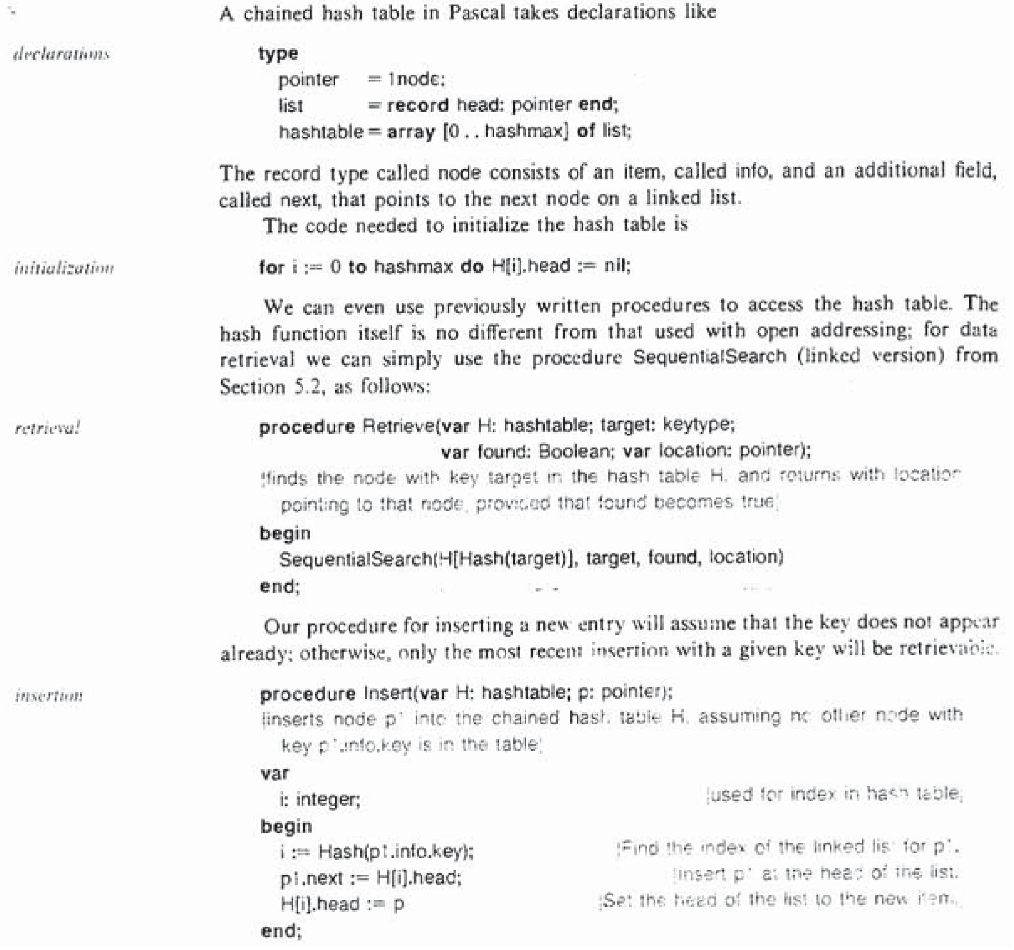
Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		nodes when they are encountered. Also make the accompanying simplifications in the procedures `AddNeighbors` and `SubtractNeighbors`." *Kruse* at 222. By way of further example, one of ordinary skill in the art would have combined linked lists as taught by this reference and one of ordinary skill in the art to the system disclosed in the admitted prior and would have seen the benefits of doing so. One such benefit, for example, is hash table collision resolution.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	*Kruse* discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, *Kruse* discloses "[t]he task of the procedure `Vivify` is to traverse the list `live`, determine whether each cell on it satisfies the conditions to become alive, and vivify it if so, else delete it from the list. The usual way to facilitate deletion from a linked list is to keep two pointers in lock step, one position apart, while traversing the list." … "Let us take advantage of the indirect linkage of our lists, and when we wish to delete an entry form the list, let us leave the node in place, but set its entry field to **nil**. In this way, the node will be flagged as empty when it is again encountered in the procedure `AddNeighbors`." *Kruse* at 219. *Kruse* also discloses an exercise where the reader "rewrite(s) the procedure Vivify to use two pointers in traversing the list `live`, and dispose of redundant nodes when they are encountered. Also make the accompanying

EXHIBIT C-18

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		simplifications in the procedures `AddNeighbors` and `SubtractNeighbors`.” *Kruse* at 222. Finally, *Kruse* discloses inserting and retrieving records from the system:

EXHIBIT C-18

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>Robert L. <em>Kruse</em>, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination</th></tr>
<tr><td></td><td></td><td>



A chained hash table in Pascal takes declarations like

declarations

```
type
  pointer   = ↑node;
  list      = record head: pointer end;
  hashtable = array [0 . . hashmax] of list;
```

The record type called node consists of an item, called info, and an additional field, called next, that points to the next node on a linked list.

The code needed to initialize the hash table is

initialization

```
for i := 0 to hashmax do H[i].head := nil;
```

We can even use previously written procedures to access the hash table. The hash function itself is no different from that used with open addressing; for data retrieval we can simply use the procedure SequentialSearch (linked version) from Section 5.2, as follows:

retrieval

```
procedure Retrieve(var H: hashtable; target: keytype;
                          var found: Boolean; var location: pointer);
{finds the node with key target in the hash table H, and returns with location
  pointing to that node, provided that found becomes true}
begin
  SequentialSearch(H[Hash(target)], target, found, location)
end;
```

Our procedure for inserting a new entry will assume that the key does not appear already; otherwise, only the most recent insertion with a given key will be retrievable.

insertion

```
procedure Insert(var H: hashtable; p: pointer);
{inserts node p↑ into the chained hash table H, assuming no other node with
  key p↑.info.key is in the table}
var
  i: integer;                                {used for index in hash table}
begin
  i := Hash(p↑.info.key);                {Find the index of the linked list for p↑.}
  p↑.next := H[i].head;                        {Insert p↑ at the head of the list.}
  H[i].head := p                          {Set the head of the list to the new item.}
end;
```

Kruse at 208.

</td></tr>
</table>

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		By way of further example, one of ordinary skill in the art would have combined linked lists as taught by this reference and one of ordinary skill in the art to the system disclosed in the admitted prior and would have seen the benefits of doing so. One such benefit, for example, is hash table collision resolution.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	*Kruse* discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in *Kruse* to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in *Kruse* with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in *Kruse* can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. Indeed, part of the motivation for the system disclosed in *Kruse* is avoiding these problems. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination
		‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Kruse combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Kruse and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Kruse. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Kruse nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Kruse and would have

<u>EXHIBIT C-18</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Kruse with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Kruse and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Kruse with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Kruse with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Kruse can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Kruse with the teachings of Thatte would solve this problem by dynamically determining how

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Kruse with Thatte. Alternatively, it would also be obvious to combine Kruse with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

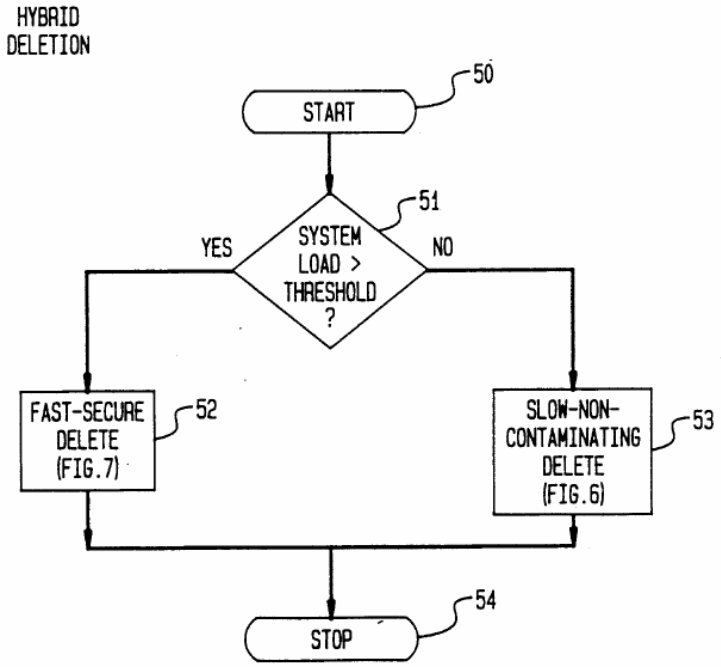
Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Kruse and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Kruse. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Kruse would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Kruse and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Kruse with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Kruse and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Kruse. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with Kruse would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Kruse and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Kruse to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Kruse with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Kruse can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine *Kruse* with Thatte, Dirks, the '663 patent, and/or the Opportunistic Garbage Collection Articles in addition to motivations within the text of *Kruse*, such as "[s]imilarly, when an item is no longer needed, its space can be returned to the system, which can then assign it to another user. In this way a program can start small and grow only as necessary, so that when it is small, it can run more efficiently, and when necessary, it can grow to the limits of the computer system." *Kruse* at 107. *Kruse* further provides motivations within the text by posing the question "[d]o we need to make deletions, and, if so, when? We could keep track of all cells until the memory is full, and then delete those that are not needed." *Kruse* at 216. To the extent that dynamically determining a maximum number of expired records is not disclosed by Kruse in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		would have been obvious to combine Linux 2.0.1 with Kruse. For example, both Linux 2.0.1 and Kruse describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination
		remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function

EXHIBIT C-18

Asserted Claims From U.S. Pat. No. 5,893,120		Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 ("Kruse") alone and in combination
		`rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not

EXHIBIT C-18

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Robert L. *Kruse*, Data Structures and Program Design, Prentice-Hall, Inc. 1984 and 1987 (“Kruse”) alone and in combination**
		limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Dixon and Calvert disclose an information storage and retrieval system. For example, Dixon and Calvert disclose a system for demultiplexing using a hash table of linked lists: "McKenney and Dove first introduced a demultiplexing algorithm that combines software caching and multiple hash chains. The algorithm maintains a linear list of PCBs for each of several hash chains. Each hash chain has an associated cache that points to the last PCB found on that chain." *See* Calvert and Dixon at 6.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Dixon and Calvert disclose a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. Dixon and Calvert also disclose a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Dixon and Calvert disclose a system for demultiplexing using a hash table of linked lists: "McKenney and Dove first introduced a demultiplexing algorithm that combines software caching and multiple hash chains. The algorithm maintains a linear list of PCBs for each of several hash chains. Each hash chain has an

EXHIBIT C-19

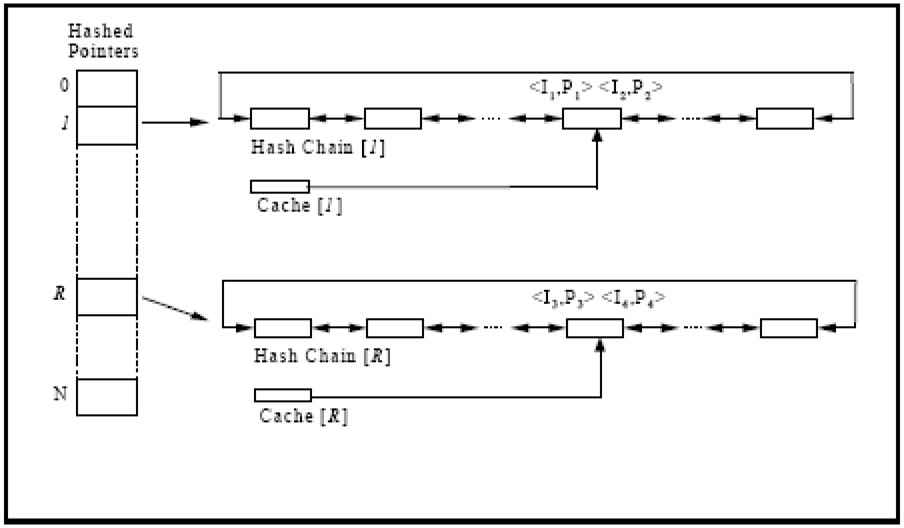
Asserted Claims From **U.S. Pat. No. 5,893,120**		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		associated cache that points to the last PCB found on that chain." *See* Calvert and Dixon at 6. "The next logical step in our investigation was to characterize the performance gains obtained by dividing the conventional single TCP PCB list into multiple shorter lists (*hash chains*) and use a single cache per hash chain to avoid lookups." *See* Calvert and Dixon at 13.  Hashed Pointers 0 *I* *R* N $<I_1,P_1> <I_2,P_2>$ Hash Chain [*I*] Cache [*I*] $<I_3,P_3> <I_4,P_4>$ Hash Chain [*R*] Cache [*R*] **Figure 4.3** Diagram of an *N* hash chain algorithm with one cache per chain. In addition, Dixon and Calvert disclose some of the records automatically

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter “Dixon and Calvert”)**
		expiring: *See*, *e.g.*, Dixon and Calvert at 8: “In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is closed.”
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Dixon and Calvert disclose a record search means utilizing a search key to access the linked list. Dixon and Calvert also disclose a record search means utilizing a search key to access a linked list of records having the same hash address. For example, Dixon and Calvert disclose using a hash key comprising connection information is used in conjunction with a hash function to access the appropriate hash chain: “When the connection is first created, a hash function uses some part of the connection’s information (e. g., IP address) to generate a hash value. The PCB is then added to the hash chain that corresponds to the generated hash value. Subsequently, the hash function will route any incoming packets destined for that PCB to the appropriate hash chain. Note that the same hash key (i. e., same connection information) must be present in the arriving packet in order to assure proper routing.” *See* Dixon and Calvert at 6.
[1c] the record search means including a means for identifying and	[5c] the record search means including means for identifying and removing	Dixon and Calvert directly or inherently disclose the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. Dixon

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	at least some expired ones of the records from the linked list of records when the linked list is accessed, and	and Calvert also directly or inherently disclose the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, Dixon and Calvert disclose a time a limit for a record remaining in the list of PCBs: "In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is closed." *See* Dixon and Calvert at 7. The existence of a limit on the time a record remains in the list requires removal at some point. Removal inherently requires the step of identification. Furthermore, because removal of a record from a linked list requires updating the links of other entries in the list, it inherently includes accessing the linked list of records.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed	Dixon and Calvert directly or inherently disclose means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Dixon and Calvert also directly or inherently disclose utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert,** ***Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)*** **(hereinafter "Dixon and Calvert")**
	linked list of records.	For example, Dixon and Calvert disclose a system for demultiplexing using a hash table of linked lists: "McKenney and Dove first introduced a demultiplexing algorithm that combines software caching and multiple hash chains. The algorithm maintains a linear list of PCBs for each of several hash chains. Each hash chain has an associated cache that points to the last PCB found on that chain. When the connection is first created, a hash function uses some part of the connection's information (e. g., IP address) to generate a hash value. The PCB is then added to the hash chain that corresponds to the generated hash value. Subsequently, the hash function will route any incoming packets destined for that PCB to the appropriate hash chain. Note that the same hash key (i. e., same connection information) must be present in the arriving packet in order to assure proper routing. The packet is assigned to its PCB via a BSD 4.3-Reno type search of the list." *See* Calvert and Dixon at 6. As described in the citation above, Calvert and Dixon disclose a search means—the combination of using a hash key and hash function to select a hash bucket and a "BSD 4.3-Reno type search" of the linked list chained to the hash bucket. As further disclosed in the citation above, insertion and retrieval when a new packet incoming packets arrive utilize the search means. In addition, Dixon and Calvert disclose some of the records automatically expiring: *See, e.g.*, Dixon and Calvert at 8: "In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is closed." The existence of a limit on the time a record remains in the list requires removal at some point. Furthermore, because removal of a record from a linked list requires updating the links of other entries in the list, it inherently includes accessing the linked list of records. Where the linked list is chained to a hash table, accessing the item to remove inherently requires use of the search means discussed above. Consequently, where a system maintains a list of PCBs using a hash table with external chaining and where said system inserts, retrieves and deletes using a record search means, then such a system is a means for inserting, retrieving, and deleting records, that utilizes a record search means and at the same time removes an expired entry from the system.
2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	Dixon and Calvert combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter “Dixon and Calvert”)**
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Dixon and Calvert and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Dixon and Calvert. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Dixon and Calvert nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Dixon and Calvert and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Dixon and Calvert with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Dixon and Calvert and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Dixon and Calvert with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Dixon and Calvert with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Dixon and Calvert can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Dixon and Calvert with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Dixon and Calvert with Thatte. Alternatively, it would also be obvious to combine Dixon and Calvert with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter “Dixon and Calvert”)**
		shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT C-19

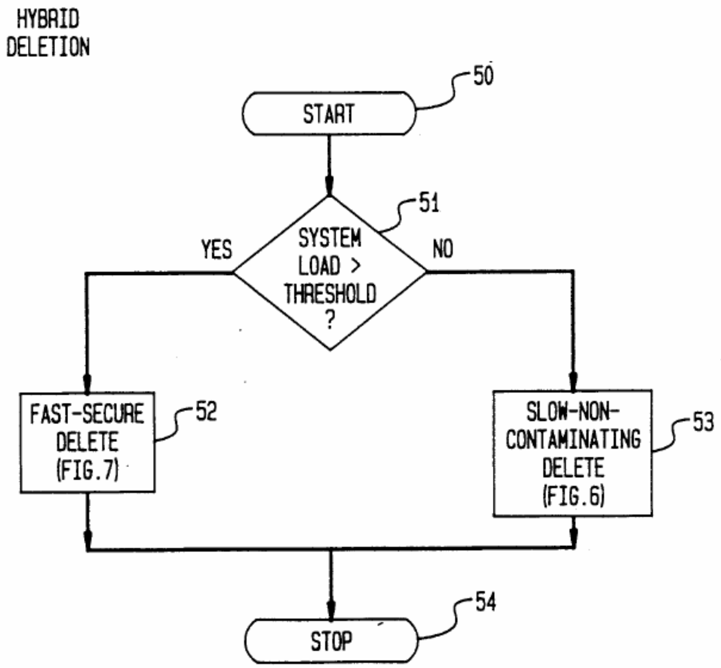
Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON- CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Dixon and Calvert and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Dixon and Calvert. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Dixon and Calvert would be nothing more than the predictable use of prior art elements

EXHIBIT C-19

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Dixon and Calvert and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Dixon and Calvert with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Dixon and Calvert and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Dixon and Calvert. Moreover, one of ordinary

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Dixon and Calvert would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Dixon and Calvert and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Dixon and Calvert to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Dixon and Calvert with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		linked list of records to solve a number of potential problems. For example, the removal of expired records described in Dixon and Calvert can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Dixon and Calvert in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Dixon and Calvert. For example, both Linux 2.0.1 and Dixon and Calvert describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold

EXHIBIT C-19

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter “Dixon and Calvert”)**
		`RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record’s last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record’s reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, Dixon and Calvert disclose a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. Dixon and Calvert also disclose a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Dixon and Calvert disclose a system for demultiplexing using a hash table of linked lists: "McKenney and Dove first introduced a demultiplexing algorithm that combines software caching and multiple hash chains. The algorithm maintains a linear list of PCBs for each of several hash chains. Each hash chain has an associated cache that points to the last PCB found on that chain." *See* Calvert and Dixon at 6. "The next logical step in our investigation was to characterize the performance gains obtained by dividing the conventional single TCP PCB list into multiple shorter lists (*hash chains*) and use a single cache per hash chain to avoid

<u>EXHIBIT C-19</u>

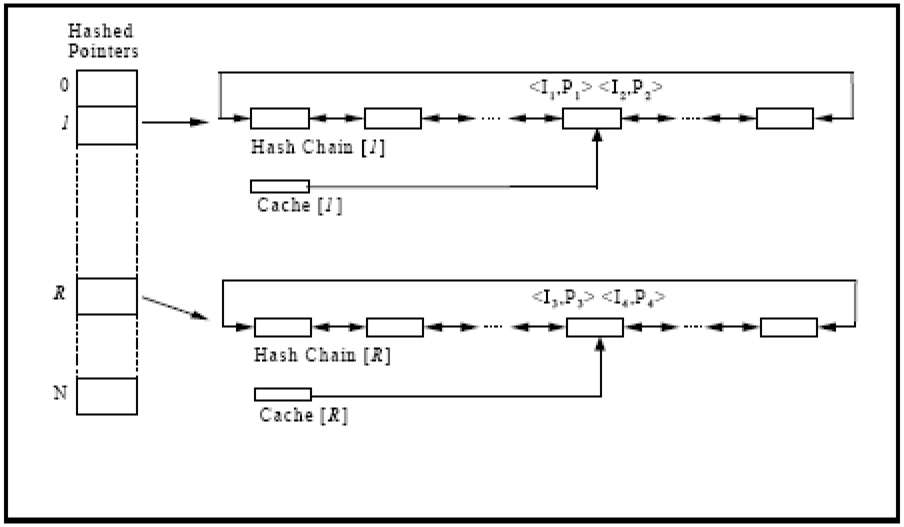
Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		lookups." *See* Calvert and Dixon at 13.  Hashed Pointers 0 *I* *R* N $<I_1,P_1>$ $<I_2,P_2>$ Hash Chain [*I*] Cache [*I*] $<I_3,P_3>$ $<I_4,P_4>$ Hash Chain [*R*] Cache [*R*] **Figure 4.3** Diagram of an *N* hash chain algorithm with one cache per chain. In addition, Dixon and Calvert disclose some of the records automatically expiring: *See, e.g.*, Dixon and Calvert at 8: "In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert,** ***Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)*** **(hereinafter "Dixon and Calvert")**
		closed."
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Dixon and Calvert disclose accessing a linked list of records. Dixon and Calvert also disclose accessing a linked list of records having same hash address. For example, Dixon and Calvert disclose using a hash key comprising connection information in conjunction with a hash function to access the appropriate hash chain: "When the connection is first created, a hash function uses some part of the connection's information (e. g., IP address) to generate a hash value. The PCB is then added to the hash chain that corresponds to the generated hash value. Subsequently, the hash function will route any incoming packets destined for that PCB to the appropriate hash chain. Note that the same hash key (i. e., same connection information) must be present in the arriving packet in order to assure proper routing." *See* Dixon and Calvert at 6.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Dixon and Calvert directly or inherently disclose identifying at least some of the automatically expired ones of the records. For example, Dixon and Calvert disclose a time a limit for a record remaining in the list of PCBs: "In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is closed." *See* Dixon and Calvert

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		at 7. The existence of a limit on the time a record remains in the list requires removal at some point. Removal inherently requires the step of identifying records to be removed.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Dixon and Calvert directly or inherently disclose removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, Dixon and Calvert disclose a time a limit for a record remaining in the list of PCBs: "In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is closed." *See* Dixon and Calvert at 7. The existence of a limit on the time a record remains in the list requires removal at some point. Removal inherently requires the step of identification. Furthermore, because removal of a record from a linked list requires updating the links of other entries in the list, it inherently includes accessing the linked list of records.
	[7d] inserting, retrieving or deleting one of the	Dixon and Calvert directly or inherently disclose inserting, retrieving or deleting one of the records from the system following the step of removing.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
	records from the system following the step of removing.	For example, Dixon and Calvert disclose a system for demultiplexing using a hash table of linked lists: "McKenney and Dove first introduced a demultiplexing algorithm that combines software caching and multiple hash chains. The algorithm maintains a linear list of PCBs for each of several hash chains. Each hash chain has an associated cache that points to the last PCB found on that chain. When the connection is first created, a hash function uses some part of the connection's information (e. g., IP address) to generate a hash value. The PCB is then added to the hash chain that corresponds to the generated hash value. Subsequently, the hash function will route any incoming packets destined for that PCB to the appropriate hash chain. Note that the same hash key (i. e., same connection information) must be present in the arriving packet in order to assure proper routing. The packet is assigned to its PCB via a BSD 4.3-Reno type search of the list." *See* Calvert and Dixon at 6. As disclosed in the citation above, insertion and retrieval when a new packet incoming packets arrive utilize the search means. In addition, Dixon and Calvert disclose some of the records automatically expiring: *See, e.g.*, Dixon and Calvert at 8: "In addition, the TCP implementation of all four servers had a *maximum segment lifetime* value of 60 seconds. We used this same value in our simulations. This value is important because a TCP connection remains in the PCB list for twice this length of time after it is

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		closed." The existence of a limit on the time a record remains in the list requires removal. Furthermore, as mentioned above, Calvert and Dixon disclose using a chained hash table in the context of demultiplexing. Demultiplexing is "the process of decomposing" a packet stream, such as a TCP/IP packet stream, to provide delivery to destination processes. *See* Calvert and Dixon at 2. A server employing a system for demultiplexing, such as the system disclosed by Calvert and Dixon, typically would receive a significant number of packets. For example, in an experiment Calvert and Dixon observed millions of incoming packets on four servers in under two hours. *See* Calvert and Dixon at 3-4, Table 3.1. As such, even with the use of a caching mechanism to avoid having to perform a lookup into a PCB list for each incoming packet, it is inherent that the disclosed system, after removing an expired entry from the PCB list, will insert a new entry or retrieve an entry in response to subsequent incoming packets.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Dixon and Calvert combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Dixon and Calvert and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of

<u>**EXHIBIT C-19**</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		determining the maximum number of records to sweep/remove in other hash tables implementations such as Dixon and Calvert. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with Dixon and Calvert nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Dixon and Calvert and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Dixon and Calvert with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert,** ***Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)*** **(hereinafter "Dixon and Calvert")**
		clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Dixon and Calvert and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Dixon and Calvert with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Dixon and Calvert with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Dixon and Calvert can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Dixon and Calvert with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Dixon and Calvert with Thatte.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		Alternatively, it would also be obvious to combine Dixon and Calvert with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The ’663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT C-19

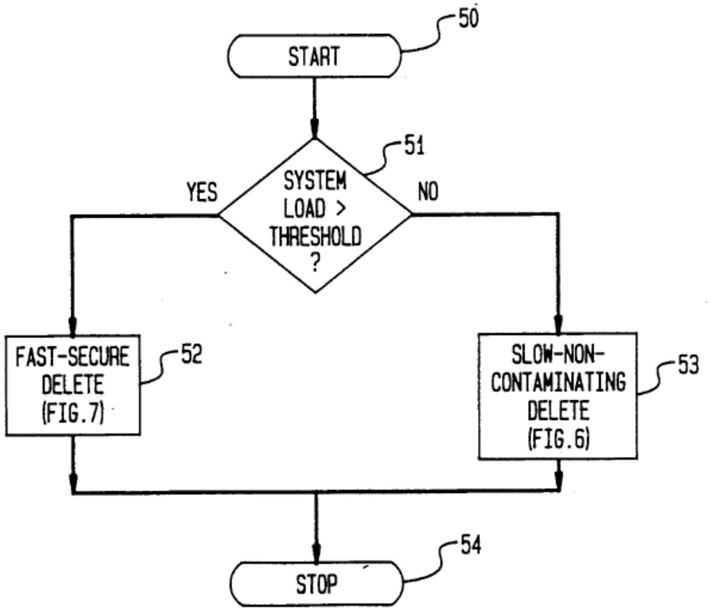
Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Dixon and Calvert and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Dixon and Calvert. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15.

EXHIBIT C-19

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Joseph T. Dixon and Kenneth Calvert,** ***Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)*** **(hereinafter "Dixon and Calvert")**
		Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Dixon and Calvert would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Dixon and Calvert and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Dixon and Calvert with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged;

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Dixon and Calvert and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Dixon and Calvert. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Dixon and Calvert would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Dixon and Calvert and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Dixon and Calvert to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		ordinary skill in the art would have been motivated to combine the system disclosed in Dixon and Calvert with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Dixon and Calvert can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Dixon and Calvert in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Dixon and Calvert. For example, both Linux 2.0.1 and Dixon and Calvert describe systems and methods for performing data storage and retrieval

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")
		using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list.

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not

EXHIBIT C-19

Asserted Claims From U.S. Pat. No. 5,893,120		**Joseph T. Dixon and Kenneth Calvert, *Increasing Demultiplexing Efficiency in TCP/IP Network Servers (1996)* (hereinafter "Dixon and Calvert")**
		limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, LINUX 1.3.52 discloses an information storage and retrieval system. For example, LINUX 1.3.52 includes the `ip_rt_hash_table` global variable, which is an information storage and retrieval system.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	LINUX 1.3.52 discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. LINUX 1.3.52 also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, LINUX 1.3.52 includes the `ip_rt_hash_table` global variable, which is composed of an array of pointers to `struct rtable`s. See line 144. Each `struct rtable` contains an `rt_next` field, which is a pointer to another `struct rtable`. See /include/net/route.h, line 124. Accordingly, `struct rtable` defines (among other things) a linked list. As suggested by its name, the `ip_rt_hash_table` global variable uses a hashing means to provide access to its stored linked lists. The access is described below; the hash address itself is computed at lines 1109 and 1467, which call the function `ip_rt_hash_code`. The records in the system LINUX 1.3.52 discloses includes records, at least some of which automatically expire. `struct rtable` also includes the `rt_lastuse` field, which is used to determine whether the record has automatically expired. See

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		/include/net/route.h, line 131 and analysis below.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	LINUX 1.3.52 discloses a record search means utilizing a search key to access the linked list. LINUX 1.3.52 also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, as detailed in part [1a/5a], the `ip_rt_hash_table` global variable contains an array of linked lists of type `struct rtable`. As suggested by its name, the `ip_rt_hash_table` global variable is accessed using a search key. Specifically, the function `rt_cache_add` uses the search key `hash` to access the linked list at the "hash" index of the `ip_rt_hash_table` array. See lines 1415 and 1426.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	LINUX 1.3.52 discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. For example, as detailed in step [1b/5b], the function `rt_cache_add` accesses a linked list within the `ip_rt_hash_table` global variable at line 1415, and appends a new record to the front of the linked list at line 1426. As detailed in the comment at line 1432, `rt_cache_add` then iterates through the same linked list to remove aged off or "automatically expired" entries. Specifically, line 1439 determines whether the record has expired, line 1442 removes the expired record from the linked list, and line 1448 deletes the expired record from memory.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("Linux 1.3.52") alone and in combination**
		Thus, the linked list at `ip_rt_hash_table[hash]` is accessed at lines 1415 through 1427, when the `rt_cache_add` method adds the new record to the front of the linked list, and from lines 1435 through 1453, when the `rt_cache_add` method iterates through the linked list looking for duplicate and expired entries.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Linux 1.3.52 discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Linux 1.3.52 also discloses means, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, `rt_cache_add` discloses a means for inserting a record into the linked list stored at `ip_rt_hash_table[hash]` and, at the same time, removing at least some of the records in that accessed linked list. `rt_cache_add` also discloses a means for retrieving records from the linked list. See, e.g., lines 1415, 1435. `rt_cache_add` also discloses a means for deleting records from the linked list. See, e.g., lines 1442, 1448. Further, the `while` loop of lines 1435 to 1453 is checking for duplicate entries (deleting entries) at the same time that it is checking for automatically expired entries. See lines 1439 and 1440. To the extent that Linux 1.3.52 does not disclose this limitation, <u>gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas</u>

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz (“Linux 1.3.52”) alone and in combination**
		Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.3.52 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since Linux 1.3.52 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of Linux 1.3.52 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Linux 1.3.52 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here: In cainsert(): `275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {` In calookup():

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**

		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining	LINUX 1.3.52 combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
maximum number for the record search means to remove in the accessed linked list of records.	maximum number for the record search means to remove in the accessed linked list of records.	Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 1.3.52 and Dirks relate to deletion of aged records upon the

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 1.3.52. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Linux 1.3.52 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 1.3.52 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.3.52 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 1.3.52 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 1.3.52 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 1.3.52 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 1.3.52 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 1.3.52 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 1.3.52 with Thatte. Alternatively, it would also be obvious to combine Linux 1.3.52 with the '663

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-1

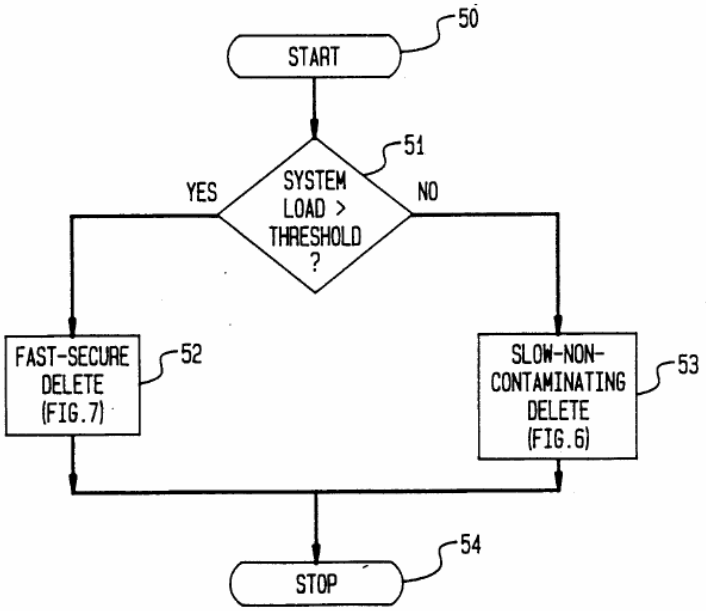
Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 1.3.52 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 1.3.52. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Linux 1.3.52 would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Linux 1.3.52 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 1.3.52 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 1.3.52 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 1.3.52. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 1.3.52 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 1.3.52 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 1.3.52 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Linux 1.3.52 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 1.3.52 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("Linux 1.3.52") alone and in combination**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Linux 1.3.52 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Linux 1.3.52 . For example, both Linux 2.0.1 and Linux 1.3.52 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		`rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the

<u>EXHIBIT D-1</u>

Asserted Claims From **U.S. Pat. No. 5,893,120**		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, LINUX 1.3.52 discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. LINUX 1.3.52 also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, LINUX 1.3.52 includes the `ip_rt_hash_table` global variable, which is composed of an array of pointers to `struct rtable`s. See line 144. Each `struct rtable` contains an `rt_next` field, which is a pointer to another `struct rtable`. See /include/net/route.h, line 124. Accordingly, `struct rtable` defines (among other things) a linked list. `struct rtable` also includes the `rt_lastuse` field, which is used to determine whether the record has automatically expired. See /include/net/route.h, line 131 and analysis below.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	LINUX 1.3.52 discloses accessing a linked list of records. Linux 1.3.52 also discloses accessing a linked list of records having same hash address. For example, the function `rt_cache_add` accesses the linked list at the "hash" index of the `ip_rt_hash_table` array. See lines 1415 and 1426. In addition, the linked list at `ip_rt_hash_table[hash]` is accessed from lines 1435 through 1453, when the `rt_cache_add` method iterates through the linked list.
[3b] identifying at least	[7b] identifying at least	LINUX 1.3.52 discloses identifying at least some of the automatically expired

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
some of the automatically expired ones of the records, and	some of the automatically expired ones of the records,	ones of the records. For example, the function `rt_cache_add` accesses a linked list within the `ip_rt_hash_table` global variable at line 1415, and appends a new record to the front of the linked list at line 1426. As detailed in the comment at line 1432, `rt_cache_add` then iterates through the same linked list to remove aged off or "automatically expired" entries. Specifically, the loop beginning at line 1435 iterates through the records in the previously-accessed linked list, and line 1439 identifies whether a particular record has expired.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	LINUX 1.3.52 discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, the loop beginning at line 1435 iterates through the records in the previously-accessed linked list, and line 1439 identifies whether a particular record has expired. Line 1442 removes the expired record from the linked list, and line 1448 deletes the expired record from memory.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	LINUX 1.3.52 discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, the loop beginning at line 1435 iterates through the records in the previously-accessed linked list, and line 1439 identifies whether a particular record has expired. Line 1442 removes the expired record from the linked list, and line 1448 deletes the expired record from memory. Further, the `while` loop of lines 1435 to 1453 is checking for duplicate entries (deleting entries) at the same time that it is checking for automatically expired entries. See lines

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		1439 and 1440. A duplicate entry may be deleted following the removal of at least some of the automatically expired records from the linked list.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	LINUX 1.3.52 combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz (“LINUX 1.3.52”) alone and in combination**
		After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 1.3.52 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 1.3.52. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Linux 1.3.52 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 1.3.52 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.`

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.3.52 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 1.3.52 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 1.3.52 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 1.3.52 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 1.3.52 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 1.3.52 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 1.3.52 with Thatte. Alternatively, it would also be obvious to combine Linux 1.3.52 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of

EXHIBIT D-1

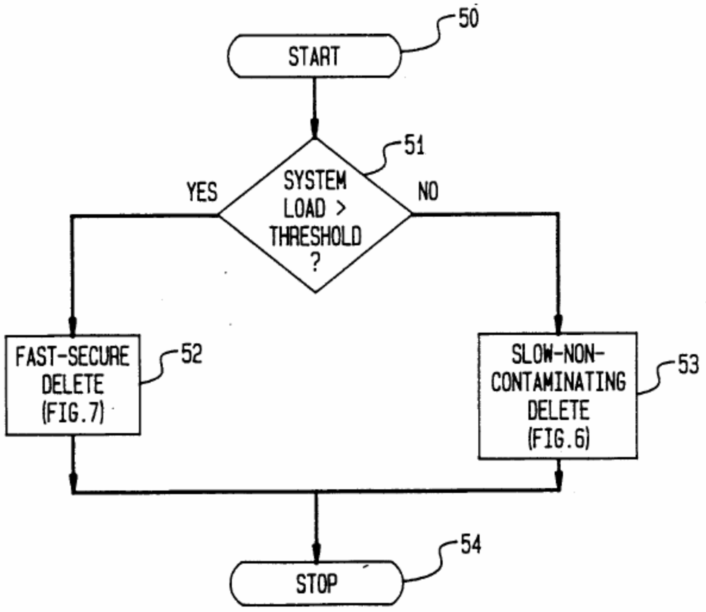
Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		Id. at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 1.3.52 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 1.3.52. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz (“LINUX 1.3.52”) alone and in combination**
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Linux 1.3.52 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Linux 1.3.52 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 1.3.52 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 1.3.52 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 1.3.52. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 1.3.52 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 1.3.52 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 1.3.52 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system

Asserted Claims From U.S. Pat. No. 5,893,120		**LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination**
		disclosed in Linux 1.3.52 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 1.3.52 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by LINUX 1.3.52 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with LINUX 1.3.52 . For example, both Linux 2.0.1 and LINUX 1.3.52 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 **`net\ipv4\route.c`**, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 **net\ipv4\route.c**, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 **net\ipv4\route.c**, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero.

EXHIBIT D-1

Asserted Claims From U.S. Pat. No. 5,893,120		LINUX 1.3.52 `net\ipv4\route.c`, available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.52.tar.gz ("LINUX 1.3.52") alone and in combination
		Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 netinet/if_ether.c, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("if_ether.c") alone and in combination
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, if_ether.c discloses an information storage and retrieval system. For example, the implementation of the Address Resolution Protocol in if_ether.c in BSD 4.2 includes an information storage and retrieval system that stores and retrieves records used by the protocol.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	if_ether.c discloses a hash table (arptab) which resolves collisions through arrays. See, e.g., lines 42-49. It would have been obvious to one of ordinary skill in the art that arptab could resolve collisions with linked lists rather than arrays, as both linked lists and arrays are fundamental data structures used to store multiple data items. See below. if_ether.c also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, the structure if_ether.c describes the use of a hash table, arptab, with external chaining to resolve collisions. See, e.g., lines 42-49. Though the external chaining involves the use of an array rather than a linked list, it would have been obvious to a person skilled in the art that a linked list could be used instead of an array. The use of linked lists for external chaining in hash tables was well known in the art. Indeed, according to Knuth, "the most obvious way to solve this problem [of collisions] is to maintain *M* linked lists, one for each possible hash code." See "The Art of Computer Programming", Sorting and Searching, D.E. Knuth, Addison-Wesley Series in

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 **`netinet/if_ether.c`**, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("**`if_ether.c`**") alone and in combination
		Computer Science and Information Processing, pp. 513, 1973. *See also* Mark A. Weiss, Data Structures and Algorithm Analysis, p. 152-157, 1993 (using linked lists to resolve collisions in external chaining but noting that any scheme besides linked lists could be used). One of ordinary skill in the art would have been motivated to try using "the most obvious" solution to external chaining, linked lists, instead of the array taught in `if_ether.c`. The records in the system `if_ether.c` discloses includes records, at least some of which automatically expire. For example, the `arptab` table in `if_ether.c` includes within each entry an `at_timer` variable which keeps track of the minutes since the last reference. That `at_time` variable is used to expire the entry when the time since the last reference exceeds a given amount. See function `arptimer()`, lines 126-130.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	`if_ether.c` discloses a record search means utilizing a search key to access an array. Similarly, `if_ether.c` discloses a record search means utilizing a search key to access an array of records having the same hash address. For example, the `arptab` structure in `if_ether.c` can be accessed with a search key. That search key provides access to a series of entries within the `arptab` structure that have the same hash address. See, e.g., function `arptnew()`, lines 376-384. As discussed in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could be used instead of an array to resolve the collisions in `arptab`, in which case the access in this element would occur on a linked list.

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	`if_ether`.c discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. For example, the function `arptimer()` accesses the `arptab` structure and removes all entries that have expired when that access occurs. The function `arptnew()` also accesses the `arptab` structure in order to add an entry, and if the structure is full, `arptnew()` removes the oldest entry in the table and inserts the new entry in its place. Though this removal of expired records occurs in an array, as discussed above in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could be used to resolve collisions in the hash table, in which case the removal taught in this element would occur in the linked list.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	`if_ether.c` discloses means, utilizing the record search means, for accessing an array and, at the same time, removing at least some of the expired ones of the records in the array. `if_ether.c` also discloses means, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed array of records. As discussed in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could be used instead of an array. For example, `arptfree()` utilizes the record search means to insert a new entry in the `arptab` structure and, at the same time, remove one of the expired records from the structure when the structure is full. It would have

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		been obvious to one skilled in the art that retrieval or deletion could have been done as well as the insert, since these are all basic functions that can be performed on a hash table. *See, e.g.*, “The Art of Computer Programming”, Sorting and Searching, D.E. Knuth, Addison-Wesley Series in Computer Science and Information Processing, pp. 506-549; “Data Structures and Program Design”, R.L. Kruse, Prentice-Hall, Inc. 1984, pp. 104-148. Though these actions occur in an array, as discussed above in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could be used to resolve collisions in the hash table, in which case the actions taught in this element would occur in the linked list. To the extent that if_ether.c does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in if_ether.c with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since if_ether.c utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of if_ether.c with the system including

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining if_ether.c with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: “*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). “*Calookup*() searches for a cached entry matching the key passed as an argument.” *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). “*Caremove*() removes the cached entry whose key is given, if one exists.” *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**

		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		“In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.” *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1	6. The information storage and retrieval system according to claim 5	`if_ether.c` discloses dynamically determining maximum number of expired ones of the records to remove when the array is accessed.

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	For example, the function `arptfree()` only removes an expired element when the `arptab` structure is full. In this way, the function dynamically determines the maximum number of elements to remove by computing whether to remove some or none of the expired elements. Though these removals of expired records occur in an array, as discussed above in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could be used to resolve collisions in the hash table, in which case the removals taught in this claim would occur in the linked list. To the extent `if_ether.c` does not disclose this element, it would have been obvious to one of ordinary skill in the art. `if_ether.c` combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		$$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both if_ether.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as if_ether.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with if_ether.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with if_ether.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in if_ether.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both if_ether.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		result of combining if_ether.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine if_ether.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in if_ether.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining if_ether.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine if_ether.c with Thatte. Alternatively, it would also be obvious to combine if_ether.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination
		the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-2

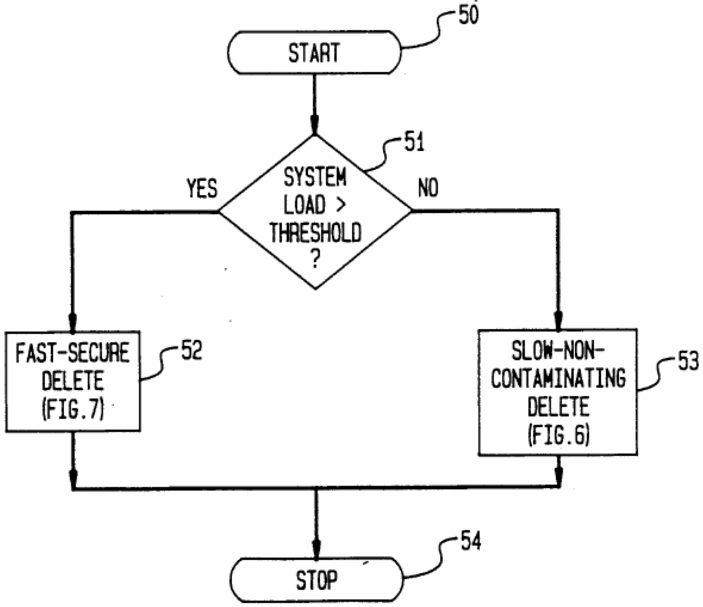
Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-2

Asserted Claims From **U.S. Pat. No. 5,893,120**		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both if_ether.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in if_ether.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with if_ether.c would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with if_ether.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine if_ether.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both if_ether.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as if_ether.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with if_ether.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with if_ether.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in if_ether.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in if_ether.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in if_ether.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by if_ether.c in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with if_ether.c. For example, both Linux 2.0.1 and if_ether.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 **netinet/if_ether.c**, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("**if_ether.c**") alone and in combination
		line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination**
		process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-2

Asserted Claims From **U.S. Pat. No. 5,893,120**		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination**
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, `if_ether.c` discloses a method for storing and retrieving information records using a chain of records to store and provide access to the records, at least some of the records automatically expiring. `if_ether.c` also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, the `arptab` structure defined in `if_ether.c` is a hash table that uses external chaining with arrays to resolve collisions between entries with the same hash address. As discussed in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could have been utilized instead of an array to resolve collisions. The records in the system `if_ether.c` discloses includes records, at least some of which automatically expire. For example, the `arptab` table in `if_ether.c` includes within each entry an `at_timer` variable which keeps track of the minutes since the last reference. That `at_time` variable is used to expire the entry when the time since the last reference exceeds a given amount. See function `arptimer()`, lines 126-130.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same	`if_ether.c` discloses accessing an array of records. Similarly, `if_ether.c` discloses accessing an array of records having the same hash

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 **netinet/if_ether.c**, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“**if_ether.c**”) alone and in combination
	hash address,	address. As discussed in [1b/5b], it would have been obvious to one or ordinary skill in the art that the access could occur in a linked list rather than an array. For example, both the `arptimer()` and `arptnew()` functions in `if_ether.c` access the array that stores records having the same hash address. See lines 123-32, 376-84.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	`if_ether.c` discloses identifying at least some of the automatically expired ones of the records. For example, `arptimer()` identifies whether any of the entries in `arptab` have expired on lines 126-29. The function `arptnew()` identifies automatically expired records on lines 380-83.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	`if_ether.c` discloses removing at least some of the automatically expired records from the array when the array is accessed. As discussed in [1c/5c], it would have been obvious to one of ordinary skill in the art that the same step could be performed on a linked list where the records were stored in a linked list. For example, the function `arptimer()` accesses the `arptab` structure and removes all entries that have expired when that access occurs. The function `arptnew()` also accesses the `arptab` structure in order to add an entry, and if the structure is full, `arptnew()` identifies an expired entry in the table and removes the entry by calling the function `arptfree()`. Lines 385-86.
	[7d] inserting, retrieving	`if_ether.c` discloses inserting, retrieving or deleting one of the records

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
	or deleting one of the records from the system following the step of removing.	from the system following the step of removing. For example, function `arptnew()` inserts a new entry into the `arptab` structure after the expired element is removed. Lines 388-89.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	`if_ether.c` discloses dynamically determining maximum number of expired ones of the records to remove when the array is accessed. For example, the function `arptfree()` only removes an expired element when the `arptab` structure is full. In this way, the function dynamically determines the maximum number of elements to remove by computing whether to remove some or none of the expired elements. Though these removals of expired records occur in an array, as discussed above in [1a/5a], it would have been obvious to one of ordinary skill in the art that a linked list could be used to resolve collisions in the hash table, in which case the removals taught in this claim would occur in the linked list. To the extent `if_ether.c` does not disclose this element, it would have been obvious to one of ordinary skill in the art. `if_ether.c` combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both if_ether.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as if_ether.c. Moreover, one of ordinary skill in the art

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with if_ether.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with if_ether.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in if_ether.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		combinations would improve the similar systems and methods in the same way. Additionally, Ass both if_ether.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining if_ether.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine if_ether.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in if_ether.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining if_ether.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine if_ether.c with Thatte. Alternatively, it would also be obvious to combine if_ether.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent:

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		> during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). > In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. > This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-2

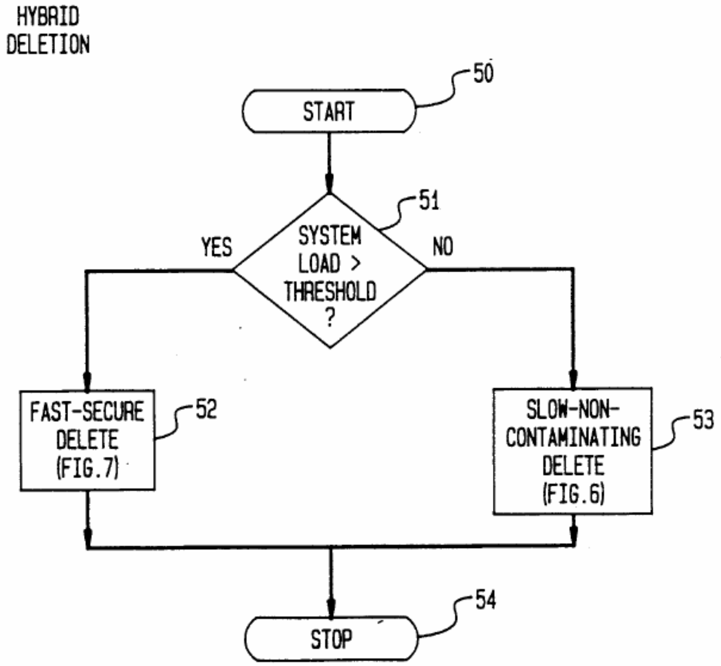
Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		FIG.5  HYBRID DELETION START 50 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both if_ether.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in if_ether.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with if_ether.c would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-2

Asserted Claims From **U.S. Pat. No. 5,893,120**		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions /4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with if_ether.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine if_ether.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both if_ether.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as if_ether.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with if_ether.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with if_ether.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in if_ether.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in if_ether.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in if_ether.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		**BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz (“`if_ether.c`”) alone and in combination**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by if_ether.c in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with if_ether.c. For example, both Linux 2.0.1 and if_ether.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 netinet/if_ether.c, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("if_ether.c") alone and in combination
		number of records in the hash table (i.e., ip_rt_hash_table). Because the function rt_cache_add automatically increments and decrements the variable rt_cache_size, the variable rt_cache_size is determined dynamically. Furthermore, LINUX 2.0.1 includes the function rt_garbage_collect_1. The function rt_garbage_collect_1 loops through each of the linked lists in the ip_rt_hash_table global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function rt_garbage_collect_1 accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination
		line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this

EXHIBIT D-2

Asserted Claims From U.S. Pat. No. 5,893,120		BSD 4.2 `netinet/if_ether.c`, available at http://ftp.math.utah.edu/pub/mirrors/minnie.tuhs.org/4BSD/Distributions/4.2BSD/srcsys.tar.gz ("`if_ether.c`") alone and in combination
		process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-2

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("vfs_cache.c") alone and in combination
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, vfs_cache.c discloses an information storage and retrieval system. For example, the implementation of the name cache in vfs_cache.c in FreeBSD includes an information storage and retrieval system that stores and retrieves names found by directory scans.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	vfs_cache.c discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. vfs_cache.c also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, vfs_cache.c describes the use of a hash table, nchashtbl, with external chaining using linked lists to resolve collisions. *See* lines 75-84. The records in the system vfs_cache.c discloses includes records, at least some of which automatically expire. *See, e.g.*, lines 51-69, 214-226. For example, vfs_cache.c maintains a list of least recently used entries in the hash table in the structure nclruhead. Line 76. An entry automatically expires when (1) it is the least recently used entry, (2) the function cache_enter() tries to insert another entry into nchashtbl and (3) nchashtbl is already full. *See* lines 51-69, 214-226.

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	`vfs_cache.c` discloses a record search means utilizing a search key to access the linked list. `vfs_cache.c` also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, the function `cache_enter()` utilizes a search key to access a linked list of records having the same hash address. *See* lines 193-246.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	`vfs_cache.c` discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. For example, the function `cache_enter()` accesses the `nchashtbl` structure and identifies an expired entry, which it removes from the linked list of records when it adds another entry to the hash table. *See* lines 214-245.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	`vfs_cache.c` discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. `vfs_cache.c` also discloses means, utilizing the record search means, for inserting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, the function `cache_enter()` accesses the `nchashtbl` structure and identifies an expired entry, which it removes from the linked list of records when it adds another entry to the hash table. *See* lines 193-245.

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		To the extent `cache_enter()` does not include means for retrieving and deleting records utilizing the record search means, it would have been obvious to one skilled in the art that retrieval or deletion could have been done as well as the insert, since these are all basic functions that can be performed on a hash table or a linked list in similar ways. *See*, *e.g.*, “The Art of Computer Programming”, Sorting and Searching, D.E. Knuth, Addison-Wesley Series in Computer Science and Information Processing, pp. 506-549; “Data Structures and Program Design”, R.L. Kruse, Prentice-Hall, Inc. 1984, pp. 104-148. To the extent that vfs_cache.c does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in vfs_cache.c with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See*,

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		e.g., Comer at 3-10. For example, since vfs_cache.c utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of vfs_cache.c with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining vfs_cache.c with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: “*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). “*Calookup*() searches for a cached entry matching the key passed as an argument.” *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex().

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**

		“*Caremove*() removes the cached entry whose key is given, if one exists.” *See* Comer at 4.
		See also, gcache.c at lines 355-376, defining caremove().
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		“In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.”

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("`vfs_cache.c`") alone and in combination**

		See Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	`vfs_cache.c` discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, the function `cache_enter()` only removes an expired element when the `arptab` structure is full. In this way, the function dynamically determines the maximum number of elements to remove by computing whether to remove some or none of the expired elements. *See* lines 193-245. To the extent `vfs_cache.c` does not disclose this element, it would have been obvious to one of ordinary skill in the art.

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("vfs_cache.c") alone and in combination
		vfs_cache.c combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both vfs_cache.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as vfs_cache.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with vfs_cache.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with vfs_cache.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.`

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("`vfs_cache.c`") alone and in combination
		Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in vfs_cache.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both vfs_cache.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining vfs_cache.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine vfs_cache.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in vfs_cache.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining vfs_cache.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("vfs_cache.c") alone and in combination**
		things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine vfs_cache.c with Thatte. Alternatively, it would also be obvious to combine vfs_cache.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-3

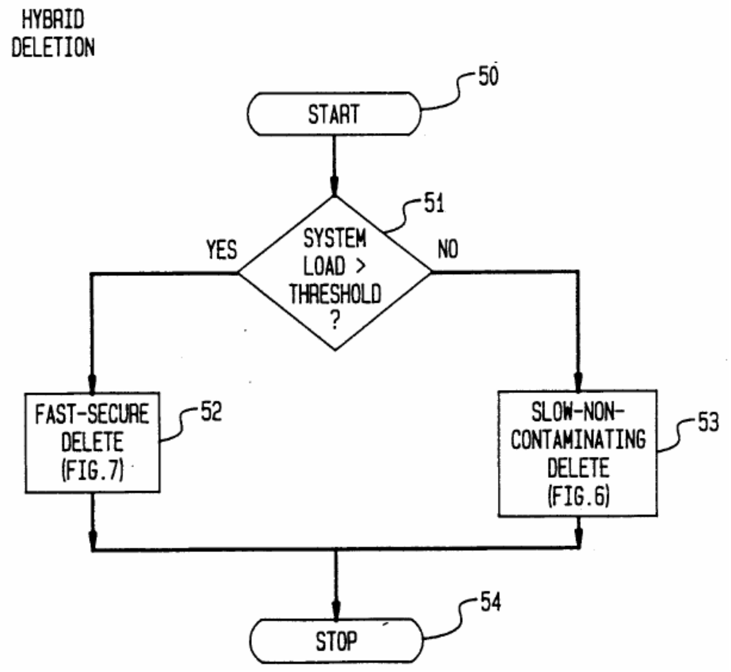
Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“vfs_cache.c”) alone and in combination
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both vfs_cache.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in vfs_cache.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with vfs_cache.c would be nothing more than the predictable use of prior art elements according

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("vfs_cache.c") alone and in combination**
		to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with vfs_cache.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine vfs_cache.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both vfs_cache.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as vfs_cache.c. Moreover, one of ordinary skill in

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with vfs_cache.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with vfs_cache.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in vfs_cache.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in vfs_cache.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		linked list of records to solve a number of potential problems. For example, the removal of expired records described in vfs_cache.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by vfs_cache.c in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with vfs_cache.c . For example, both Linux 2.0.1 and vfs_cache.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("`vfs_cache.c`") alone and in combination
		integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		`RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“vfs_cache.c”) alone and in combination
		RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock’s proposed claim constructions, the records removed by the function rt_garbage_collect_1 are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record’s last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record’s reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1,

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("`vfs_cache.c`") alone and in combination
		route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, `vfs_cache.c` discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. `vfs_cache.c` also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, `vfs_cache.c` describes the use of a hash table, `nchashtbl`, with external chaining using linked lists to resolve collisions. *See* lines 75-84. The records in the system `vfs_cache.c` discloses includes records, at least some of which automatically expire. *See*, *e.g.*, lines 51-69, 214-226. For example, `vfs_cache.c` maintains a list of least recently used entries in the hash table in the structure `nclruhead`. Line 76. An entry automatically expires when (1) it is the least recently used entry, (2) the function

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("`vfs_cache.c`") alone and in combination**
		`cache_enter()` tries to insert another entry into `nchashtbl` and (3) `nchashtbl` is already full. *See*, lines 51-69, 214-226.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	`vfs_cache.c` discloses accessing the linked list of records. `vfs_cache.c` also discloses accessing a linked list of records having same hash address For example, the `cache_enter()` function in `vfs_cache.c` accesses the linked list that stores records having the same hash address. *See* lines 193-245.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	`vfs_cache.c` discloses identifying at least some of the automatically expired ones of the records. For example, the function `cache_enter()` accesses the `nchashtbl` structure and identifies an expired entry. *See* lines 193-245.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	`vfs_cache.c` discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, the function `cache_enter()` removes the previously identified expired record from the linked list of records when it adds another entry to the hash table. *See* lines 193-245.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	`vfs_cache.c` discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, function `cache_enter()` inserts a new entry into the `nchashtbl` structure after the expired element is removed. *See* lines 236-45.

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	`vfs_cache.c` discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, the function `cache_enter()` only removes an expired element when the `arptab` structure is full. In this way, the function dynamically determines the maximum number of elements to remove. *See* lines 193-245. To the extent `vfs_cache.c` does not disclose this element, it would have been obvious to one of ordinary skill in the art. `vfs_cache.c` combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“vfs_cache.c”) alone and in combination**
		$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both vfs_cache.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as vfs_cache.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with vfs_cache.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with vfs_cache.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in vfs_cache.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same

EXHIBIT D-3

Asserted Claims From **U.S. Pat. No. 5,893,120**		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		way. Additionally, Ass both vfs_cache.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining vfs_cache.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine vfs_cache.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in vfs_cache.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining vfs_cache.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine vfs_cache.c with Thatte. Alternatively, it would also be obvious to combine vfs_cache.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent:

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-3

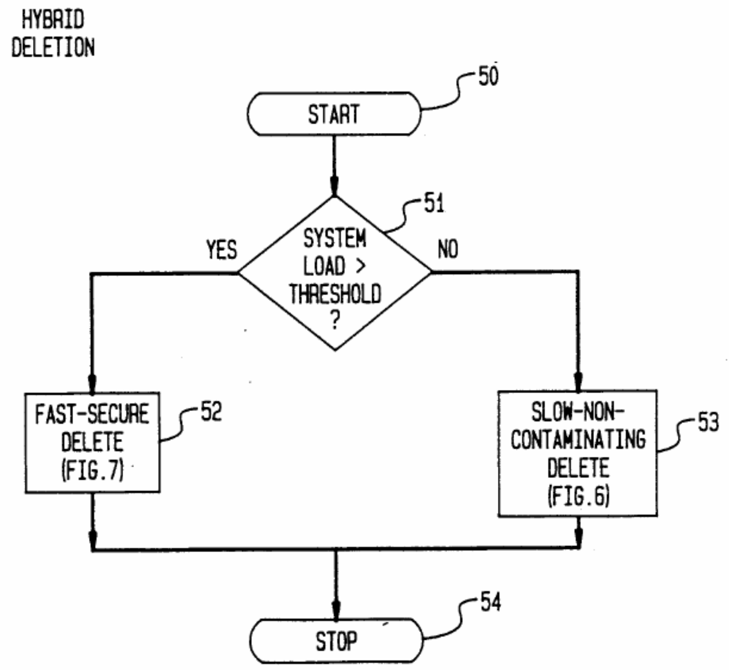
Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“vfs_cache.c”) alone and in combination**
		slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both vfs_cache.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in vfs_cache.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with vfs_cache.c would be nothing more than the predictable use of prior art elements according

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with vfs_cache.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine vfs_cache.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT D-3

Asserted Claims From **U.S. Pat. No. 5,893,120**		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both vfs_cache.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as vfs_cache.c. Moreover, one of ordinary skill in

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with vfs_cache.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with vfs_cache.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in vfs_cache.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in vfs_cache.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		linked list of records to solve a number of potential problems. For example, the removal of expired records described in vfs_cache.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by vfs_cache.c in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with vfs_cache.c . For example, both Linux 2.0.1 and vfs_cache.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold

<u>EXHIBIT D-3</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination**
		`RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD sys/kernel/vfs_cache.c v. 1.17, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain ("vfs_cache.c") alone and in combination
		RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1,

EXHIBIT D-3

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD `sys/kernel/vfs_cache.c v. 1.17`, available at http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/kern/vfs_cache.c?rev=1.18;content-type=text%2Fplain (“`vfs_cache.c`”) alone and in combination
		route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, arp.c discloses an information storage and retrieval system. For example, in arp.c, discloses a hash table of linked lists of automatically expiring data. *See*, arp.c from FreeBSD (1994) (hereinafter "arp.c") at Lines 360-448. In arp.c, the "entry" data structure is a linked list *pentry = entry->next; /* delete from linked list */ *See* arp.c at line 416. *See also*, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	arp.c discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. arp.c also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.
		
		For example, in arp.c, the "entry" data structure is a linked list *pentry = entry->next; /* delete from linked list */
		See, e.g., arp.c at line 416.
		
		arp.c includes a function that uses a hash to determine which linked to traverse:
		`hash = HASH(entry >ip);`
		`pentry = &arp_tables[hash];`
		`while (_pentry != NULL){`

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**

		if (_pentry == entry){
		entry = entry >next; / delete from linked list */
		del_timer(&entry >timer);
		restore_flags(flags);
		arp_release_entry(entry);
		return;
		420 }
		pentry = &(_pentry) >next;
		}
		
		See, *e.g.*, arp.c at Lines 195-210.
		
		If the entry is not resolved within a specific amount of time, the entry (which is a linked list element) is automatically freed (expired).
		
		/*
		 * This function is called, if an entry is not resolved in ARP_RES_TIME.
		 * Either resend a request, or give it up and free the entry.
		 */
		See, arp.c at lines 361-362.
		
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	arp.c discloses a record search means utilizing a search key to access the linked list. arp.c also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, arp.c includes a function that uses a hash to determine which linked to traverse:

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**

		hash = HASH(entry >ip);
		pentry = &arp_tables[hash];
		while (_pentry != NULL){
		if (_pentry == entry){
		entry = entry >next; / delete from linked list */
		del_timer(&entry >timer);
		restore_flags(flags);
		arp_release_entry(entry);
		return;
		420 }
		pentry = &(_pentry) >next;
		}
		See, e.g., arp.c at Lines 195-210.
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	arp.c discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. arp.c also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed
		For example, arp.c deletes an entry that meets specified criteria:
		while (*pentry != NULL)
		        {
		                if (*pentry == entry)
		                {
		                        pentry = entry->next; /
		delete from linked list */

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**

		`del_timer(&entry->timer);`
		`restore_flags(flags);`
		`arp_release_entry(entry);`
		`arp_cache_stamp++;`
		`return;`
		`}`
		`pentry = &(*pentry)->next;`
		`}`
		See, e.g., arp.c at Lines 195-210.
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	arp.c discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. arp.c also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		For example, arp.c includes a function that uses a hash to determine which linked to traverse:
		`hash = HASH(entry >ip);`
		`pentry = &arp_tables[hash];`
		`while (_pentry != NULL){`
		`if (_pentry == entry){`
		`*entry = entry >next; /* delete from linked list */`
		`del_timer(&entry >timer);`
		`restore_flags(flags);`
		`arp_release_entry(entry);`
		`return;`
		`420 }`

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**

		`pentry = &(_pentry) >next;`
		`}`
		See, e.g., arp.c at Lines 195-210.
		arp.c also includes a function that deletes an entry that meets specified criteria:
		`while (*pentry != NULL)`
		`        {`
		`                if (*pentry == entry)`
		`                {`
		`                        *pentry = entry->next;  /* delete from linked list */`
		`                        del_timer(&entry->timer);`
		`                        restore_flags(flags);`
		`                        arp_release_entry(entry);`
		`                        arp_cache_stamp++;`
		`                        return;`
		`                }`
		`                pentry = &(*pentry)->next;`
		`    }`
		See, e.g., arp.c at Lines 195-210.
		Any code that calls this function would meet this limitation because it is "utilizing the record search mean . . ."i.e., this function.
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		To the extent that arp.c does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in arp.c with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since arp.c utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of arp.c with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining arp.c with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**

		records:
		
		“*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4.
		
		See also, gcache.c at lines 246-304, defining cainsert().
		
		“*Calookup*() searches for a cached entry matching the key passed as an argument.” *See* Comer at 4.
		
		See also, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex().
		
		“*Caremove*() removes the cached entry whose key is given, if one exists.” *See* Comer at 4.
		
		See also, gcache.c at lines 355-376, defining caremove().
		
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination

		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to	arp.c combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
remove in the accessed linked list of records.	remove in the accessed linked list of records.	particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both arp.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as arp.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with arp.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with arp.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in arp.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both arp.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		combining arp.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine arp.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in arp.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining arp.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine arp.c with Thatte. Alternatively, it would also be obvious to combine arp.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-4

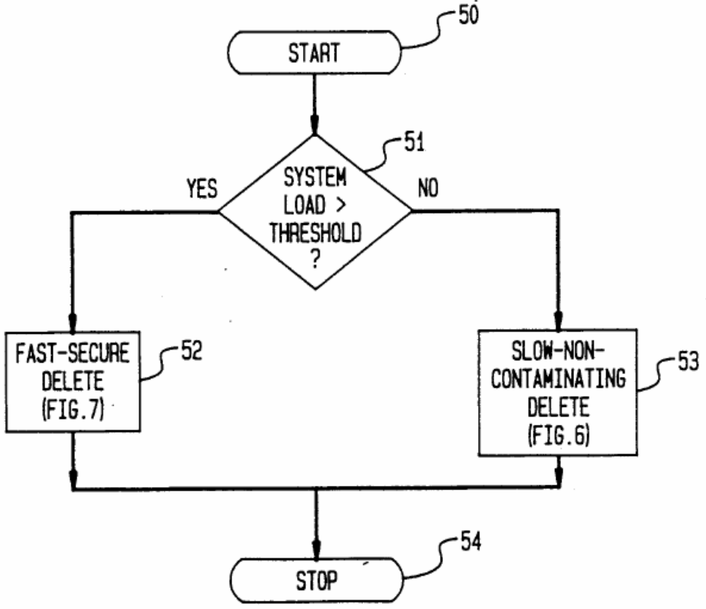
Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both arp.c and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in arp.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with arp.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with arp.c and would have seen the benefits of doing so. One such benefit, for example, is that the

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine arp.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both arp.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as arp.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with arp.c would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with arp.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in arp.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in arp.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in arp.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. *See e.g.*, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994). To the extent that dynamically determining a maximum number of expired records is not disclosed by arp.c in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with arp.c. For example, both Linux 2.0.1 and arp.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record’s last use time plus the fixed timeout value `RT_CACHE_TIMEOUT`

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, arp.c discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. arp.c also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, in arp.c, the "entry" data structure is a linked list *pentry = entry->next; /* delete from linked list */ *See*, *e.g.*, arp.c at line 416.

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination

		arp.c includes a function that uses a hash to determine which linked to traverse:
		`hash = HASH(entry >ip);`
		`pentry = &arp_tables[hash];`
		`while (_pentry != NULL){`
		`if (_pentry == entry){`
		`*entry = entry >next; /* delete from linked list */`
		`del_timer(&entry >timer);`
		`restore_flags(flags);`
		`arp_release_entry(entry);`
		`return;`
		`420 }`
		`pentry = &(_pentry) >next;`
		`}`
		
		If the entry is not resolved within a specific amount of time, the entry (which is a linked list element) is automatically freed (expired).
		
		`/*`
		` *      This function is called, if an entry is not resolved in ARP_RES_TIME.`
		` *      Either resend a request, or give it up and free the entry.`
		` */`
		See, *e.g.*, arp.c at lines 361-362.
		
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	arp.c discloses accessing a linked list of records. arp.c also discloses accessing a linked list of records having same hash address.

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		In arp.c, the “entry” data structure is a linked list *pentry = entry->next; /* delete from linked list */ *See* arp.c at line 416. *See also*, arp.c at Lines 360-448. *See also*, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	arp.c discloses identifying at least some of the automatically expired ones of the records.
		
		For example, arp.c includes a function that uses a hash to determine which linked to traverse:
		`hash = HASH(entry >ip);`
		`pentry = &arp_tables[hash];`
		`while (_pentry != NULL){`
		`if (_pentry == entry){`
		`*entry = entry >next; /* delete from linked list */`
		`del_timer(&entry >timer);`
		`restore_flags(flags);`
		`arp_release_entry(entry);`
		`return;`
		`420 }`
		`pentry = &(_pentry) >next;`
		`}`
		
		See, e.g., arp.c at Lines 195-210.
		
		See also, arp.c at Lines 360-448. *See also*, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

[3c] removing at least some of the automatically	[7c] removing at least some of the automatically	arp.c discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed.

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination

expired records from the linked list when the linked list is accessed.	expired records from the linked list when the linked list is accessed, and	For example, arp. deletes an entry that meets specified criteria:
		`while (*pentry != NULL)`
		`{`
		`if (*pentry == entry)`
		`{`
		`*pentry = entry->next;   /*`
		`delete from linked list */`
		`del_timer(&entry->timer);`
		`restore_flags(flags);`
		`arp_release_entry(entry);`
		`arp_cache_stamp++;`
		`return;`
		`}`
		`pentry = &(*pentry)->next;`
		`}`
		See, *e.g.*, arp.c at Lines 195-210.
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	arp.c discloses inserting, retrieving or deleting one of the records from the system following the step of removing.
		For example, arp.c deletes an entry that meets specified criteria after arp.c traverses the linked list searching for the record of that criteria:
		`while (*pentry != NULL)`
		`{`
		`if (*pentry == entry)`
		`{`

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**

		`*pentry = entry->next;  /*`
		`delete from linked list */`
		`del_timer(&entry->timer);`
		`restore_flags(flags);`
		`arp_release_entry(entry);`
		`arp_cache_stamp++;`
		`return;`
		`}`
		`pentry = &(*pentry)->next;`
		`}`
		See, *e.g.*, arp.c at Lines 195-210.
		See also, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994).

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	arp.c combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both arp.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as arp.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with arp.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with arp.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in arp.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both arp.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining arp.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine arp.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in arp.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining arp.c with the

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine arp.c with Thatte. Alternatively, it would also be obvious to combine arp.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41.

EXHIBIT D-4

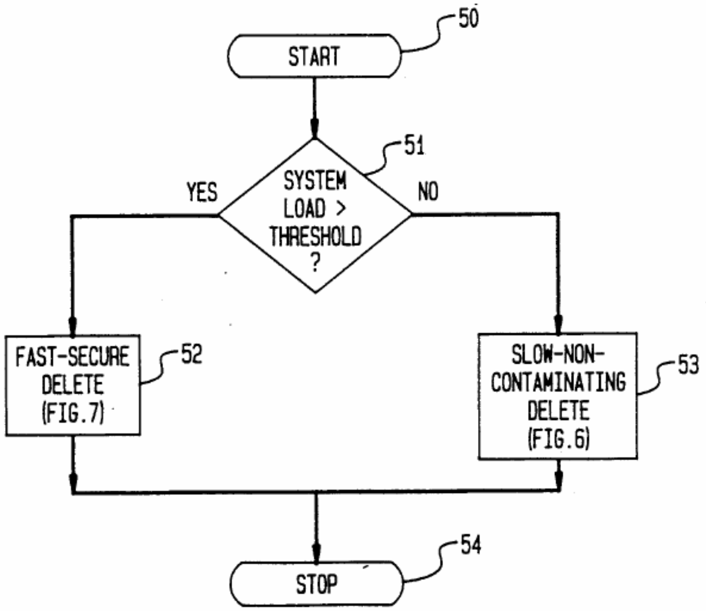
Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		Id. at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both arp.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in arp.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with arp.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with arp.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine arp.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage*

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		Collector at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both arp.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as arp.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with arp.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with arp.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in arp.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in arp.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in arp.c can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter "arp.c"), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. *See e.g.*, arp.c at Lines 360-448. *See also*, arp.c from Linux 1.1.20 (1994). To the extent that dynamically determining a maximum number of expired records is not disclosed by arp.c in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with arp.c. For example, both Linux 2.0.1 and arp.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination
		`rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`.

EXHIBIT D-4

Asserted Claims From U.S. Pat. No. 5,893,120		**arp.c from FreeBSD (1994) (hereinafter “arp.c”), *See also*, arp.c from Linux 1.1.20 (1994) alone and in combination**
		Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record’s last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record’s reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination

1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, wavelan_cs.c discloses an information storage and retrieval system.
		For example, an information storage and retrieval system disclosed by wavelan_cs.c is a linked list:
		`  /* Remove the interface data from the linked list */`
		`  if(dev_list == link)`
		`    dev_list = link->next;`
		See, wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) at Lines 4630-4635. *See also*, wavelan_cs.c from Linux 2.4.26.
		See *also*, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	wavelan_cs.c discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. wavelan_cs.c also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.
		For example, wavelan_cs.c includes a function that deletes an instance of a driver from the linked list if the device is released. Thus, releasing the device causes the device to automatically expire.
		`/*`
		` * This deletes a driver "instance".  The device is`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		de-registered with * Card Services. If it has been released, all local data structures * are freed. Otherwise, the structures will be freed when the device * is released. */ *See, e.g.*, wavelan_cs.c at Lines 4595-4600. The data structure is a linked list: /* Remove the interface data from the linked list */ if(dev_list == link) dev_list = link->next; *See* wavelan_cs.c at Lines 4630-4635. *See also*, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	wavelan_cs.c discloses a record search means utilizing a search key to access the linked list. wavelan_cs.c also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.
		
		For example, wavelan_cs.c includes functionality to use a pointer to traverse a linked list.
		/* Remove the interface data from the linked list */
		  if(dev_list == link)
		    dev_list = link->next;
		  else
		    {
		      dev_link_t * prev = dev_list;

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination**

		`while((prev != (dev_link_t *) NULL) && (prev->next != link))`
		`    prev = prev->next;`
		
		`if(prev == (dev_link_t *) NULL)`
		`  {`
		`#ifdef DEBUG_CONFIG_ERRORS`
		`      printk(KERN_WARNING "wavelan_detach : Attempting to remove a nonexistent device.\n");`
		`#endif`
		`      return;`
		`  }`
		`prev->next = link->next`
		See, e.g., wavelan_cs.c at Lines 4632-4644.
		
		See also, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	wavelan_cs.c discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. wavelan_cs.c also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed.
		
		For example, wavelan_cs.c includes the functionality to remove expired records from the linked list:
		`/* Remove the interface data from the linked list */`
		`  if(dev_list == link)`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		`    dev_list = link->next;` *See*, *e.g.*, wavelan_cs.c at Lines 4632-4644. *See also*, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	wavelan_cs.c discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. wavelan_cs.c also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		
		For example, wavelan_cs.c includes functionality to use a pointer to traverse a linked list. Further, wavelan_cs.c will remove records from the linked list as it traverses the linked list. Any code that calls this function would "utilize the record search means."
		
		`/* Remove the interface data from the linked list */`
		`  if(dev_list == link)`
		`    dev_list = link->next;`
		`  else`
		`    {`
		`      dev_link_t *      prev = dev_list;`
		
		`      while((prev != (dev_link_t *) NULL) && (prev-`
		`>next != link))`
		`        prev = prev->next;`
		
		`      if(prev == (dev_link_t *) NULL)`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination

		{
		#ifdef DEBUG_CONFIG_ERRORS
		printk(KERN_WARNING "wavelan_detach : Attempting to remove a nonexistent device.\n");
		#endif
		return;
		}
		prev->next = link->next
		See, *e.g.*, wavelan_cs.c at Lines 4632-4644.
		
		See, *e.g.*, wavelan_cs.c at Lines 4596-4678.
		
		To the extent that wavelan_cs.c does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in wavelan_cs.c with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See*, *e.g.*, Comer at 3-10. For example, since wavelan_cs.c utilizes a linked list for

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of wavelan_cs.c with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining wavelan_cs.c with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4.

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination

		See also, gcache.c at lines 355-376, defining caremove().
		
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		“In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.” *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination**

		`if (caisold(pcb,pce)) {`
		`++pcb->cb_tos;`
		`caunlink(pcb,ix);`
		`return(NULL_IX);`
		`} else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	wavelan_cs.c combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both wavelan_cs.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as wavelan_cs.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with wavelan_cs.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with wavelan_cs.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in wavelan_cs.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both wavelan_cs.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining wavelan_cs.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine wavelan_cs.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in wavelan_cs.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining wavelan_cs.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine wavelan_cs.c with Thatte. Alternatively, it would also be obvious to combine wavelan_cs.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT D-5

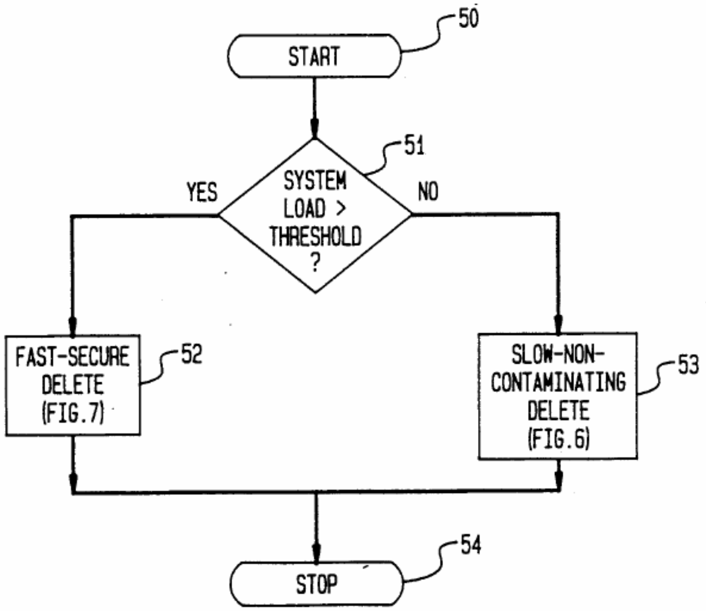
Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64,

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both wavelan_cs.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in wavelan_cs.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with wavelan_cs.c would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with wavelan_cs.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine wavelan_cs.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both wavelan_cs.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as wavelan_cs.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with wavelan_cs.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with wavelan_cs.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in wavelan_cs.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in wavelan_cs.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in wavelan_cs.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by wavelan_cs.c in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with wavelan_cs.c. For example, both Linux 2.0.1 and wavelan_cs.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically.

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination**
		Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination**
		record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired.

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than the maximum number of records that the function rt_cache_add can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the	To the extent the preamble is a limitation, wavelan_cs.c discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. wavelan_cs.c also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, wavelan_cs.c includes a function that deletes an instance of a driver from the linked list if the device is released. Thus, releasing the

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination

	method comprising the steps of:	device causes the record to automatically expire.
		`/*`
		` * This deletes a driver "instance".  The device is`
		`de-registered with`
		` * Card Services.  If it has been released, all`
		`local data structures`
		` * are freed.  Otherwise, the structures will be`
		`freed when the device`
		` * is released.`
		` */`
		See, *e.g.*, wavelan_cs.c at Lines 4595-4600.
		The data structure is a linked list:
		`  /* Remove the interface data from the linked list`
		`*/`
		`  if(dev_list == link)`
		`    dev_list = link->next;`
		See, *e.g.*, wavelan_cs.c at Lines 4630-4635.
		See also, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	wavelan_cs.c discloses accessing a linked list of records. wavelan_cs.c also discloses accessing a linked list of records having same hash address.
		For example, wavelan_cs.c includes functionality to use a pointer to traverse a linked list.
		`/* Remove the interface data from the linked list */`
		`  if(dev_list == link)`
		`    dev_list = link->next;`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination

		`else`
		`{`
		`dev_link_t *      prev = dev_list;`
		
		`while((prev != (dev_link_t *) NULL) && (prev->next != link))`
		`prev = prev->next;`
		
		`if(prev == (dev_link_t *) NULL)`
		`{`
		`#ifdef DEBUG_CONFIG_ERRORS`
		`printk(KERN_WARNING "wavelan_detach : Attempting to remove a nonexistent device.\n");`
		`#endif`
		`return;`
		`}`
		
		`prev->next = link->next`
		See, *e.g.*, wavelan_cs.c at Lines 4632-4644.
		
		See also, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	wavelan_cs.c discloses identifying at least some of the automatically expired ones of the records.
		
		For example, wavelan_cs.c includes functionality to use a pointer to traverse a linked list.
		`/* Remove the interface data from the linked list */`
		`if(dev_list == link)`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination

		`    dev_list = link->next;`
		`  else`
		`    {`
		`      dev_link_t *      prev = dev_list;`
		
		`      while((prev != (dev_link_t *) NULL) && (prev-`
		`>next != link))`
		`        prev = prev->next;`
		
		`      if(prev == (dev_link_t *) NULL)`
		`        {`
		`#ifdef DEBUG_CONFIG_ERRORS`
		`          printk(KERN_WARNING "wavelan_detach :`
		`Attempting to remove a nonexistent device.\n");`
		`#endif`
		`          return;`
		`        }`
		
		`      prev->next = link->next`
		See, e.g., wavelan_cs.c at Lines 4632-4644.
		
		For example, wavelan_cs.c includes a function that deletes an instance of a driver from the linked list if the device is released. Thus, releasing the device causes the record to automatically expire.
		
		`/*`
		` * This deletes a driver "instance".  The device is`
		`de-registered with`
		` * Card Services.  If it has been released, all`
		`local data structures`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		* are freed. Otherwise, the structures will be freed when the device * is released. */ *See*, *e.g.*, wavelan_cs.c at Lines 4595-4600. *See also*, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	wavelan_cs.c discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed.
		
		For example, wavelan_cs.c includes functionality to use a pointer to traverse a linked list. Further, wavelan_cs.c will remove records from the linked list as it traverses the linked list.
		
		/* Remove the interface data from the linked list */
		  if(dev_list == link)
		    dev_list = link->next;
		  else
		    {
		      dev_link_t * prev = dev_list;
		
		      while((prev != (dev_link_t *) NULL) && (prev->next != link))
		        prev = prev->next;
		
		      if(prev == (dev_link_t *) NULL)
		        {
		#ifdef DEBUG_CONFIG_ERRORS
		          printk(KERN_WARNING "wavelan_detach :

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination

		`Attempting to remove a nonexistent device.\n");`
		`#endif`
		`          return;`
		`        }`
		`      prev->next = link->next`
		See, *e.g.*, wavelan_cs.c at Lines 4632-4644.
		
		See also, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26.

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	wavelan_cs.c discloses inserting, retrieving or deleting one of the records from the system following the step of removing.
		
		For example, wavelan_cs.c includes functionality to use a pointer to traverse a linked list. Further, wavelan_cs.c will remove records from the linked list as it traverses the linked list. The deletion will occur after wavelan_cs.c traverses through the element.
		
		`/* Remove the interface data from the linked list */`
		`  if(dev_list == link)`
		`    dev_list = link->next;`
		`  else`
		`    {`
		`      dev_link_t *      prev = dev_list;`
		
		`      while((prev != (dev_link_t *) NULL) && (prev-`
		`>next != link))`
		`        prev = prev->next;`
		
		`      if(prev == (dev_link_t *) NULL)`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination**

		{
		#ifdef DEBUG_CONFIG_ERRORS
		printk(KERN_WARNING "wavelan_detach : Attempting to remove a nonexistent device.\n");
		#endif
		return;
		}
		prev->next = link->next
		See, e.g., wavelan_cs.c at Lines 4632-4644.
		
		See also, wavelan_cs.c at Lines 4596-4678. *See also*, wavelan_cs.c from Linux 2.4.26

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	wavelan_cs.c combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks:

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both wavelan_cs.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as wavelan_cs.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with wavelan_cs.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with wavelan_cs.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in wavelan_cs.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both wavelan_cs.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining wavelan_cs.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine wavelan_cs.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in wavelan_cs.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		wavelan_cs.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine wavelan_cs.c with Thatte. Alternatively, it would also be obvious to combine wavelan_cs.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-5

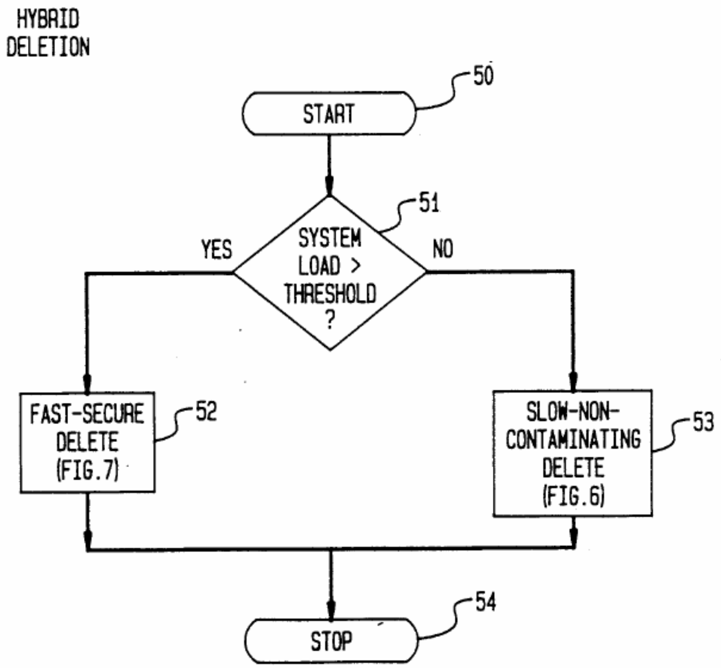
Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both wavelan_cs.c and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in wavelan_cs.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with wavelan_cs.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with wavelan_cs.c

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		**wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination**
		and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine wavelan_cs.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both wavelan_cs.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as wavelan_cs.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with wavelan_cs.c would be nothing more than the predictable use

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with wavelan_cs.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in wavelan_cs.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in wavelan_cs.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in wavelan_cs.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by wavelan_cs.c in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with wavelan_cs.c. For example, both Linux 2.0.1 and wavelan_cs.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1`

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter “wavelan_cs.c”) alone and in combination
		function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record’s last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record’s reference count is zero. *See* Linux

EXHIBIT D-5

Asserted Claims From U.S. Pat. No. 5,893,120		wavelan_cs.c from FreeBSD (1995) (hereinafter "wavelan_cs.c") alone and in combination
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-6

Asserted Claims From **U.S. Pat. No. 5,893,120**		**LISP alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, LISP discloses an information storage and retrieval system. For example, "This paper should be useful to readers interested in data structures and their applications in compiler construction, language design, and database management." Jacques Cohen, *Garbage Collection of Linked Data Structures*, Computing Surveys, 341, 342 (hereinafter "Cohen"). "This model consists of a memory, *i.e.* a one-dimensional array of words, each of which is large enough to hold (at least) the representation of a nonnegative integer which is an index into that array." Henry G. Baker, *List Processing in Real Time on a Serial Computer*, Communications of the ACM 21, 280, second page, (April 1978) (hereinafter "Baker"). "There are two fundamental kinds of data in LISP: list cells and atoms . . . CAR(x) and CDR(x) return the car and cdr components of the list cell x, respectively." Baker at 2. "If the method is used for the management of a large database residing on secondary storage." Baker at 6. "We conceive of a huge database having millions of records, which may contain pointers to other records, being managed by our algorithm." Baker at 12.
[1a] a linked list to store	[5a] a hashing means to	LISP discloses a linked list to store and provide access to records stored in a

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		**LISP alone and in combination**
and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	memory of the system, at least some of the records automatically expiring. LISP also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, "Of the hundreds of thousands of computer languages which have been invented, there is one particular family of languages whose common ancestor was the original LISP, developed by McCarthy and others in the late 1950's. [LISP History] These languages are generally characterized by a simple, fully parenthesized ("Cambridge Polish") syntax; the ability to manipulate general, linked-list data structures; a standard representation for programs of the language in terms of these structures; and an interactive programming system based on an interpreter for the standard representation. Examples of such languages are LISP 1.5 [LISP 1.5M], MacLISP [Moon], InterLISP [Teiteiman], CONNIVER UIcDermott and Sussman], QM [Rul1fson], PLASNA [Smith and Hewitt] [Hewitt and Smith], and SCHUIE [SCHEME] [Revised Report]. We will call this family the LISP-like languages." Guy Lewis Steele, Jr., *The Art of the Interpreter or The Modularity Complex* (Parts Zero, One, and Two), Massachusetts Institute of Technology AI Memo No. 453 at 2 (May 1978). (Hereinafter "Steele"). "A concise and unified view of the numerous existing algorithms for performing garbage collection of linked data structures is presented." Cohen Abstract.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		“The primary list processing language in use today is LISP.” Baker at 2. “We conceive of a huge database having millions of records, which may contain pointers to other records, being managed by our algorithm.” Baker at 12. ”A cell becomes unused, or garbage, when it can no longer be accessed through any pointer fields of any reachable cell.” Cohen at 342. “the property list can be found by looking in a hash table, using the address of the list cell as the key.” Baker at 10. “Our copying scheme gives each semispace its own hash table, and when a cell is copied over into to space, its property list pointer is entered in the "to" table under the cell's new address. Then when the copied cell is encountered by the "scan" pointer, its property list pointer is updated along with its normal components.” Baker at 10. There are two well-known approaches to solving the problem of collisions within a hash table, which occur whenever two entries “hash” or are assigned to the same “bucket” within the hash table. The computer programmer may store the records external to the hash table—that is, using memory separate from the memory allocated to the hash table—or he may store the records internal to the hash table—that is, using memory that is allocated to other buckets within the hash table. Using external memory is termed “external chaining,” while using internal memory is termed “open addressing.” The applicant has conceded that both forms of collision resolution are known to those of ordinary skill in the art. *See, e.g.*, ‘120 patent at 1:53-57 (describing

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		linear probing—a type of open addressing—as being "often used" for "collision resolution"); 1:58-2:6 (citing to several prior art resources that describe external chaining as using linked lists). Double hashing is another form of open addressing. It would have been obvious to one skilled in the art to apply the teachings in LISP to a hash table which resolves collisions using external chaining with linked lists. The method of LISP is a method for processing and garbage collecting on linked structures generally, which includes externally chained records. Externally chained records would still need a method of memory management, so it would be obvious to use LISP as a method of memory management and list processing on externally chained records with the same hash address.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	LISP discloses a record search means utilizing a search key to access the linked list. LISP also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, "This model consists of a memory, *i.e.* a one-dimensional array of words, each of which is large enough to hold (at least) the representation of a nonnegative integer which is an index into that array." Baker at 2. "the property list can be found by looking in a hash table, using the address of the list cell as the key." Baker at 10. As discussed in [1a/5a], it would have been obvious to one of ordinary skill in

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		**LISP alone and in combination**
		the art to use LISP for list processing and memory management on externally chained structures. As such, the search means utilizing the search key would be accessing a linked list of records beginning at the same hash address.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	LISP discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. LISP also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, "For example, in LISP, the function *cons* also calls the garbage collector." Cohen at 342. "Baker's modification is such that each time a cell is requested (*i.e.*, a *cons* is requested), a fixed number of cells, k, are moved from one semispace to the other." Cohen at 355. "The amount of storage and time used by a real-time list processing system can be compared with that used by a classical list processing system using garbage collection on tasks not requiring bounded response times." Baker at 11. As discussed in [1a/5a], it would have been obvious to one of ordinary skill in the art to use LISP for list processing and memory management on externally chained structures. In such a system, the probe that resulted from a collision would occur on the linked list used to resolve the collision. As such, the expired records from the linked list would be removed when the linked list is accessed.

EXHIBIT D-6

Asserted Claims From **U.S. Pat. No. 5,893,120**		**LISP alone and in combination**
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	LISP discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. LISP also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, "The moving of k cells during a *cons* corresponds to the tracing of that many cells in classical garbage collection. By distributing some of the garbage collection tasks during list processing, Baker's method provides a guarantee that actual garbage collection cannot last more than a fixed (tolerable) amount of time." Cohen at 355. "A real-time list processing system is presented which continuously reclaims garbage." Baker Abstract. "In order to convert MFYCA into a real-time algorithm, we force the mark ratio m to be constant by changing CONS so that it does k iterations of garbage collection before performing each allocation." Baker at 4. "There is another problem caused by interleaving garbage collection with normal list processing." Baker at 4. "garbage collection in our real-time system is almost identical to that in the

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		MFYCA system, except that it is done incrementally during calls to CONS. In other words, the user program pays for the cost of cell's reclamation at the time the cell is created by tracing some other cell." Baker at 11. "We have exhibited a method for doing list-processing on a serial computer in a real-time environment . . . Our real time scheme is strikingly similar to the incremental garbage collector proposed independently by Barbacci." Baker at 13. As discussed in [1a/5a], it would have been obvious to one of ordinary skill in the art to use LISP for list processing and memory management on externally chained structures. In such a system, the probe that resulted from a collision would occur on the linked list used to resolve the collision. As such, the expired records from the linked list would be removed when the linked list is accessed. To the extent that LISP does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		how to, combine the system disclosed in LISP with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since LISP utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of LISP with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining LISP with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination

		See also, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex().
		
		"*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4.
		
		See also, gcache.c at lines 355-376, defining caremove().
		
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned."

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination

		See Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	LISP discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. For example, "Baker's modification is such that each time a cell is requested (*i.e.*, a *cons* is requested), a fixed number of cells, k, are moved from one semispace to the other." Cohen at 355. "With a little more effort, k can even be made variable in our method, thus allowing the program to dynamically optimize its space-time tradeoff." Baker at 6.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		Lisp combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20).

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		Id. at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both LISP and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as LISP. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with LISP nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with LISP and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in LISP with the means for dynamically determining maximum number for the record search means to

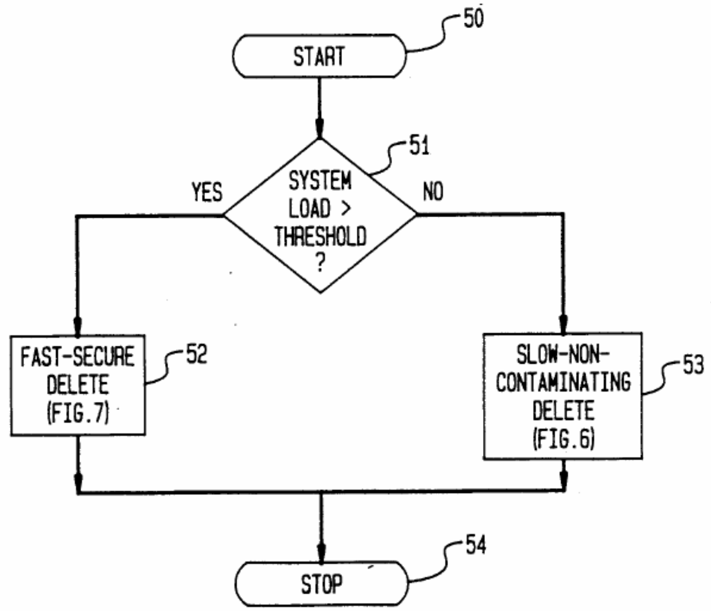
EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both LISP and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining LISP with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine LISP with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in LISP can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining LISP with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		patent provides motivations to combine LISP with Thatte. Alternatively, it would also be obvious to combine LISP with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both LISP and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in LISP. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		patent’s deletion decision procedure with LISP would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with LISP and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine LISP with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage*

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		Collector at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both LISP and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as LISP. Moreover, one of ordinary skill in the art

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with LISP would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with LISP and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in LISP to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in LISP with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		records described in LISP can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine LISP (*e.g.* the system disclosed in Baker or Cohen) with Thatte, Dirks, the '663 patent, and/or the Opportunistic Garbage Collection Articles, in addition to motivations within the text of Baker or Cohen. Baker at 1 ("Third, processing had to be halted periodically to reclaim storage by a long process know as garbage collection, which laboriously traced every accessible cell so that those inaccessible cells could be recycled. . . . This paper presents a solution to the third problem . . . and removes the roadblock to their more general use."); Cohen at 342 ("A most vexing aspect of garbage collection is that program execution comes to a halt while the collector attempts to reclaim storage space."). To the extent that dynamically determining a maximum number of expired records is not disclosed by LISP in combination with Dirks, Thatte, the '663

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with LISP. For example, both Linux 2.0.1 and LISP describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, LISP discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. LISP also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, "This paper should be useful to readers interested in data structures and their applications in compiler construction, language design, and database management." Cohen at 342. "This model consists of a memory, *i.e*. a one-dimensional array of words, each

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		of which is large enough to hold (at least) the representation of a nonnegative integer which is an index into that array." Baker at 2. "There are two fundamental kinds of data in LISP: list cells and atoms . . . . CAR(x) and CDR(x) return the car and cdr components of the list cell x, respectively." Baker at 2. "If the method is used for the management of a large database residing on secondary storage." Baker at 6. "We conceive of a huge database having millions of records, which may contain pointers to other records, being managed by our algorithm." Baker at 12. "the property list can be found by looking in a hash table, using the address of the list cell as the key." Baker at 10. "Our copying scheme gives each semispace its own hash table, and when a cell is copied over into to space, its property list pointer is entered in the "to" table under the cell's new address. Then when the copied cell is encountered by the "scan" pointer, its property list pointer is updated along with its normal components." Baker at 10. "Of the hundreds of thousands of computer languages which have been invented, there is one particular family of languages whose common ancestor was the original LISP, developed by McCarthy and others in the late 1950's. [LISP History] These languages are generally characterized by a simple, fully

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		parenthesized ("Cambridge Polish") syntax; the ability to manipulate general, linked-list data structures; a standard representation for programs of the language in terms of these structures; and an interactive programming system based on an interpreter for the standard representation. Examples of such languages are LISP 1.5 [LISP 1.5M], MacLISP [Moon], InterLISP [Teiteiman], CONNIVER McDermott and Sussman], QM [Rulfson], PLASNA [Smith and Hewitt] [Hewitt and Smith], and SCHUIE [SCHEME] [Revised Report]. We will call this family the LISP-like languages." Steele at 2. "A cell becomes unused, or garbage, when it can no longer be accessed through any pointer fields of any reachable cell." Cohen at 342. It would have been obvious to one skilled in the art to apply the teachings in LISP to a hash table which resolves collisions using external chaining with linked lists. The method of LISP is a method for processing and garbage collecting on linked structures generally, which includes externally chained records. Externally chained records would still need a method of memory management, so it would be obvious to use LISP as a method of memory management and list processing on externally chained records with the same hash address.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	LISP discloses accessing a linked list of records. LISP also discloses accessing a linked list of records having same hash address. For example, "For example, in LISP, the function *cons* also calls the garbage collector." Cohen at 342.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		"The amount of storage and time used by a real-time list processing system can be compared with that used by a classical list processing system using garbage collection on tasks not requiring bounded response times." Baker at 11. "the property list can be found by looking in a hash table, using the address of the list cell as the key." Baker at 10. "This model consists of a memory, *i.e*. a one-dimensional array of words, each of which is large enough to hold (at least) the representation of a nonnegative integer which is an index into that array." Baker at 2. As discussed in [3/7], it would have been obvious to one of ordinary skill in the art to use LISP for list processing and memory management on externally chained structures having the same hash address.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	LISP discloses identifying at least some of the automatically expired ones of the records. LISP also discloses identifying at least some of the automatically expired ones of the records. For example, "For example, in LISP, the function *cons* also calls the garbage collector." Cohen at 342. "Baker's modification is such that each time a cell is requested (*i.e.*, a *cons* is requested), a fixed number of cells, k, are moved from one semispace to the other." Cohen at 355.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		"The amount of storage and time used by a real-time list processing system can be compared with that used by a classical list processing system using garbage collection on tasks not requiring bounded response times." Baker at 11. "A cell becomes unused, or garbage, when it can no longer be accessed through any pointer fields of any reachable cell." Cohen at 342.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	LISP discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. LISP also discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, "For example, in LISP, the function *cons* also calls the garbage collector." Cohen at 342. "In order to convert MFYCA into a real-time algorithm, we force the mark ratio m to be constant by changing CONS so that it does k iterations of garbage collection before performing each allocation." Baker at 4. "There is another problem caused by interleaving garbage collection with normal list processing." Baker at 4. "garbage collection in our real-time system is almost identical to that in the MFYCA system, except that it is done incrementally during calls to CONS. In

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		other words, the user program pays for the cost of cell's reclamation at the time the cell is created by tracing some other cell." Baker at 11. "We have exhibited a method for doing list-processing on a serial computer in a real-time environment . . . . Our real time scheme is strikingly similar to the incremental garbage collector proposed independently by Barbacci." Baker at 13.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	LISP discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, "For example, in LISP, the function *cons* also calls the garbage collector." Cohen at 342. "In order to convert MFYCA into a real-time algorithm, we force the mark ratio m to be constant by changing CONS so that it does k iterations of garbage collection before performing each allocation." Baker at 4. "There is another problem caused by interleaving garbage collection with normal list processing." Baker at 4. "garbage collection in our real-time system is almost identical to that in the MFYCA system, except that it is done incrementally during calls to CONS. In other words, the user program pays for the cost of cell's reclamation at the time the cell is created by tracing some other cell." Baker at 11.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		"We have exhibited a method for doing list-processing on a serial computer in a real-time environment . . . . Our real time scheme is strikingly similar to the incremental garbage collector proposed independently by Barbacci." Baker at 13.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	LISP discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, "Baker's modification is such that each time a cell is requested (*i.e.*, a *cons* is requested), a fixed number of cells, k, are moved from one semispace to the other." Cohen at 355. "With a little more effort, k can even be made variable in our method, thus allowing the program to dynamically optimize its space-time tradeoff." Baker at 6. Lisp combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both LISP and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as LISP. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with LISP nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with LISP and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in LISP with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both LISP and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining LISP with Thatte would be nothing more than the predictable use of

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine LISP with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in LISP can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining LISP with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine LISP with Thatte. Alternatively, it would also be obvious to combine LISP with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-6

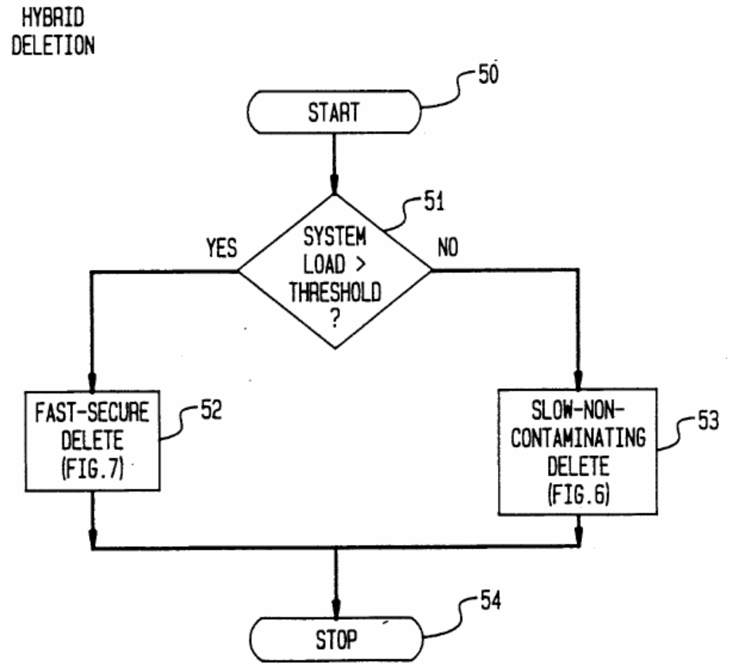
Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both LISP and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in LISP. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with LISP would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-6

Asserted Claims From **U.S. Pat. No. 5,893,120**		**LISP alone and in combination**
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with LISP and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine LISP with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		**LISP alone and in combination**
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both LISP and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as LISP. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with LISP would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with LISP and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in LISP to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in LISP with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in LISP can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. Thus, the ’120 patent provides motivations to combine LISP (*e.g.* Baker or Cohen) with Thatte, Dirks, the '663 patent, and/or the Opportunistic Garbage Collection Articles, in addition to motivations within the text of Baker or Cohen. Baker at 1 (“Third, processing had to be halted periodically to reclaim storage by a long process know as garbage collection, which laboriously traced every accessible cell so that those inaccessible cells could be recycled. . . . This paper presents a solution to the third problem . . . and removes the roadblock to their more general use.”); Cohen at 342 (“A most vexing aspect of garbage collection is that program execution comes to a halt while the collector attempts to reclaim storage space.”). To the extent that dynamically determining a maximum number of expired records is not disclosed by LISP in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with LISP. For example,

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		both Linux 2.0.1 and LISP describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		`rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not

EXHIBIT D-6

Asserted Claims From U.S. Pat. No. 5,893,120		LISP alone and in combination
		limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, FreeBSD 2.0.5 discloses an information storage and retrieval system. For example, in kern_proc.c and proc.h, FreeBSD 2.0.5 discloses a hash table of linked lists of automatically expiring data. See, e.g., struct pgrp defined at lines 61-70 of proc.h and struct proc defined at lines 72-172 of proc.h. Excerpts are below: 61 /* 62 * One structure allocated per process group. 63 */ 64 struct pgrp { 65 struct pgrp *pg_hforw; /* Forward link in hash bucket. */ 66 struct proc *pg_mem; /* Pointer to pgrp members. */ 67 struct session *pg_session; /* Pointer to session. */ 68 pid_t pg_id; /* Pgrp id. */ 69 int pg_jobc; /* # procs qualifying pgrp for job control */ 70 }; 72 /* 73 * Description of a process. 74 * 75 * This structure contains the information needed to manage a thread of 76 * control, known in UN*X as a process; it has references to substructures 77 * containing descriptions of things that the process uses, but may share 78 * with related processes. The process structure and the substructures 79 * are always addressible except for those marked "(PROC ONLY)" below, 80 * which might be addressible only on a processor on which the process

[1] Publicly available as of June 10, 1995; available at ftp://ftp-archive.freebsd.org/pub/FreeBSD-Archive/old-releases/i386/2.0.5-RELEASE/src/.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		81 * is running. 82 */ 83 struct proc { 84 struct proc *p_forw; /* Doubly-linked run/sleep queue. */ 85 struct proc *p_back; 86 struct proc *p_next; /* Linked list of active procs */ 87 struct proc **p_prev; /* and zombies. */
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	FreeBSD discloses “a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring” and “a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.” For example, the pidhash[] and pgrphash[] structures defined in param.c meet the “hashing means” limitation. 206 struct proc *pidhash[PIDHSZ]; 207 struct pgrp *pgrphash[PIDHSZ]; Also, FreeBSD defines the pgrp structure as including a forward link in the hash bucket as well as a pointer to a linked list of proc structures. This is an example of how FreeBSD meets the “linked list” and “external chaining” limitations, as shown in the excerpts from proc.h below. 64 struct pgrp { 65 struct pgrp *pg_hforw; /* Forward link in hash bucket. */ 66 struct proc *pg_mem; /* Pointer to pgrp members. */ 67 struct session *pg_session; /* Pointer to session. */ 68 pid_t pg_id; /* Pgrp id. */ 69 int pg_jobc; /* # procs qualifying pgrp for job control */ 70 };

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		83 struct proc { 84 struct proc *p_forw; /* Doubly-linked run/sleep queue. */ 85 struct proc *p_back; 86 struct proc *p_next; /* Linked list of active procs */ 87 struct proc **p_prev; /* and zombies. */ Examples of how FreeBSD uses the hashing technique with external chaining can be found in the enterpgrp() function in kern_proc.c, such as the following: 230 pgrp->pg_hforw = pgrphash[n = PIDHASH(pgid)]; 231 pgrphash[n] = pgrp; The function that calls enterpgrp() passes in a proc structure, as shown in lines 175-79. 175 int 176 enterpgrp(p, pgid, mksess) 177 register struct proc *p; 178 pid_t pgid; 179 int mksess; Code within the enterpgrp() structure unlinks the proc from its old process group, as shown below in lines 248-53 of kern_proc.c. Also, enterpgrp() calls pgdelete() if the process group is empty, as shown in lines 261-62. Depending on claim construction, these are two examples of automatic expiration. 245 /* 246 * unlink p from old process group 247 */

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		if (*pp == p) {
		*pp = p->p_pgrpnxt;
		break;
		}
		}
		
		/*
		* delete old if empty
		*/
		if (p->p_pgrp->pg_mem == 0)
		pgdelete(p->p_pgrp);

[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	FreeBSD discloses "a record search means utilizing a search key to access the linked list" and "a record search means utilizing a search key to access a linked list of records having the same hash address."
		
		The following code from the enterpgrp() function in kern_proc.c is an example of accessing a linked list of records having the same hash address and using a search key to access a linked list. The [n] index is an example of a search key. The enterpgrp() function is an example of a "record search means" as claimed.
		
		pgrp->pg_hforw = pgrphash[n = PIDHASH(pgid)];
		pgrphash[n] = pgrp;

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of	FreeBSD discloses "the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed" and "the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed," as claimed.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

list when the linked list is accessed, and	records when the linked list is accessed, and	For example, code within the enterpgrp() structure unlinks the proc from its old process group, as shown below in lines 248-53 of kern_proc.c. Also, enterpgrp() calls pgdelete() if the process group is empty, as shown in lines 261-62. These are two examples of automatic expiration.
		245 /*
		246 * unlink p from old process group
		247 */
		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {
		250 *pp = p->p_pgrpnxt;
		251 break;
		252 }
		253 }
		258 /*
		259 * delete old if empty
		260 */
		261 if (p->p_pgrp->pg_mem == 0)
		262 pgdelete(p->p_pgrp);

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of	FreeBSD discloses "means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list " and "meals [*sic* "means"], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records," as claimed. An example of a "means utilizing the record search means" can be found in kern_prot.c. For example, the setsid() function calls enterpgrp(), as shown below at line 196.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

	records.	186 int
		187 setsid(p, uap, retval)
		188 register struct proc *p;
		189 struct args *uap;
		190 int *retval;
		191 {
		192
		193 if (p->p_pgid == p->p_pid || pgfind(p->p_pid)) {
		194 return (EPERM);
		195 } else {
		196 (void)enterpgrp(p, p->p_pid, 1);
		197 *retval = p->p_pid;
		198 return (0);
		199 }
		200 }
		
		An example of “retrieving” can be found in enterpgrp(), in the for loop found at lines 248-53. Depending on claim construction, an example of “removing” and “deleting” can be found in the call to pgdelete() at line 262, and the operation of pgdelete() at lines 300-22. An example of “inserting” can be found at lines 266-68. Each of these steps is performed within enterpgrp() and “at the same time,” as recited in the claims.
		
		245 /*
		246 * unlink p from old process group
		247 */
		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {
		250 *pp = p->p_pgrpnxt;
		251 break;

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		}
		}
		
		/*
		* delete old if empty
		*/
		if (p->p_pgrp->pg_mem == 0)
		pgdelete(p->p_pgrp);
		/*
		* link into new one
		*/
		p->p_pgrp = pgrp;
		p->p_pgrpnxt = pgrp->pg_mem;
		pgrp->pg_mem = p;
		return (0);
		
		/*
		* delete a process group
		*/
		void
		pgdelete(pgrp)
		register struct pgrp *pgrp;
		{
		register struct pgrp **pgp = &pgrphash[PIDHASH(pgrp->pg_id)];
		
		if (pgrp->pg_session->s_ttyp != NULL &&
		pgrp->pg_session->s_ttyp->t_pgrp == pgrp)
		pgrp->pg_session->s_ttyp->t_pgrp = NULL;
		for (; *pgp; pgp = &(*pgp)->pg_hforw) {
		if (*pgp == pgrp) {

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		311 *pgp = pgrp->pg_hforw; 312 break; 313 } 314 } 315 #ifdef DIAGNOSTIC 316 if (pgp == NULL) 317 panic("pgdelete: can't find pgrp on hash chain"); 318 #endif 319 if (--pgrp->pg_session->s_count == 0) 320 FREE(pgrp->pg_session, M_SESSION); 321 FREE(pgrp, M_PGRP); 322 } FreeBSD 2.0.5 defines FREE() as used in lines 320-21 of kern_proc.c in malloc.h, as shown below. Depending on whether KMEMSTATS or DIAGNOSTIC is defined, FREE() is either set to free() in line 288 or defined as in lines 304-20. 283 /* 284 * Macro versions for the usual cases of malloc/free 285 */ 286 #if defined(KMEMSTATS) || defined(DIAGNOSTIC) 287 #define MALLOC(space, cast, size, type, flags) \\ 288 (space) = (cast)malloc((u_long)(size), type, flags) 289 #define FREE(addr, type) free((caddr_t)(addr), type) 290 291 #else /* do not collect statistics */ 292 #define MALLOC(space, cast, size, type, flags) { \\ 293 register struct kmembuckets *kbp = &bucket[BUCKETINDX(size)]; \\ 294 long s = splimp(); \\ 295 if (kbp->kb_next == NULL) { \\

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

```
                (space) = (cast)malloc((u_long)(size), type, flags); \
        } else { \
                (space) = (cast)kbp->kb_next; \
                kbp->kb_next = *(caddr_t *)(space); \
        } \
        splx(s); \
}

#define FREE(addr, type) { \
        register struct kmembuckets *kbp; \
        register struct kmemusage *kup = btokup(addr); \
        long s = splimp(); \
        if (1 << kup->ku_indx > MAXALLOCSAVE) { \
                free((caddr_t)(addr), type); \
        } else { \
                kbp = &bucket[kup->ku_indx]; \
                if (kbp->kb_next == NULL) \
                        kbp->kb_next = (caddr_t)(addr); \
                else \
                        *(caddr_t *)(kbp->kb_last) = (caddr_t)(addr); \
                *(caddr_t *)(addr) = NULL; \
                kbp->kb_last = (caddr_t)(addr); \
        } \
        splx(s); \
}
#endif /* do not collect statistics */
```

The free() function, as defined in kern_malloc.c, is as follows.

```
void
```

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

```
free(addr, type)
	void *addr;
	int type;
{
	register struct kmembuckets *kbp;
	register struct kmemusage *kup;
	register struct freelist *freep;
	long size;
	int s;
#ifdef DIAGNOSTIC
	caddr_t cp;
	long *end, *lp, alloc, copysize;
#endif
#ifdef KMEMSTATS
	register struct kmemstats *ksp = &kmemstats[type];
#endif

#ifdef DIAGNOSTIC
	if ((char *)addr < kmembase || (char *)addr >= kmemlimit) {
	panic("free: address 0x%x out of range", addr);
	}
	if ((u_long)type > M_LAST) {
		panic("free: type %d out of range", type);
	}
#endif
	kup = btokup(addr);
	size = 1 << kup->ku_indx;
	kbp = &bucket[kup->ku_indx];
	s = splhigh();
#ifdef DIAGNOSTIC
```

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		/*
		* Check for returns of data that do not point to the
		* beginning of the allocation.
		*/
		if (size > NBPG * CLSIZE)
		alloc = addrmask[BUCKETINDX(NBPG * CLSIZE)];
		else
		alloc = addrmask[kup->ku_indx];
		if (((u_long)addr & alloc) != 0)
		panic("free: unaligned addr 0x%x, size %d, type %s, mask %d",
		addr, size, memname[type], alloc);
		#endif /* DIAGNOSTIC */
		if (size > MAXALLOCSAVE) {
		kmem_free(kmem_map, (vm_offset_t)addr, ctob(kup->ku_pagecnt));
		#ifdef KMEMSTATS
		size = kup->ku_pagecnt << PGSHIFT;
		ksp->ks_memuse -= size;
		kup->ku_indx = 0;
		kup->ku_pagecnt = 0;
		if (ksp->ks_memuse + size >= ksp->ks_limit &&
		ksp->ks_memuse < ksp->ks_limit)
		wakeup((caddr_t)ksp);
		ksp->ks_inuse--;
		kbp->kb_total -= 1;
		#endif
		splx(s);
		return;
		}

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		To the extent that FreeBSD does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in FreeBSD with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since FreeBSD utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of FreeBSD with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining FreeBSD with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		See also, gcache.c at lines 246-304, defining cainsert().
		"*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4.
		See also, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex().
		"*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4.
		See also, gcache.c at lines 355-376, defining caremove().
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	FreeBSD includes code that meets the "dynamically determining maximum number for the record search means to remove in the accessed linked list of records" claim limitation.
		
		For example, in lines 248-53 of kern_proc.c this first piece of code, the *if* and *for* statements dynamically determine whether the maximum number to remove is 0 or 1. If the *if* statement evaluates TRUE, then the maximum number to remove 1. If the *if* statement is FALSE and the *for* loop is not reached the last record, then the maximum number to remove is 1. If the *for* loop has reached the last record, and the *if* is FALSE, then it's 0.
		
		245 /*
		246 * unlink p from old process group
		247 */
		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		250 *pp = p->p_pgrpnxt; 251 break; 252 } 253 } Another example of the “dynamically determining” limitation can be found at lines 261-62 of kern_proc.c. The *if* statement dynamically determines the maximum number to remove. If the *if* statement evaluates TRUE, then the maximum number to remove is 1; if the *if* statement evaluates FALSE, then the maximum number to remove is 0. 258 /* 259 * delete old if empty 260 */ 261 if (p->p_pgrp->pg_mem == 0) 262 pgdelete(p->p_pgrp); Further, FreeBSD 2.0.5 combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks,

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD 2.0.5**[1]
		each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		$$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both FreeBSD 2.05 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as FreeBSD 2.05. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with FreeBSD 2.05 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with FreeBSD 2.05 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in FreeBSD 2.05 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both FreeBSD 2.05 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		result of combining FreeBSD 2.05 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine FreeBSD 2.05 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in FreeBSD 2.05 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining FreeBSD 2.05 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine FreeBSD 2.05 with Thatte. Alternatively, it would also be obvious to combine FreeBSD 2.05 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-7

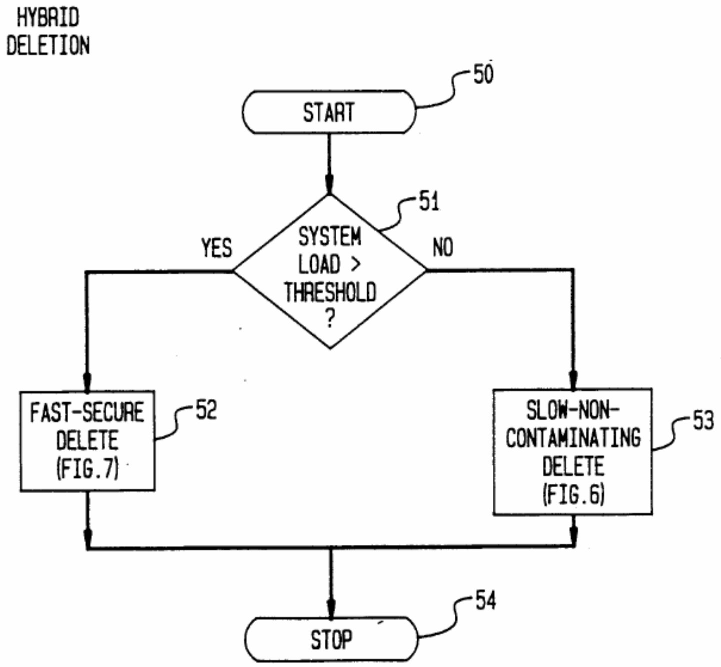
Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both FreeBSD 2.05 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in FreeBSD 2.05. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with FreeBSD 2.05 would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with FreeBSD 2.05 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine FreeBSD 2.05 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both FreeBSD 2.05 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as FreeBSD 2.05. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with FreeBSD 2.05 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with FreeBSD 2.05 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in FreeBSD 2.05 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in FreeBSD 2.05 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in FreeBSD 2.05 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by FreeBSD 2.0.5 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with FreeBSD 2.0.5. For example, both Linux 2.0.1 and FreeBSD 2.0.5 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, FreeBSD 2.0.5 discloses a "method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring" and a "method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring," as claimed. For example, in kern_proc.c and proc.h, FreeBSD 2.0.5 discloses a hash table of linked lists of automatically expiring data. See, e.g., struct pgrp defined at lines 61-70 of proc.h and struct proc defined at lines 72-172 of proc.h. Excerpts are below: 61 /* 62 * One structure allocated per process group. 63 */ 64 struct pgrp { 65 struct pgrp *pg_hforw; /* Forward link in hash bucket. */

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		struct proc *pg_mem; /* Pointer to pgrp members. */
		struct session *pg_session; /* Pointer to session. */
		pid_t pg_id; /* Pgrp id. */
		int pg_jobc; /* # procs qualifying pgrp for job control */
		};
		
		/*
		* Description of a process.
		*
		* This structure contains the information needed to manage a thread of
		* control, known in UN*X as a process; it has references to substructures
		* containing descriptions of things that the process uses, but may share
		* with related processes. The process structure and the substructures
		* are always addressible except for those marked "(PROC ONLY)" below,
		* which might be addressible only on a processor on which the process
		* is running.
		*/
		struct proc {
		struct proc *p_forw; /* Doubly-linked run/sleep queue. */
		struct proc *p_back;
		struct proc *p_next; /* Linked list of active procs */
		struct proc **p_prev; /* and zombies. */
		
		FreeBSD discloses a hashing means in connection with a linked list using an external chaining technique to store records with the same hash address. For example, the pidhash[] and pgrphash[] structures defined in param.c meet the “hashing means” limitation.
		
		struct proc *pidhash[PIDHSZ];

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		207 struct pgrp *pgrphash[PIDHSZ]; As shown in lines 61-70 and 72-172 of proc.h (portions of which are excerpted above), FreeBSD defines the pgrp structure as including a forward link in the hash bucket as well as a pointer to a linked list of proc structures. This is an example of how FreeBSD meets the “linked list” and “external chaining” limitations. Examples of how FreeBSD uses the hashing technique with external chaining can be found in the enterpgrp() function in kern_proc.c, such as the following: 230 pgrp->pg_hforw = pgrphash[n = PIDHASH(pgid)]; 231 pgrphash[n] = pgrp; The function that calls enterpgrp() passes in a proc structure, as shown in lines 175-79. 175 int 176 enterpgrp(p, pgid, mksess) 177 register struct proc *p; 178 pid_t pgid; 179 int mksess; Code within the enterpgrp() structure unlinks the proc from its old process group, as shown below in lines 248-53 of kern_proc.c. Also, enterpgrp() calls pgdelete() if the process group is empty, as shown in lines 261-62. Depending on claim construction, these are two examples of automatic expiration. 245 /* 246 * unlink p from old process group 247 */

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {
		250 *pp = p->p_pgrpnxt;
		251 break;
		252 }
		253 }
		
		258 /*
		259 * delete old if empty
		260 */
		261 if (p->p_pgrp->pg_mem == 0)
		262 pgdelete(p->p_pgrp);

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	FreeBSD discloses “accessing the linked list of records” and “accessing a linked list of records having same hash address,” as claimed. For example, the following code from the enterpgrp() function in kern_proc.c is an example of accessing a linked list of records having the same hash address. The [n] index is an example of a search key. The enterpgrp() function is an example of a “record search means” as claimed.
		
		230 pgrp->pg_hforw = pgrphash[n = PIDHASH(pgid)];
		231 pgrphash[n] = pgrp;

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	FreeBSD includes the step of “identifying at least some of the automatically expired ones of the records,” as claimed. For example, code from enterpgrp() in kern_proc.c discloses these limitations. One such example is the *if* statement at line 249 which identifies an automatically-expired record. Another example is the *if* statement at line 261 which identifies an empty record, which is an automatically-expired record. 245 /*

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		246 * unlink p from old process group
		247 */
		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {
		250 *pp = p->p_pgrpnxt;
		251 break;
		252 }
		253 }
		
		258 /*
		259 * delete old if empty
		260 */
		261 if (p->p_pgrp->pg_mem == 0)
		262 pgdelete(p->p_pgrp);

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	FreeBSD includes the step of "removing at least some of the automatically expired records from the linked list when the linked list is accessed," as claimed. For example, code from enterpgrp() in kern_proc.c discloses these limitations. One such example is the call to pgdelete() at line 262. The operation of pgdelete() is discussed in more detail herein at the discussion of elements 1d and 5d, herein.
		
		Depending on claim construction, the code at line 250 also meets the "removing" limitation.
		
		245 /*
		246 * unlink p from old process group
		247 */
		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {
		250 *pp = p->p_pgrpnxt;

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]

		251 break;
		252 }
		253 }
		
		258 /*
		259 * delete old if empty
		260 */
		261 if (p->p_pgrp->pg_mem == 0)
		262 pgdelete(p->p_pgrp);

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	FreeBSD includes the step of "inserting, retrieving or deleting one of the records from the system following the step of removing," as claimed. For example, the code at lines 266-68 of kern_proc.c inserts records into the system, immediately following the call to pgdelete() at line 262, which is an example of FreeBSD code that meets the "deleting" limitation.
		
		263 /*
		264 * link into new one
		265 */
		266 p->p_pgrp = pgrp;
		267 p->p_pgrpnxt = pgrp->pg_mem;
		268 pgrp->pg_mem = p;
		269 return (0);

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the	FreeBSD includes code that meets the "dynamically determining maximum number for the record search means to remove in the accessed linked list of records" claim limitation. For example, in lines 248-53 of kern_proc.c this first piece of code, the *if* and *for* statements dynamically determine whether the maximum number to remove is 0 or 1.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD 2.0.5[1]**

linked list is accessed.	linked list is accessed.	If the *if* statement evaluates TRUE, then the maximum number to remove 1. If the *if* statement is FALSE and the *for* loop is not reached the last record, then the maximum number to remove is 1. If the *for* loop has reached the last record, and the *if* is FALSE, then it's 0.
		
		245 /*
		246 * unlink p from old process group
		247 */
		248 for (pp = &p->p_pgrp->pg_mem; *pp; pp = &(*pp)->p_pgrpnxt) {
		249 if (*pp == p) {
		250 *pp = p->p_pgrpnxt;
		251 break;
		252 }
		253 }
		
		Another example of the "dynamically determining" limitation can be found at lines 261-62 of kern_proc.c. The *if* statement dynamically determines the maximum number to remove. If the *if* statement evaluates TRUE, then the maximum number to remove is 1; if the *if* statement evaluates FALSE, then the maximum number to remove is 0.
		
		258 /*
		259 * delete old if empty
		260 */
		261 if (p->p_pgrp->pg_mem == 0)
		262 pgdelete(p->p_pgrp);
		
		Further, FreeBSD 2.0.5 combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		As both FreeBSD 2.05 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as FreeBSD 2.05. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with FreeBSD 2.05 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with FreeBSD 2.05 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in FreeBSD 2.05 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both FreeBSD 2.05 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining FreeBSD 2.05 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine FreeBSD 2.05 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in FreeBSD 2.05 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining FreeBSD 2.05 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine FreeBSD 2.05 with Thatte.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		Alternatively, it would also be obvious to combine FreeBSD 2.05 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46.

EXHIBIT D-7

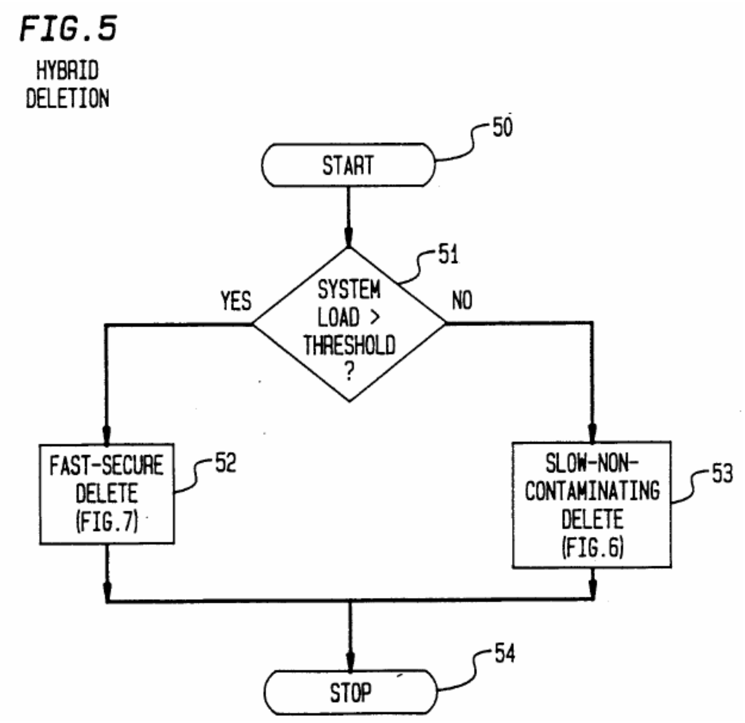
Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		This hybrid deletion is shown in Figure 5.  FIG.5 HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64,

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		**FreeBSD 2.0.5[1]**
		Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both FreeBSD 2.05 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in FreeBSD 2.05. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with FreeBSD 2.05 would be nothing more than the predictable use of prior art elements according

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with FreeBSD 2.05 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine FreeBSD 2.05 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both FreeBSD 2.05 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as FreeBSD 2.05. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with FreeBSD 2.05 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with FreeBSD 2.05 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in FreeBSD 2.05 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in FreeBSD 2.05 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in FreeBSD 2.05 can be burdensome on the system, adding to the system's load and slowing down the system's

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by FreeBSD 2.0.5 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with FreeBSD 2.0.5. For example, both Linux 2.0.1 and FreeBSD 2.0.5 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at

EXHIBIT D-7

Asserted Claims From U.S. Pat. No. 5,893,120		FreeBSD 2.0.5[1]
		line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, Linux 1.2.13 discloses an "information storage and retrieval system," as claimed. For example, in arp.c, discloses a hash table of linked lists of automatically expiring data. See, e.g., struct arp_table defined at lines 79-98.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	Linux 1.2.13 discloses "a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring" and "a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring." For example, the arp_table structure is a linked list, as shown in the code below. 72/* 73 * This structure defines the ARP mapping cache. As long as we make changes 74 * in this structure, we keep interrupts of. But normally we can copy the 75 * hardware address and the device pointer in a local variable and then make 76 * any "long calls" to send a packet out. 77 */ 78 79 struct arp_table 80 { 81 struct arp_table *next; /* Linked entry list */ 82 unsigned long last_used; /* For expiry */ 83 unsigned int flags; /* Control status */ 84 unsigned long ip; /* ip address of entry */ 85 unsigned long mask; /* netmask - used for generalised proxy arps (tridge) */

[1] Publicly available as of August 2, 1995; available at http://www.kernel.org/pub/linux/kernel/v1.2/linux-1.2.13.tar.gz.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		unsigned char ha[MAX_ADDR_LEN];/* Hardware address */ unsigned char hlen; /* Length of hardware address */ unsigned short htype; /* Type of hardware in use */ struct device *dev; /* Device the entry is tied to */ /* * The following entries are only used for unresolved hw addresses. */ struct timer_list timer; /* expire timer */ int retries; /* remaining retries */ struct sk_buff_head skb; /* list of queued packets */ The arp_table structure is also used in the context of hashing and external chaining. An example of this is shown in the following code from arp.c. /* * The size of the hash table. Must be a power of two. * Maybe we should remove hashing in the future for arp and concentrate * on Patrick Schaaf's Host-Cache-Lookup... */ #define ARP_TABLE_SIZE 16 /* The ugly +1 here is to cater for proxy entries. They are put in their own list for efficiency of lookup. If you don't want to find a proxy entry then don't look in the last entry, otherwise do */

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		170 #define FULL_ARP_TABLE_SIZE (ARP_TABLE_SIZE+1) 171 172 struct arp_table *arp_tables[FULL_ARP_TABLE_SIZE] = 173 { 174 NULL, 175 }; Also, functions such as arp_expire_request() deals with automatically-expiring records in the linked list. 367 /* 368 * This function is called, if an entry is not resolved in ARP_RES_TIME. 369 * Either resend a request, or give it up and free the entry. 370 */ 371 372 static void arp_expire_request (unsigned long arg)
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Linux 1.2.13 discloses "a record search means utilizing a search key to access the linked list" and "a record search means utilizing a search key to access a linked list of records having the same hash address." For example, the following code from arp_expire_request() in arp.c meets the "record search means" limitation. An example of using a search key to access a linked list of records having the same hash address is the hash value set at line 416 and used as at line 424. As discussed herein, the arp_tables [] structure is a hash table that uses linked lists to perform external chaining. 409 /* 410 * Arp request timed out. Delete entry and all waiting packets. 411 * If we give each entry a pointer to itself, we don't have to

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		412 * loop through everything again. Maybe hash is good enough, but 413 * I will look at it later. 414 */ 415 416 hash = HASH(entry->ip); 417 418 /* proxy entries shouldn't really time out so this is really 419 only here for completeness 420 */ 421 if (entry->flags & ATF_PUBL) 422 pentry = &arp_tables[PROXY_HASH]; 423 else 424 pentry = &arp_tables[hash];

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	Linux 1.2.13 discloses "the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed" and "the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed," as claimed.
		For example, in arp_expire_request() in arp.c, the while loop beginning at line 425 accesses the linked list as claimed. The if statement at line 427 identifies an expired record. Depending on claim construction, the "removing" limitation is met at, for example, line 429 and/or 432.
		425 while (*pentry != NULL)
		426 {
		427 if (*pentry == entry)
		428 {
		429 *pentry = entry->next; /* delete from linked list */

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.2.13[1]**

		430 del_timer(&entry->timer);
		431 restore_flags(flags);
		432 arp_release_entry(entry);
		433 return;
		434 }
		435 pentry = &(*pentry)->next;
		436 }

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Linux 1.2.13 discloses “means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list “ and “meals [*sic* “means”], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records,” as claimed. The “means, utilizing the record search means” limitation is met, for example, by a function that calls arp_expire_request(). As shown in the comments below, other code calls arp_expire_request(). 367/ * 368 * This function is called, if an entry is not resolved in ARP_RES_TIME. 369 * Either resend a request, or give it up and free the entry. 370 */ 371 372 static void arp_expire_request (unsigned long arg) For example, depending on claim construction, lines 429 and 432 in arp_expire_request() meet the “deleting” and “removing” limitations. An example of the “retrieving” step is line 435. Also, line 435 provides an example of “inserting.” These operations take place within a single while loop and “at the same time,” as claimed.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		while (*pentry != NULL)
		{
		    if (*pentry == entry)
		    {
		        pentry = entry->next; / delete from linked list */
		        del_timer(&entry->timer);
		        restore_flags(flags);
		        arp_release_entry(entry);
		        return;
		    }
		    pentry = &(*pentry)->next;
		}
		
		To the extent that Linux 1.2.13 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.2.13 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since Linux 1.2.13 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		ordinary skill in the art would be motivated to combine the linked list of Linux 1.2.13 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Linux 1.2.13 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		In cainsert():
		275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		
		In calookup():
		333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		
		In caremove():
		370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		666 if (caisold(pcb,pce)) {
		667 ++pcb->cb_tos;
		668 caunlink(pcb,ix);
		669 return(NULL_IX);
		670 } else {

2. The information storage and retrieval system according to claim 1 further	6. The information storage and retrieval system according to claim 5 further	Linux 1.2.13 includes code that meets the "dynamically determining maximum number for the record search means to remove in the accessed linked list of records" claim limitation.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	For example, lines each time the if statement at line 427 in arp_expire_request() is executed, it dynamically determines the maximum number of records to remove—it is either 1 or 0. If the if statement evaluates TRUE, then it's 1; if FALSE, then it's 0.
		425 while (*pentry != NULL)
		426 {
		427 if (*pentry == entry)
		428 {
		429 *pentry = entry->next; /* delete from linked list */
		430 del_timer(&entry->timer);
		431 restore_flags(flags);
		432 arp_release_entry(entry);
		433 return;
		434 }
		435 pentry = &(*pentry)->next;
		436 }
		Further, Linux 1.2.13 combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 1.2.13 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 1.2.13. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with Linux 1.2.13 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 1.2.13 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.2.13 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 1.2.13 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		result of combining Linux 1.2.13 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 1.2.13 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 1.2.13 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 1.2.13 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 1.2.13 with Thatte. Alternatively, it would also be obvious to combine Linux 1.2.13 with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-8

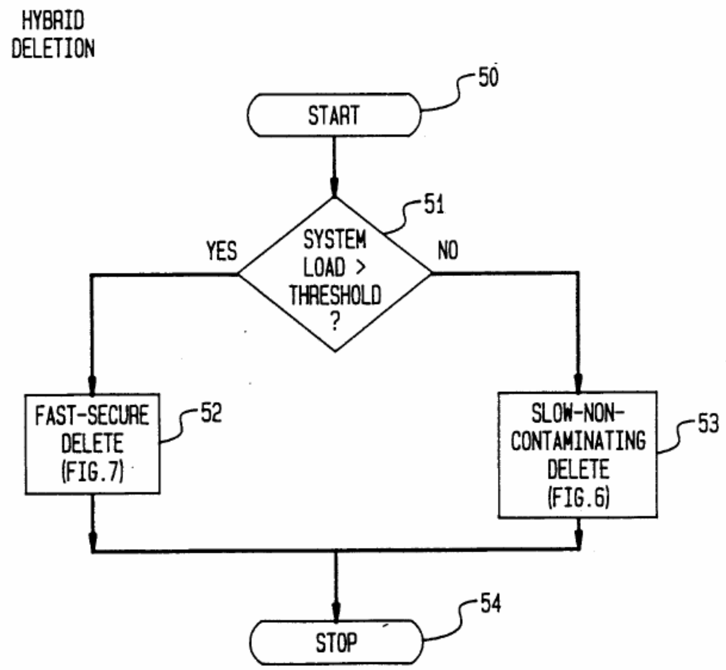
Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 1.2.13 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 1.2.13. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Linux 1.2.13 would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Linux 1.2.13 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 1.2.13 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 1.2.13 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 1.2.13. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 1.2.13 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 1.2.13 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 1.2.13 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Linux 1.2.13 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 1.2.13 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Linux 1.2.13 in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Linux 1.2.13. For example, both Linux 2.0.1 and Linux 1.2.13 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, Linux 1.2.13 discloses a "method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring" and a "method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring," as claimed. For exmaple, the arp_table structure defined in arp.c is an example of a linked list used to store and provide access to records, some of which are automatically expiring. /* * This structure defines the ARP mapping cache. As long as we make changes * in this structure, we keep interrupts of. But normally we can copy the * hardware address and the device pointer in a local variable and then make * any "long calls" to send a packet out. */

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		struct arp_table
		{
		    struct arp_table *next; /* Linked entry list */
		    unsigned long last_used; /* For expiry */
		    unsigned int flags; /* Control status */
		    unsigned long ip; /* ip address of entry */
		    unsigned long mask; /* netmask - used for generalised proxy arps (tridge) */
		    unsigned char ha[MAX_ADDR_LEN];/* Hardware address */
		    unsigned char hlen; /* Length of hardware address */
		    unsigned short htype; /* Type of hardware in use */
		    struct device *dev; /* Device the entry is tied to */
		
		    /*
		     * The following entries are only used for unresolved hw addresses.
		     */
		
		    struct timer_list timer; /* expire timer */
		    int retries; /* remaining retries */
		    struct sk_buff_head skb; /* list of queued packets */
		};
		
		The arp_table structure is also used in the context of hashing and external chaining. An example of this is shown in the following code from arp.c.
		
		/*
		 * The size of the hash table. Must be a power of two.
		 * Maybe we should remove hashing in the future for arp and concentrate
		 * on Patrick Schaaf's Host-Cache-Lookup...
		 */

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		162
		163 #define ARP_TABLE_SIZE 16
		164
		165 /* The ugly +1 here is to cater for proxy entries. They are put in their
		166 own list for efficiency of lookup. If you don't want to find a proxy
		167 entry then don't look in the last entry, otherwise do
		168 */
		169
		170 #define FULL_ARP_TABLE_SIZE (ARP_TABLE_SIZE+1)
		171
		172 struct arp_table *arp_tables[FULL_ARP_TABLE_SIZE] =
		173 {
		174 NULL,
		175 };
		
		Also, functions such as arp_check_expire() deal with automatically-expiring records in the linked list. For example, the comments at lines 187-91 discuss records that automatically expire.
		
		186/*
		187 * Check if there are too old entries and remove them. If the ATF_PERM
		188 * flag is set, they are always left in the arp cache (permanent entry).
		189 * Note: Only fully resolved entries, which don't have any packets in
		190 * the queue, can be deleted, since ARP_TIMEOUT is much greater than
		191 * ARP_MAX_TRIES*ARP_RES_TIME.
		192 */
		193
		194 static void arp_check_expire(unsigned long dummy)
		195 {
		196 int i;

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

```
unsigned long now = jiffies;
unsigned long flags;
save_flags(flags);
cli();

for (i = 0; i < FULL_ARP_TABLE_SIZE; i++)
{
        struct arp_table *entry;
        struct arp_table **pentry = &arp_tables[i];

        while ((entry = *pentry) != NULL)
        {
                if ((now - entry->last_used) > ARP_TIMEOUT
                        && !(entry->flags & ATF_PERM))
                {
                        *pentry = entry->next;  /* remove from list */
                        del_timer(&entry->timer);       /* Paranoia */
                        kfree_s(entry, sizeof(struct arp_table));
                }
                else
                        pentry = &entry->next;  /* go to next entry */
        }
}
restore_flags(flags);

/*
 *      Set the timer again.
 */

del_timer(&arp_timer);
```

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		227 arp_timer.expires = ARP_CHECK_INTERVAL; 228 add_timer(&arp_timer); 229 } Also, functions such as arp_expire_request() deal with automatically-expiring records in the linked list. 367 /* 368 * This function is called, if an entry is not resolved in ARP_RES_TIME. 369 * Either resend a request, or give it up and free the entry. 370 */ 371 372 static void arp_expire_request (unsigned long arg) art.

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Linux 1.2.13 discloses "accessing the linked list of records" and "accessing a linked list of records having same hash address," as claimed. For example, as discussed herein, the arp_tables[] structure is a hash table, and each linked list to which it points contains records having the same hash address. For example, the for loop at line 202 iterates through each hash value, and the while loop at line 207 iterates through the linked list associated with each has value. Thus, the while loop accesses the linked list of records having the same hash address, as claimed.
		
		202 for (i = 0; i < FULL_ARP_TABLE_SIZE; i++)
		203 {
		204 struct arp_table *entry;
		205 struct arp_table **pentry = &arp_tables[i];
		206

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

```
207            while ((entry = *pentry) != NULL)
208            {
209                 if ((now - entry->last_used) > ARP_TIMEOUT
210                      && !(entry->flags & ATF_PERM))
211                 {
212                      *pentry = entry->next;  /* remove from list */
213                      del_timer(&entry->timer);       /* Paranoia */
214                      kfree_s(entry, sizeof(struct arp_table));
215                 }
216                 else
217                      pentry = &entry->next;  /* go to next entry */
218            }
219       }
```

As another example, the following code from arp_expire_request() in arp.c meets the “accessing a linked list of records” and “accessing a linked list of records having same hash address” limitation. An example is the hash value set at line 416 and used as at line 424. As discussed herein, the arp_tables [] structure is a hash table that uses linked lists to perform external chaining.

```
409       /*
410        *      Arp request timed out. Delete entry and all waiting packets.
411        *      If we give each entry a pointer to itself, we don't have to
412        *      loop through everything again. Maybe hash is good enough, but
413        *      I will look at it later.
414        */
415
416       hash = HASH(entry->ip);
417
```

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		418 /* proxy entries shouldn't really time out so this is really 419 only here for completeness 420 */ 421 if (entry->flags & ATF_PUBL) 422 pentry = &arp_tables[PROXY_HASH]; 423 else 424 pentry = &arp_tables[hash];

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Linux 1.2.13 includes the step of "identifying at least some of the automatically expired ones of the records," as claimed. For example, the if statement at line 209-10 identifies expired records by comparing the last_used element to ARP_TIMEOUT. If last_used is greater than ARP_TIMEOUT, then that entry has expired.
		207 while ((entry = *pentry) != NULL)
		208 {
		209 if ((now - entry->last_used) > ARP_TIMEOUT
		210 && !(entry->flags & ATF_PERM))
		211 {
		212 *pentry = entry->next; /* remove from list */
		213 del_timer(&entry->timer); /* Paranoia */
		214 kfree_s(entry, sizeof(struct arp_table));
		215 }
		216 else
		217 pentry = &entry->next; /* go to next entry */
		218 }
		Another example is arp_expire_request() in arp.c, the while loop beginning at line 425 accesses the linked list as claimed. The if statement at line 427 identifies an expired record.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		425 while (*pentry != NULL)
		426 {
		427 if (*pentry == entry)
		428 {
		429 *pentry = entry->next; /* delete from linked list */
		430 del_timer(&entry->timer);
		431 restore_flags(flags);
		432 arp_release_entry(entry);
		433 return;
		434 }
		435 pentry = &(*pentry)->next;
		436 }

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Linux 1.2.13 includes the step of "removing at least some of the automatically expired records from the linked list when the linked list is accessed," as claimed. For example, line 212 moves the pointer so that the entry is no longer in the linked list. Also, line 214 calls the kfree_s() function (found in kmalloc.c), which removes the expired element by marking the memory that it occupied as free. Depending on claim construction, at least one of these actions is an example of "removing," as claimed. 209 if ((now - entry->last_used) > ARP_TIMEOUT 210 && !(entry->flags & ATF_PERM)) 211 { 212 *pentry = entry->next; /* remove from list */ 213 del_timer(&entry->timer); /* Paranoia */ 214 kfree_s(entry, sizeof(struct arp_table)); 215 } Another example is arp_expire_request() in arp.c, the while loop beginning at line 425 accesses the linked list as claimed. The if statement at line 427 identifies an

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		expired record. Depending on claim construction, the “removing” limitation is met at, for example, line 429 and/or 432.
		425 while (*pentry != NULL)
		426 {
		427 if (*pentry == entry)
		428 {
		429 *pentry = entry->next; /* delete from linked list */
		430 del_timer(&entry->timer);
		431 restore_flags(flags);
		432 arp_release_entry(entry);
		433 return;
		434 }
		435 pentry = &(*pentry)->next;
		436 }

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	Linux 1.2.13 includes the step of “inserting, retrieving or deleting one of the records from the system following the step of removing,” as claimed. For example, the function arp_check_expire() from arp.c is an example of code from Linux 1.2.13 that meets this element. Note that the code at line 212 moves the pointer so that the element is no longer in the linked list, then at line 214, kfree_s() is called which frees the memory associated with the element. After kfree_s() is called, control passes back to the *while* loop at line 207 and the next record is retrieved. If that record is NULL, control passes back to the *for* loop at line 202 and, unless the end of the hash table has been reached, the linked list associated with the next hash entry is retrieved. Thus, this is an example of inserting, retrieving, or deleing one of the records from the system following the step of removing.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		202 for (i = 0; i < FULL_ARP_TABLE_SIZE; i++)
		203 {
		204 struct arp_table *entry;
		205 struct arp_table **pentry = &arp_tables[i];
		206
		207 while ((entry = *pentry) != NULL)
		208 {
		209 if ((now - entry->last_used) > ARP_TIMEOUT
		210 && !(entry->flags & ATF_PERM))
		211 {
		212 *pentry = entry->next; /* remove from list */
		213 del_timer(&entry->timer); /* Paranoia */
		214 kfree_s(entry, sizeof(struct arp_table));
		215 }
		216 else
		217 pentry = &entry->next; /* go to next entry */
		218 }
		219 }
		
		Another example is arp_expire_request() in arp.c, the while loop beginning at line 425 accesses the linked list as claimed. Depending on claim construction, the “removing” limitation is met at, for example, line 429 and/or 432. Also, an example of the “deleting” following removing step is found in the call to arp_release_entry() and its operations.
		
		425 while (*pentry != NULL)
		426 {
		427 if (*pentry == entry)
		428 {

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		429 *pentry = entry->next; /* delete from linked list */
		430 del_timer(&entry->timer);
		431 restore_flags(flags);
		432 arp_release_entry(entry);
		433 return;
		434 }
		435 pentry = &(*pentry)->next;
		436 }
		
		The code for arp_release_entry() can be found at lines 236-254 of arp.c.
		
		232 /*
		233 * Release all linked skb's and the memory for this entry.
		234 */
		235
		236 static void arp_release_entry(struct arp_table *entry)
		237 {
		238 struct sk_buff *skb;
		239 unsigned long flags;
		240
		241 save_flags(flags);
		242 cli();
		243 /* Release the list of `skb' pointers. */
		244 while ((skb = skb_dequeue(&entry->skb)) != NULL)
		245 {
		246 skb_device_lock(skb);
		247 restore_flags(flags);
		248 dev_kfree_skb(skb, FREE_WRITE);
		249 }

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		250 restore_flags(flags);
		251 del_timer(&entry->timer);
		252 kfree_s(entry, sizeof(struct arp_table));
		253 return;
		254 }

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Linux 1.2.13 includes code that meets the “dynamically determining maximum number for the record search means to remove in the accessed linked list of records” claim limitation. For example, the following code from arp_check_expire() in arp.c is an example of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.
		This code meets this limitation in at least two ways. First, when the list is first accessed by the *while* loop beginning at line 207, the maximum number of records to remove is equal to the number of records in the list. But each time the *while* loop iterates and the *if* statement evaluates FALSE, that number decreases by one. Hence, it is dynamic.
		Second, each time the *if* statement at line 209-10 is called, the maximum number of records to remove is either 1 or 0, depending on whether the *if* statement evaluates to TRUE or FALSE. If the if statement evaluates TRUE, then the maximum number to remove is 1; if the if statement evaluates FALSE, the maximum number to remove is 0.
		207 while ((entry = *pentry) != NULL)
		208 {
		209 if ((now - entry->last_used) > ARP_TIMEOUT
		210 && !(entry->flags & ATF_PERM))
		211 {
		212 *pentry = entry->next; /* remove from list */

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]

		del_timer(&entry->timer); /* Paranoia */
		kfree_s(entry, sizeof(struct arp_table));
		}
		else
		pentry = &entry->next; /* go to next entry */
		}
		
		Further, Linux 1.2.13 combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.
		
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.
		
		For example, as summarized in Dirks,
		
		each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 1.2.13 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 1.2.13. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Linux 1.2.13 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 1.2.13 and

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.2.13 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 1.2.13 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 1.2.13 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 1.2.13 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 1.2.13 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 1.2.13 with the teachings of Thatte would solve this problem by

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 1.2.13 with Thatte. Alternatively, it would also be obvious to combine Linux 1.2.13 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-8

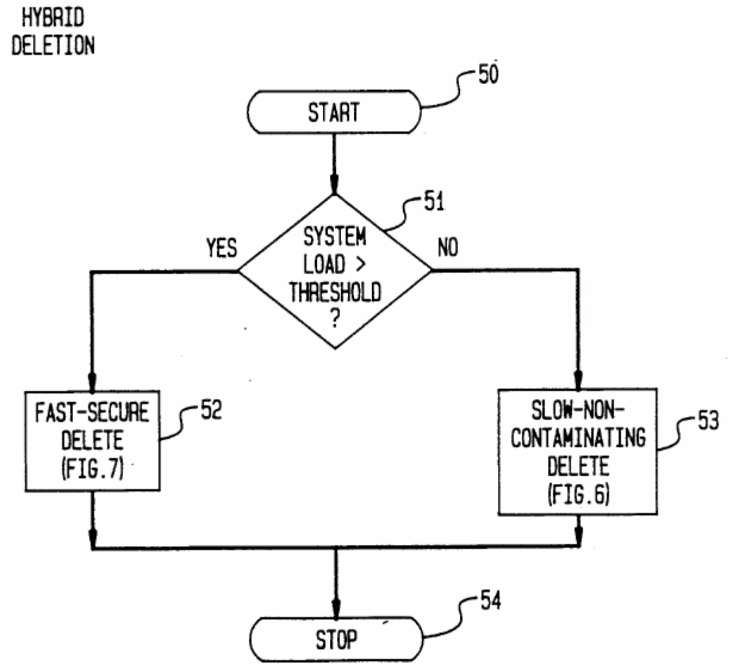
Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON- CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 1.2.13 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 1.2.13. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Linux 1.2.13 would be nothing more than the predictable use of prior art elements according to their established functions.

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.2.13**[1]
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Linux 1.2.13 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 1.2.13 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 1.2.13 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 1.2.13. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 1.2.13 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 1.2.13 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 1.2.13 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Linux 1.2.13 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 1.2.13 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Linux 1.2.13 in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Linux 1.2.13. For example, both Linux 2.0.1 and Linux 1.2.13 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically.

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts

EXHIBIT D-8

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.2.13[1]
		are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is limiting, Linux 1.3.51 discloses an “information storage and retrieval system,” as claimed.
		For example, route.c in Linux 1.3.51 includes fib_node and fib_zone structures that are used to provide hashing with external chaining using one or more linked lists. These structures are defined at lines 77-85, 104-112, and 114-117.
		73 /*
		74 * Forwarding Information Base definitions.
		75 */
		76
		77 struct fib_node
		78 {
		79 struct fib_node *fib_next;
		80 __u32 fib_dst;
		81 unsigned long fib_use;
		82 struct fib_info *fib_info;
		83 short fib_metric;
		84 unsigned char fib_tos;
		85 };
		86
		87 /*
		88 * This structure contains data shared by many of routes.
		89 */
		90
		91 struct fib_info
		92 {
		93 struct fib_info *fib_next;

[1] Publicly available as of December 27, 1995; available at http://www.kernel.org/pub/linux/kernel/v1.3/linux-1.3.51.tar.gz.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		struct fib_info *fib_prev;
		__u32 fib_gateway;
		struct device *fib_dev;
		int fib_refcnt;
		unsigned long fib_window;
		unsigned short fib_flags;
		unsigned short fib_mtu;
		unsigned short fib_irtt;
		};
		
		struct fib_zone
		{
		struct fib_zone *fz_next;
		struct fib_node **fz_hash_table;
		struct fib_node *fz_list;
		int fz_nent;
		int fz_logmask;
		__u32 fz_mask;
		};
		
		static struct fib_zone *fib_zones[33];
		static struct fib_zone *fib_zone_list;
		static struct fib_node *fib_loopback = NULL;
		static struct fib_info *fib_info_list;

[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store	Linux 1.3.51 discloses "a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring" and "a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring."

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

expiring,	the records with same hash address, at least some of the records automatically expiring,	For example, the fib_node structure defined at lines 77-85 of route.c includes a pointer to the next fib_node structure in a linked list (see line 79), which is used in the context of hashing with external chaining.
		The fib_zone structure can contain a pointer to a hash table (see line 107), which uses external chaining.
		An example of how these structures operate can be seen in the fib_add_1() function in route.c, which creates a hash table. The *fz* variable (e.g., at line 624) represents a *fib_zone* structure, which, as described above, includes a hash table, which is a pointer to a pointer to a *fig_node* element. The *fib_node* structure is a linked list, as shown by the fact that each element contains a pointer to the next element in the list (i.e., *fib_next*).
		620 /*
		621 * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
		622 */
		623
		624 if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table && logmask<32)
		625 {
		626 struct fib_node ** ht;
		627 #if RT_CACHE_DEBUG
		628 printk("fib_add_1: hashing for zone %d started\n", logmask);
		629 #endif
		630 ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*), GFP_KERNEL);
		631
		632 if (ht)
		633 {

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		634 memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
		635 cli();
		636 f1 = fz->fz_list;
		637 while (f1)
		638 {
		639 struct fib_node * next;
		640 unsigned hash = fz_hash_code(f1->fib_dst, logmask);
		641 next = f1->fib_next;
		642 f1->fib_next = ht[hash];
		643 ht[hash] = f1;
		644 f1 = next;
		645 }
		646 fz->fz_list = NULL;
		647 fz->fz_hash_table = ht;
		648 sti();
		649 }
		650 }
		651
		652 if (fz->fz_hash_table)
		653 fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
		654 else
		655 fp = &fz->fz_list;
		656
		657 /*
		658 * Scan list to find the first route with the same destination
		659 */
		660 while ((f1 = *fp) != NULL)
		661 {
		662 if (f1->fib_dst == dst)
		663 break;

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		664 fp = &f1->fib_next; 665 } 666 An example of code that meets the "automatically expiring" limitation can be found at lines 670-82. For example, a route with the same destination and less than or equal metric value has automatically expired, and, according to the comment at lines 675-77, is purged. 667 /* 668 * Find route with the same destination and less (or equal) metric. 669 */ 670 while ((f1 = *fp) != NULL && f1->fib_dst == dst) 671 { 672 if (f1->fib_metric >= metric) 673 break; 674 /* 675 * Record route with the same destination and gateway, 676 * but less metric. We'll delete it 677 * after instantiation of new route. 678 */ 679 if (f1->fib_info->fib_gateway == gw) 680 dup_fp = fp; 681 fp = &f1->fib_next; 682 }
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	Linux 1.3.51 discloses "a record search means utilizing a search key to access the linked list" and "a record search means utilizing a search key to access a linked list of records having the same hash address."

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		For example, the fib_add_1() function is an example of a record search means, as claimed. An example of how this uses a "search key" can be found at lines 652-53. This code uses a hash value to find the address of the first element of a linked list associated with a particular hash value. 652 if (fz->fz_hash_table) 653 fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	Linux 1.3.51 discloses "the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed" and "the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed," as claimed. For example, the code at lines 670-82 of route.c identifies a record in the linked list corresponding to a route with the same destination and less or equal metric. Thus, it accesses the linked list and identifies automatically expiring records. 667 /* 668 * Find route with the same destination and less (or equal) metric. 669 */ 670 while ((f1 = *fp) != NULL && f1->fib_dst == dst) 671 { 672 if (f1->fib_metric >= metric) 673 break; 674 /* 675 * Record route with the same destination and gateway, 676 * but less metric. We'll delete it 677 * after instantiation of new route. 678 */

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		679 if (f1->fib_info->fib_gateway == gw)
		680 dup_fp = fp;
		681 fp = &f1->fib_next;
		682 }
		
		Code within Linux 1.3.51 also performs the “removing” step when the list is accessed. An example of this can be found at lines 707-732 of route.c. For example, the *if* statement at line 718 identifies expired records, and if an expired record is found then line 721 moves the pointer so that record is no longer in the list. Depending on claim construction, this is the “removing” step. Also, the call to fib_free_node() at line 727 frees the memory used by the record. Depending on claim construction, this is the “removing” step. Both steps “identifying and removing” are performed within the *while* loop that starts at line 716 which accesses the list.
		
		707 /*
		708 * Delete route with the same destination and gateway.
		709 * Note that we should have at most one such route.
		710 */
		711 if (dup_fp)
		712 fp = dup_fp;
		713 else
		714 fp = &f->fib_next;
		715
		716 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		717 {
		718 if (f1->fib_info->fib_gateway == gw)
		719 {
		720 cli();
		721 *fp = f1->fib_next;
		722 if (fib_loopback == f1)

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		723 fib_loopback = NULL;
		724 sti();
		725 ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
		726 metric, f1->fib_info->fib_dev->name);
		727 fib_free_node(f1);
		728 fz->fz_nent--;
		729 break;
		730 }
		731 fp = &f1->fib_next;
		732 }
		
		The fib_free_node() function is found at lines 185-203 in route.c. As shown below, fib_free_node() moves pointers (lines 193-98) and calls kfree_s() to mark the memory as available (line 200).
		
		181 /*
		182 * Free FIB node.
		183 */
		184
		185 static void fib_free_node(struct fib_node * f)
		186 {
		187 struct fib_info * fi = f->fib_info;
		188 if (!--fi->fib_refcnt)
		189 {
		190 #if RT_CACHE_DEBUG >= 2
		191 printk("fib_free_node: fi %08x/%s is free\n", fi-
		>fib_gateway, fi->fib_dev->name);
		192 #endif
		193 if (fi->fib_next)
		194 fi->fib_next->fib_prev = fi->fib_prev;

EXHIBIT D-9

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>Linux 1.3.51[1]</th></tr>
<tr><td></td><td></td><td>
<pre>
			if (fi->fib_prev)
				fi->fib_prev->fib_next = fi->fib_next;
			if (fi == fib_info_list)
				fib_info_list = fi->fib_next;
		}
		kfree_s(f, sizeof(struct fib_node));
	}
</pre>
<p>The malloc.h file in Linux 1.3.51 defines kfree_s() as follows:</p>
<pre>
#define kfree_s(a,b) kfree(a)
</pre>
<p>The kmalloc.c file in Linux 1.3.51 defines kfree() as follows:</p>
<pre>
void kfree(void *ptr)
{
	int size;
	unsigned long flags;
	int order;
	register struct block_header *p;
	struct page_descriptor *page, **pg;

	if (!ptr)
		return;
	p = ((struct block_header *) ptr) - 1;
	page = PAGE_DESC(p);
	order = page->order;
	pg = &sizes[order].firstfree;
	if (p->bh_flags == MF_DMA) {
		p->bh_flags = MF_USED;
</pre>
</td></tr>
</table>

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		pg = &sizes[order].dmafree;
		}
		
		if ((order < 0) ||
		(order >= sizeof(sizes) / sizeof(sizes[0])) ||
		(((long) (page->next)) & ~PAGE_MASK) ||
		(p->bh_flags != MF_USED)) {
		printk("kfree of non-kmalloced memory: %p, next= %p, order=%d\n",
		p, page->next, page->order);
		return;
		}
		size = p->bh_length;
		p->bh_flags = MF_FREE; /* As of now this block is officially free */
		save_flags(flags);
		cli();
		p->bh_next = page->firstfree;
		page->firstfree = p;
		page->nfree++;
		
		if (page->nfree == 1) {
		/* Page went from full to one free block: put it on the freelist. */
		page->next = *pg;
		*pg = page;
		}
		/* If page is completely free, free it */
		if (page->nfree == NBLOCKS(order)) {
		for (;;) {
		struct page_descriptor *tmp = *pg;
		if (!tmp) {

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		321 printk("Ooops. page %p doesn't show on freelist.\n", page);
		322 break;
		323 }
		324 if (tmp == page) {
		325 *pg = page->next;
		326 break;
		327 }
		328 pg = &tmp->next;
		329 }
		330 sizes[order].npages--;
		331 free_pages((long) page, sizes[order].gfporder);
		332 }
		333 sizes[order].nfrees++;
		334 sizes[order].nbytesmalloced -= size;
		335 restore_flags(flags);
		336 }

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Linux 1.3.51 discloses “means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list “ and “meals [*sic* “means”], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records,” as claimed. For example, functions such as rt_add() call fib_add_1()—e.g., at line 1310 of route.c below. Such functions are examples of “means utilizing the record search means,” as claimed.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		static void rt_add(short flags, __u32 dst, __u32 mask,
		__u32 gw, struct device *dev, unsigned short mss,
		unsigned long window, unsigned short irtt, short metric)
		{
		while (ip_rt_lock)
		sleep_on(&rt_wait);
		ip_rt_fast_lock();
		fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
		ip_rt_unlock();
		wake_up(&rt_wait);
		}
		
		The fib_add_1() function is an example of code that meets the "inserting" limitations. For example, see the code below from route.c.
		
		/*
		* Insert new entry to the list.
		*/
		
		cli();
		f->fib_next = f1;
		*fp = f;
		if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
		fib_loopback = f;
		sti();
		fz->fz_nent++;
		ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric, fi->fib_dev->name);

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		The fib_add_l() function also is an example of code that meets the “accessing,” “retrieving” and “deleting” limitations at the same time as “removing.” The *while* loop beginning at line 716 is an example of code that meets the “retrieving” and “accessing” claim limitations. In order to iterate through the linked list, the *while* loop must retrieve and access each element of the list. Examples of code meeting the “removing” and “deleting” limitations can be found at lines 721 and 727. Line 721 moves the pointer to the next element. Depending on claim construction, this is the “removing” step. Alternatively, depending on claim construction, this is the “deleting” step, and the call to fib_free_node() is the “removing” step. The *if* statement at line 718 is an example of identifying expired records. Both of these steps (identifying and removing) take place when the list is accessed. For example, the *while* loops above iterate through the elements of the list in order to test each element to determine whether it should be removed. In order to identify the elements, the list must be accessed 707 /* 708 * Delete route with the same destination and gateway. 709 * Note that we should have at most one such route. 710 */ 711 if (dup_fp) 712 fp = dup_fp; 713 else 714 fp = &f->fib_next; 715 716 while ((f1 = *fp) != NULL && f1->fib_dst == dst) 717 {

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		if (f1->fib_info->fib_gateway == gw)
		{
		cli();
		*fp = f1->fib_next;
		if (fib_loopback == f1)
		fib_loopback = NULL;
		sti();
		ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
		metric, f1->fib_info->fib_dev->name);
		fib_free_node(f1);
		fz->fz_nent--;
		break;
		}
		fp = &f1->fib_next;
		}
		
		To the extent that Linux 1.3.51 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.3.51 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example,

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		since Linux 1.3.51 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of Linux 1.3.51 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Linux 1.3.51 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		“In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.” *See* Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage	6. The information storage	Linux 1.3.51 includes code that meets the “dynamically determining maximum

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.3.51[1]**

and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	number for the record search means to remove in the accessed linked list of records" claim limitation.
		For example, code in the fib_add_1() function in route.c determines the maximum number of records to remove. As the comment at lines 708-09 states, there should be no more than one route removed. Thus, the maximum number to remove is either 0 or 1. The *if* statement at line 718 dynamically determines whether that number is 0 or 1.
		707 /*
		708 * Delete route with the same destination and gateway.
		709 * Note that we should have at most one such route.
		710 */
		711 if (dup_fp)
		712 fp = dup_fp;
		713 else
		714 fp = &f->fib_next;
		715
		716 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		717 {
		718 if (f1->fib_info->fib_gateway == gw)
		719 {
		720 cli();
		721 *fp = f1->fib_next;
		722 if (fib_loopback == f1)
		723 fib_loopback = NULL;
		724 sti();
		725 ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
		726 metric, f1->fib_info->fib_dev->name);
		727 fib_free_node(f1);
		728 fz->fz_nent--;
		729 break;

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		730 } 731 fp = &f1->fib_next; 732 } Further, Linux 1.3.51 combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 1.3.51 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 1.3.51. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Linux 1.3.51 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 1.3.51 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.`

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.3.51 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 1.3.51 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 1.3.51 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 1.3.51 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 1.3.51 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 1.3.51 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 1.3.51 with Thatte. Alternatively, it would also be obvious to combine Linux 1.3.51 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of

EXHIBIT D-9

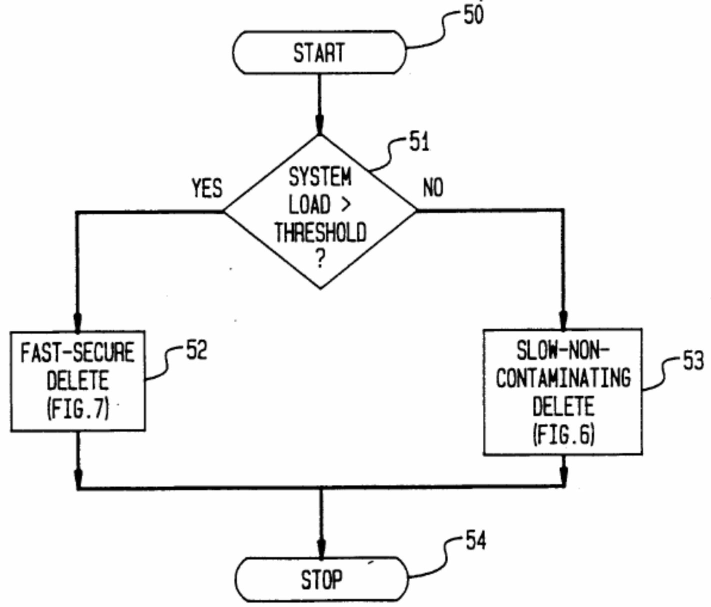
Asserted Claims From **U.S. Pat. No. 5,893,120**		**Linux 1.3.51**[1]
		> automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		Id. at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 1.3.51 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 1.3.51. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.3.51[1]**
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Linux 1.3.51 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Linux 1.3.51 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 1.3.51 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 1.3.51 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 1.3.51. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 1.3.51 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 1.3.51 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 1.3.51 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		disclosed in Linux 1.3.51 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 1.3.51 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Linux 1.3.51 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Linux 1.3.51. For example, both Linux 2.0.1 and Linux 1.3.51 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		function rt_cache_add removes an expired record, the function rt_cache_add decrements the variable rt_cache_size. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable rt_cache_size indicates the number of records in the hash table (i.e., ip_rt_hash_table). Because the function rt_cache_add automatically increments and decrements the variable rt_cache_size, the variable rt_cache_size is determined dynamically. Furthermore, LINUX 2.0.1 includes the function rt_garbage_collect_1. The function rt_garbage_collect_1 loops through each of the linked lists in the ip_rt_hash_table global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function rt_garbage_collect_1 accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than the maximum number of records that the function rt_cache_add can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method	To the extent the preamble is a limitation, Linux 1.3.51 discloses a "method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring" and a "method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring," as claimed. For example, route.c in Linux 1.3.51 includes fib_node and fib_zone structures that are used to provide hashing with external chaining using one or more linked lists.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

	comprising the steps of:	These structures are defined at lines 77-85, 104-112, and 114-117.
		
		73 /*
		74 * Forwarding Information Base definitions.
		75 */
		76
		77 struct fib_node
		78 {
		79 struct fib_node *fib_next;
		80 __u32 fib_dst;
		81 unsigned long fib_use;
		82 struct fib_info *fib_info;
		83 short fib_metric;
		84 unsigned char fib_tos;
		85 };
		86
		87 /*
		88 * This structure contains data shared by many of routes.
		89 */
		90
		91 struct fib_info
		92 {
		93 struct fib_info *fib_next;
		94 struct fib_info *fib_prev;
		95 __u32 fib_gateway;
		96 struct device *fib_dev;
		97 int fib_refcnt;
		98 unsigned long fib_window;
		99 unsigned short fib_flags;
		100 unsigned short fib_mtu;

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		101 unsigned short fib_irtt;
		102 };
		103
		104 struct fib_zone
		105 {
		106 struct fib_zone *fz_next;
		107 struct fib_node **fz_hash_table;
		108 struct fib_node *fz_list;
		109 int fz_nent;
		110 int fz_logmask;
		111 __u32 fz_mask;
		112 };
		113
		114 static struct fib_zone *fib_zones[33];
		115 static struct fib_zone *fib_zone_list;
		116 static struct fib_node *fib_loopback = NULL;
		117 static struct fib_info *fib_info_list;
		
		An example of code disclosing a hashing technique as claimed can be found in fib_del_1() in route.c, such as at lines 415 and 429. As shown in the discussion of the fib_node and fib_zone structures, this hashing technique uses external chaining, wherein a linked list is associated with elements having the same hash value.
		
		409 if (!mask)
		410 {
		411 for (fz=fib_zone_list; fz; fz = fz->fz_next)
		412 {
		413 int tmp;
		414 if (fz->fz_hash_table)
		415 fp = &fz->fz_hash_table[fz_hash_code(dst, fz-

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		>fz_logmask)];
		416 else
		417 fp = &fz->fz_list;
		418
		419 tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
		420 fz->fz_nent -= tmp;
		421 found += tmp;
		422 }
		423 }
		424 else
		425 {
		426 if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
		427 {
		428 if (fz->fz_hash_table)
		429 fp = &fz->fz_hash_table[fz_hash_code(dst, fz-
		>fz_logmask)];
		430 else
		431 fp = &fz->fz_list;
		432
		433 found = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
		434 fz->fz_nent -= found;
		435 }
		436 }
		
		Linux 1.3.51 also includes examples of automatically expiring records. For example, as shown in the comments at line 1286, rt_del() is only called by user processes. The condition that triggers the user process to call rt_del is an external condition. Note that in line 1297, rt_del() calls fib_del_1() to delete the expired records passed to it by

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		the user program that called rt_del(). 1286 * rt_{del|add|flush} called only from USER process. Waiting is OK. 1287 */ 1288 1289 static int rt_del(__u32 dst, __u32 mask, 1290 struct device * dev, __u32 gtw, short rt_flags, short metric) 1291 { 1292 int retval; 1293 1294 while (ip_rt_lock) 1295 sleep_on(&rt_wait); 1296 ip_rt_fast_lock(); 1297 retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric); 1298 ip_rt_unlock(); 1299 wake_up(&rt_wait); 1300 return retval; 1301 } Another example of how these structures operate can be seen in the fib_add_1() function in route.c, which creates a hash table. The *fz* variable (e.g., at line 624) represents a *fib_zone* structure, which, as described above, includes a hash table, which is a pointer to a pointer to a *fig_node* element. The *fib_node* structure is a linked list, as shown by the fact that each element contains a pointer to the next element in the list (i.e., *fib_next*). 620 /* 621 * If zone overgrows RTZ_HASHING_LIMIT, create hash table. 622 */

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table && logmask<32)
		{
		    struct fib_node ** ht;
		#if RT_CACHE_DEBUG
		    printk("fib_add_1: hashing for zone %d started\n", logmask);
		#endif
		    ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*), GFP_KERNEL);
		
		    if (ht)
		    {
		        memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
		        cli();
		        f1 = fz->fz_list;
		        while (f1)
		        {
		            struct fib_node * next;
		            unsigned hash = fz_hash_code(f1->fib_dst, logmask);
		            next = f1->fib_next;
		            f1->fib_next = ht[hash];
		            ht[hash] = f1;
		            f1 = next;
		        }
		        fz->fz_list = NULL;
		        fz->fz_hash_table = ht;
		        sti();
		    }
		}

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		652 if (fz->fz_hash_table)
		653 fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
		654 else
		655 fp = &fz->fz_list;
		656
		657 /*
		658 * Scan list to find the first route with the same destination
		659 */
		660 while ((f1 = *fp) != NULL)
		661 {
		662 if (f1->fib_dst == dst)
		663 break;
		664 fp = &f1->fib_next;
		665 }
		666
		
		An example of code that meets the "automatically expiring" limitation can be found at lines 670-82. For example, a route with the same destination and less than or equal metric value has automatically expired, and, according to the comment at lines 675-77, is purged.
		
		667 /*
		668 * Find route with the same destination and less (or equal) metric.
		669 */
		670 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		671 {
		672 if (f1->fib_metric >= metric)
		673 break;
		674 /*
		675 * Record route with the same destination and gateway,

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		676 * but less metric. We'll delete it 677 * after instantiation of new route. 678 */ 679 if (f1->fib_info->fib_gateway == gw) 680 dup_fp = fp; 681 fp = &f1->fib_next; 682 }

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	Linux 1.3.51 discloses "accessing the linked list of records" and "accessing a linked list of records having same hash address," as claimed. For example, line 415 and 429 each identify the location of a linked list of records pointed to by the hash table by calling the fz_hash_code() function.
		
		409 if (!mask)
		410 {
		411 for (fz=fib_zone_list; fz; fz = fz->fz_next)
		412 {
		413 int tmp;
		414 if (fz->fz_hash_table)
		415 fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
		416 else
		417 fp = &fz->fz_list;
		418
		419 tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
		420 fz->fz_nent -= tmp;
		421 found += tmp;
		422 }
		423 }
		424 else

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		425 {
		426 if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
		427 {
		428 if (fz->fz_hash_table)
		429 fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
		430 else
		431 fp = &fz->fz_list;
		432
		433 found = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
		434 fz->fz_nent -= found;
		435 }
		436 }
		
		As another example, the fib_add_1() function is an example of a accessing a linked list of records and accessing a linked list of records having the same hash address, as claimed. Lines 652-53 of route.c use a hash value to find the address of the first element of a linked list associated with a particular hash value.
		
		652 if (fz->fz_hash_table)
		653 fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	Linux 1.3.51 includes the step of "identifying at least some of the automatically expired ones of the records," as claimed. For example, fib_del_1() function calls fib_del_list() at lines 419 and 433 to perform the identifying, removing, retrieving, and deleting functions. 409 if (!mask) 410 { 411 for (fz=fib_zone_list; fz; fz = fz->fz_next)

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		{
		int tmp;
		if (fz->fz_hash_table)
		fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
		else
		fp = &fz->fz_list;
		
		tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
		fz->fz_nent -= tmp;
		found += tmp;
		}
		}
		else
		{
		if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
		{
		if (fz->fz_hash_table)
		fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
		else
		fp = &fz->fz_list;
		
		found = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
		fz->fz_nent -= found;
		}
		}

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		The fib_del_list() function is also found in route.c in Linux 1.3.51. The *while* loop beginning at line 373 in fib_del_list() iterates through the linked list and identifies the elements. The elements are “automatically expired” because, for example, the user program that called rt_del() determined that the elements needed to be removed. Then, rt_del() called fib_del_1(), which in turn called fib_del_list() to remove the automatically expired records.

```
367 static int fib_del_list(struct fib_node **fp, __u32 dst,
368       struct device * dev, __u32 gtw, short flags, short metric, __u32 mask)
369 {
370       struct fib_node *f;
371       int found=0;
372
373       while((f = *fp) != NULL)
374       {
375               struct fib_info * fi = f->fib_info;
376
377               /*
378               * Make sure the destination and netmask match.
379               * metric, gateway and device are also checked
380               * if they were specified.
381               */
382               if (f->fib_dst != dst ||
383               (gtw && fi->fib_gateway != gtw) ||
384               (metric >= 0 && f->fib_metric != metric) ||
385               (dev && fi->fib_dev != dev) )
386               {
387                       fp = &f->fib_next;
```

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		388 continue;
		389 }
		390 cli();
		391 *fp = f->fib_next;
		392 if (fib_loopback == f)
		393 fib_loopback = NULL;
		394 sti();
		395 ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi->fib_dev->name);
		396 fib_free_node(f);
		397 found++;
		398 }
		399 return found;
		400 }
		
		As another example, the code at lines 670-82 of route.c identifies a record in the linked list corresponding to a route with the same destination and less or equal metric. Thus, it accesses the linked list and identifies automatically expiring records.
		
		667 /*
		668 * Find route with the same destination and less (or equal) metric.
		669 */
		670 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		671 {
		672 if (f1->fib_metric >= metric)
		673 break;
		674 /*
		675 * Record route with the same destination and gateway,

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		676 * but less metric. We'll delete it
		677 * after instantiation of new route.
		678 */
		679 if (f1->fib_info->fib_gateway == gw)
		680 dup_fp = fp;
		681 fp = &f1->fib_next;
		682 }
		
		Code within Linux 1.3.51 also performs the “removing” step when the list is accessed. An example of this can be found at lines 707-732 of route.c. For example, the *if* statement at line 718 identifies expired records, and if an expired record is found then line 721 moves the pointer so that record is no longer in the list. Depending on claim construction, this is the “removing” step. Also, the call to fib_free_node() at line 727 frees the memory used by the record. Depending on claim construction, this is the “removing” step. Both steps “identifying and removing” are performed within the *while* loop that starts at line 716 which accesses the list.
		
		707 /*
		708 * Delete route with the same destination and gateway.
		709 * Note that we should have at most one such route.
		710 */
		711 if (dup_fp)
		712 fp = dup_fp;
		713 else
		714 fp = &f->fib_next;
		715
		716 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		717 {
		718 if (f1->fib_info->fib_gateway == gw)
		719 {

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		720 cli();
		721 *fp = f1->fib_next;
		722 if (fib_loopback == f1)
		723 fib_loopback = NULL;
		724 sti();
		725 ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
		726 metric, f1->fib_info->fib_dev->name);
		727 fib_free_node(f1);
		728 fz->fz_nent--;
		729 break;
		730 }
		731 fp = &f1->fib_next;
		732 }
		
		The fib_free_node() function is found at lines 185-203 in route.c. As shown below, fib_free_node() moves pointers (lines 193-98) and calls kfree_s() to mark the memory as available (line 200).
		
		181 /*
		182 * Free FIB node.
		183 */
		184
		185 static void fib_free_node(struct fib_node * f)
		186 {
		187 struct fib_info * fi = f->fib_info;
		188 if (!--fi->fib_refcnt)
		189 {
		190 #if RT_CACHE_DEBUG >= 2
		191 printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway, fi->fib_dev->name);

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		#endif
		if (fi->fib_next)
		fi->fib_next->fib_prev = fi->fib_prev;
		if (fi->fib_prev)
		fi->fib_prev->fib_next = fi->fib_next;
		if (fi == fib_info_list)
		fib_info_list = fi->fib_next;
		}
		kfree_s(f, sizeof(struct fib_node));
		}
		
		The malloc.h file in Linux 1.3.51 defines kfree_s() as follows:
		
		#define kfree_s(a,b) kfree(a)
		
		The kmalloc.c file in Linux 1.3.51 defines kfree() as follows:
		
		void kfree(void *ptr)
		{
		int size;
		unsigned long flags;
		int order;
		register struct block_header *p;
		struct page_descriptor *page, **pg;
		
		if (!ptr)
		return;
		p = ((struct block_header *) ptr) - 1;
		page = PAGE_DESC(p);
		order = page->order;

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

```
	pg = &sizes[order].firstfree;
	if (p->bh_flags == MF_DMA) {
		p->bh_flags = MF_USED;
		pg = &sizes[order].dmafree;
	}

	if ((order < 0) ||
		(order >= sizeof(sizes) / sizeof(sizes[0])) ||
		(((long) (page->next)) & ~PAGE_MASK) ||
		(p->bh_flags != MF_USED)) {
			printk("kfree of non-kmalloced memory: %p, next= %p, order=%d\n",
			p, page->next, page->order);
			return;
	}
	size = p->bh_length;
	p->bh_flags = MF_FREE; /* As of now this block is officially free */
	save_flags(flags);
	cli();
	p->bh_next = page->firstfree;
	page->firstfree = p;
	page->nfree++;

	if (page->nfree == 1) {
	/* Page went from full to one free block: put it on the freelist. */
		page->next = *pg;
		*pg = page;
	}
	/* If page is completely free, free it */
	if (page->nfree == NBLOCKS(order)) {
```

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		318 for (;;) {
		319 struct page_descriptor *tmp = *pg;
		320 if (!tmp) {
		321 printk("Ooops. page %p doesn't show on freelist.\n", page);
		322 break;
		323 }
		324 if (tmp == page) {
		325 *pg = page->next;
		326 break;
		327 }
		328 pg = &tmp->next;
		329 }
		330 sizes[order].npages--;
		331 free_pages((long) page, sizes[order].gfporder);
		332 }
		333 sizes[order].nfrees++;
		334 sizes[order].nbytesmalloced -= size;
		335 restore_flags(flags);
		336 }

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	Linux 1.3.51 includes the step of "removing at least some of the automatically expired records from the linked list when the linked list is accessed," as claimed. For example, the *while* loop beginning at line 373 access the linked list. The *if* statement at lines 382-89 causes the loop to move to the next element if there is no match. If there is a match, lines 391 and/or 396 meet the "removing" limitation, depending on claim construction. The "removing" takes place when the linked list is accessed. For example, the commands are executed within a while loop in the fib_del_list() function. 367 static int fib_del_list(struct fib_node **fp, __u32 dst,

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		struct device * dev, __u32 gtw, short flags, short metric, __u32 mask)
		{
		struct fib_node *f;
		int found=0;
		
		while((f = *fp) != NULL)
		{
		struct fib_info * fi = f->fib_info;
		
		/*
		* Make sure the destination and netmask match.
		* metric, gateway and device are also checked
		* if they were specified.
		*/
		if (f->fib_dst != dst ||
		(gtw && fi->fib_gateway != gtw) ||
		(metric >= 0 && f->fib_metric != metric) ||
		(dev && fi->fib_dev != dev))
		{
		fp = &f->fib_next;
		continue;
		}
		cli();
		*fp = f->fib_next;
		if (fib_loopback == f)
		fib_loopback = NULL;
		sti();
		ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric,

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		fi->fib_dev->name);
		fib_free_node(f);
		found++;
		}
		return found;
		}
		
		The fib_add_l() function also is another example of code that provides the step of removing at least some of the expired records when the list is accessed. For example, the *while* loop below iterates through the elements of the list in order to test each element to determine whether it should be removed. In order to identify the elements, the list must be accessed.
		
		/*
		* Delete route with the same destination and gateway.
		* Note that we should have at most one such route.
		*/
		if (dup_fp)
		fp = dup_fp;
		else
		fp = &f->fib_next;
		
		while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		{
		if (f1->fib_info->fib_gateway == gw)
		{
		cli();
		*fp = f1->fib_next;
		if (fib_loopback == f1)

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		723 fib_loopback = NULL;
		724 sti();
		725 ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
		726 metric, f1->fib_info->fib_dev->name);
		727 fib_free_node(f1);
		728 fz->fz_nent--;
		729 break;
		730 }
		731 fp = &f1->fib_next;
		732 }

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	Linux 1.3.51 includes the step of "inserting, retrieving or deleting one of the records from the system following the step of removing," as claimed. For example, depending on claim construction, the code at line 391 meets the "removing" limitation, and the code at 396 meets the "deleting" limitation. And because line 396 follows line 391, the deleting takes place "following the step of removing" as claimed. As another example, depending on claim construction, line 396 constitutes "removing," then when control passes back to the *while* loop at line 373, that code performs the "retrieving" step. Because this takes place after line 396 is executed, the "retrieving" step takes place "following the step of removing," as claimed. 367 static int fib_del_list(struct fib_node **fp, __u32 dst, 368 struct device * dev, __u32 gtw, short flags, short metric, __u32 mask) 369 { 370 struct fib_node *f; 371 int found=0; 372 373 while((f = *fp) != NULL)

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		See code below

```
    {
        struct fib_info * fi = f->fib_info;

        /*
        * Make sure the destination and netmask match.
        * metric, gateway and device are also checked
        * if they were specified.
        */
        if (f->fib_dst != dst ||
        (gtw && fi->fib_gateway != gtw) ||
        (metric >= 0 && f->fib_metric != metric) ||
        (dev && fi->fib_dev != dev) )
        {
            fp = &f->fib_next;
            continue;
        }
        cli();
        *fp = f->fib_next;
        if (fib_loopback == f)
            fib_loopback = NULL;
        sti();
        ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi->fib_dev->name);
        fib_free_node(f);
        found++;
    }
    return found;
}
```

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		The fib_free_node() function called by fib_del_list() is found at lines 185-203 in route.c. As shown below, fib_free_node() moves pointers (lines 193-98) and calls kfree_s() to mark the memory as available (line 200). For example, depending on claim construction, code in fib_free_node() meets the “removing” limitation and the code in kfree() meets the “deleting” limitation.

```
181     /*
182     * Free FIB node.
183     */
184
185     static void fib_free_node(struct fib_node * f)
186     {
187             struct fib_info * fi = f->fib_info;
188             if (!--fi->fib_refcnt)
189             {
190     #if RT_CACHE_DEBUG >= 2
191                     printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway, fi->fib_dev->name);
192     #endif
193                     if (fi->fib_next)
194                             fi->fib_next->fib_prev = fi->fib_prev;
195                     if (fi->fib_prev)
196                             fi->fib_prev->fib_next = fi->fib_next;
197                     if (fi == fib_info_list)
198                             fib_info_list = fi->fib_next;
199             }
200             kfree_s(f, sizeof(struct fib_node));
201     }
```

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		The malloc.h file in Linux 1.3.51 defines kfree_s() as follows:
		
		#define kfree_s(a,b) kfree(a)
		
		The kmalloc.c file in Linux 1.3.51 defines kfree() as follows:
		
		void kfree(void *ptr)
		{
		    int size;
		    unsigned long flags;
		    int order;
		    register struct block_header *p;
		    struct page_descriptor *page, **pg;
		
		    if (!ptr)
		        return;
		    p = ((struct block_header *) ptr) - 1;
		    page = PAGE_DESC(p);
		    order = page->order;
		    pg = &sizes[order].firstfree;
		    if (p->bh_flags == MF_DMA) {
		        p->bh_flags = MF_USED;
		        pg = &sizes[order].dmafree;
		    }
		
		    if ((order < 0) ||
		        (order >= sizeof(sizes) / sizeof(sizes[0])) ||
		        (((long) (page->next)) & ~PAGE_MASK) ||
		        (p->bh_flags != MF_USED)) {
		            printk("kfree of non-kmalloced memory: %p, next= %p,

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		order=%d\n",
		p, page->next, page->order);
		return;
		}
		size = p->bh_length;
		p->bh_flags = MF_FREE; /* As of now this block is officially free */
		save_flags(flags);
		cli();
		p->bh_next = page->firstfree;
		page->firstfree = p;
		page->nfree++;
		
		if (page->nfree == 1) {
		/* Page went from full to one free block: put it on the freelist. */
		page->next = *pg;
		*pg = page;
		}
		/* If page is completely free, free it */
		if (page->nfree == NBLOCKS(order)) {
		for (;;) {
		struct page_descriptor *tmp = *pg;
		if (!tmp) {
		printk("Ooops. page %p doesn't show on freelist.\n", page);
		break;
		}
		if (tmp == page) {
		*pg = page->next;
		break;
		}

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		328 pg = &tmp->next;
		329 }
		330 sizes[order].npages--;
		331 free_pages((long) page, sizes[order].gfporder);
		332 }
		333 sizes[order].nfrees++;
		334 sizes[order].nbytesmalloced -= size;
		335 restore_flags(flags);
		336 }
		
		Another example, depending on claim construction the step of removing is disclosed at lines 721 and/or 727. Also, an example of the step of deleting one of the records from the system can be found at line 733 with the call to rt_cache_flush().
		
		716 while ((f1 = *fp) != NULL && f1->fib_dst == dst)
		717 {
		718 if (f1->fib_info->fib_gateway == gw)
		719 {
		720 cli();
		721 *fp = f1->fib_next;
		722 if (fib_loopback == f1)
		723 fib_loopback = NULL;
		724 sti();
		725 ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
		726 metric, f1->fib_info->fib_dev->name);
		727 fib_free_node(f1);
		728 fz->fz_nent--;
		729 break;
		730 }
		731 fp = &f1->fib_next;

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		732 }
		733 rt_cache_flush();
		734 return;
		
		The rt_cache_flush() function is found at lines 1146-1185 of route.c:
		
		1146 static void rt_cache_flush(void)
		1147 {
		1148 int i;
		1149 struct rtable * rth, * next;
		1150
		1151 for (i=0; i<RT_HASH_DIVISOR; i++)
		1152 {
		1153 int nr=0;
		1154
		1155 cli();
		1156 if (!(rth = ip_rt_hash_table[i]))
		1157 {
		1158 sti();
		1159 continue;
		1160 }
		1161
		1162 ip_rt_hash_table[i] = NULL;
		1163 sti();
		1164
		1165 for (; rth; rth=next)
		1166 {
		1167 next = rth->rt_next;
		1168 rt_cache_size--;
		1169 nr++;

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		1170 rth->rt_next = NULL;
		1171 rt_free(rth);
		1172 }
		1173 #if RT_CACHE_DEBUG >= 2
		1174 if (nr > 0)
		1175 printk("rt_cache_flush: %d@%02x\n", nr, i);
		1176 #endif
		1177 }
		1178 #if RT_CACHE_DEBUG >= 1
		1179 if (rt_cache_size)
		1180 {
		1181 printk("rt_cache_flush: bug rt_cache_size=%d\n", rt_cache_size);
		1182 rt_cache_size = 0;
		1183 }
		1184 #endif
		1185 }

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Linux 1.3.51 includes code that meets the "dynamically determining maximum number for the record search means to remove in the accessed linked list of records" claim limitation. For example, the *while* loop beginning at line 373 of route.c iterates through the linked list passed in as **fp at line 367. Thus, when the fib_del_list() function is called, the maximum number of expired records to remove is the length of the **fp linked list. This number is dynamic because, if the *if* statement beginning at line 382 evaluates TRUE for an element, the maximum number to delete decreases by one. 367 static int fib_del_list(struct fib_node **fp, __u32 dst, 368 struct device * dev, __u32 gtw, short flags, short metric, __u32 mask)

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		{
		struct fib_node *f;
		int found=0;
		
		while((f = *fp) != NULL)
		{
		struct fib_info * fi = f->fib_info;
		
		/*
		* Make sure the destination and netmask match.
		* metric, gateway and device are also checked
		* if they were specified.
		*/
		if (f->fib_dst != dst ||
		(gtw && fi->fib_gateway != gtw) ||
		(metric >= 0 && f->fib_metric != metric) ||
		(dev && fi->fib_dev != dev))
		{
		fp = &f->fib_next;
		continue;
		}
		cli();
		*fp = f->fib_next;
		if (fib_loopback == f)
		fib_loopback = NULL;
		sti();
		ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi->fib_dev->name);

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]

		fib_free_node(f);
		found++;
		}
		return found;
		}
		
		Further, Linux 1.3.51 combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.
		
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.
		
		For example, as summarized in Dirks,
		
		each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 1.3.51 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 1.3.51. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Linux 1.3.51 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 1.3.51 and

EXHIBIT D-9

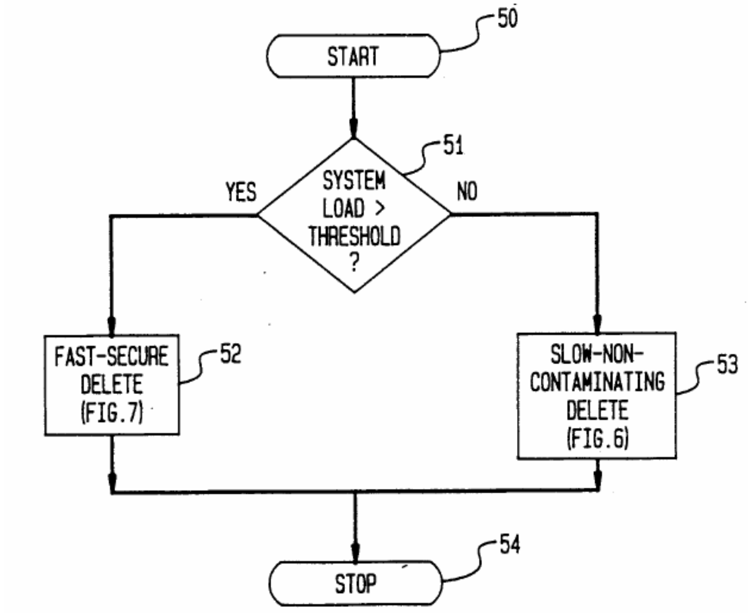
Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 1.3.51 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 1.3.51 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 1.3.51 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 1.3.51 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 1.3.51 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 1.3.51 with the teachings of Thatte would solve this problem by

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 1.3.51 with Thatte. Alternatively, it would also be obvious to combine Linux 1.3.51 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 1.3.51 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 1.3.51. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with Linux 1.3.51 would be nothing more than the predictable use of prior art elements according to their established functions.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with Linux 1.3.51 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 1.3.51 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 1.3.51 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 1.3.51. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with Linux 1.3.51 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 1.3.51 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 1.3.51 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Linux 1.3.51 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 1.3.51 can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		**Linux 1.3.51[1]**
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by Linux 1.3.51 in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with Linux 1.3.51. For example, both Linux 2.0.1 and Linux 1.3.51 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically.

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts

EXHIBIT D-9

Asserted Claims From U.S. Pat. No. 5,893,120		Linux 1.3.51[1]
		are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, GCache discloses an information storage and retrieval system. For example, Comer discloses an information storage and retrieval system using hash tables of linked lists. *See, e.g.*, Comer at 2-11, Fig. 1. "This section describes our implementation of a generalized caching system." *See* Comer at 2. *See also*, gcache.c which implements the generalized caching mechanism as described in Comer.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	GCache discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. GCache also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Comer discloses using linked lists to store records, the linked lists chained to a hash table using an external chaining technique: "GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion." *See* Comer at 3. "GCache keeps all *cacheentry* structures for an active cache either on the free

EXHIBIT D-10

Asserted Claims From **U.S. Pat. No. 5,893,120**		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table" *See* Comer at 5. "GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain." *See* Comer at 6.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

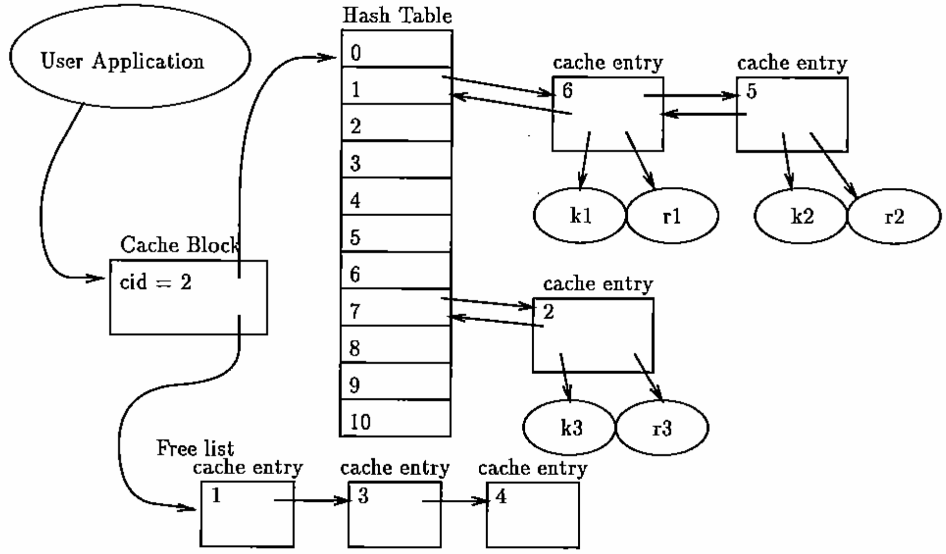
		
		 User Application
		Hash Table
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		cache entry
		6
		cache entry
		5
		k1
		r1
		k2
		r2
		Cache Block
		cid = 2
		cache entry
		2
		k3
		r3
		Free list
		cache entry
		1
		cache entry
		3
		cache entry
		4
		Figure 1: GCache Data Structures
		See also, gcache.c at lines 53-64, defining cacheentry as a linked list, as shown in the code below:
		`53 struct cacheentry {`
		`54 ce_status ce_status; /* INUSE or FREE */`
		`55 char *ce_keyptr; /* pointer to the key */`
		`56 tcelen ce_keylen; /* length of the key */`
		`57 char *ce_resptr; /* pointer to the result */`
		`58 tcelen ce_reslen; /* length of the result */`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		59 thval ce hash; /* value that was hashed in */ 60 ttstamp ce_tsinsert; /* timestamp - time inserted */ 61 ttstamp ce_tsaccess; /* timestamp - last access */ 62 tceix ce_prev; /* next entry on list */ 63 tceix ce_next; /* prev entry on list */ 64 }; Comer discloses storing records in a linked list, for example: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gache.c at lines 241-304, defining cainsert(). Comer discloses providing access to records stored in a linked list, for example: "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 307-347 and 637-678, defining calookup() and cagetindex(). Comer discloses at least some of the records automatically expiring, for example: "In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10. *See also*, gcache.c at lines 617-634, defining caisold() which determines if the record has expired: 617 /* 618 * === ===== 619 * caisold - return TRUE if the given entry is "too old" 620 * === ===== 621 */ 622 LOCAL int caisold(pcb,pce) 623 struct cacheblk *pcb; 624 struct cacheentry *pce; 625 { 626 unsigned now; 627 628 if (pcb->cb_maxlife == 0) 629 return(FALSE); 630 631 gettime(&now); 632 633 return ((now - pce->ce_tsaccess) > pcb->cb_maxlife); 634 }
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same	GCache discloses a record search means utilizing a search key to access the linked list. GCache also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

	hash address,	For example, Comer discloses utilizing a search key to access a linked list of records having the same hash address. In the quoted text below, the use of hash value to determine a hash table with an attached linked list of records having the same hash address is an example of “utilizing a search key to access a linked list.”
		
		“GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion.” *See* Comer at 3.
		
		“GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table” *See* Comer at 5.
		
		“GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain.” *See* Comer at 6.
		
		See also, gcache.c at lines 637-677, defining cagetindex(), which contains code constituting a “record search means” that uses a hash value to access and traverse a linked list of records having the same hash address:
		
		`637 /*`
		`638 * ====================================================================`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

		* cagetindex - return the index of a matching entry, or SYSERR
		* N.B. assumes MUTEX is already held
		*
		==
		*/
		LOCAL tceix cagetindex(pcb,pkey,keylen,hash)
		struct cacheblk *pcb;
		char *pkey;
		tcelen keylen;
		thval hash;
		{
		struct cacheentry *pce;
		tceix ix;
		tceix nextix;
		
		++pcb->cb_lookups;
		
		ix = pcb->cb_hash[HASHTOIX(hash,pcb)].he_ix;
		
		while (ix != NULL_IX) {
		pce = &pcb->cb_cache[ix];
		nextix = pce->ce_next;
		
		if ((pce->ce_hash == hash) &&
		(pce->ce_keylen == keylen) &&
		(blkequ(pkey,pce->ce_keyptr,keylen))) {
		/* this is a match */
		++pcb->cb_hits;
		if (caisold(pcb,pce)) {
		++pcb->cb_tos;
		caunlink(pcb,ix);
		return(NULL_IX);
		} else {
		return(ix);
		}

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		`673 }`
		`674 ix = nextix;`
		`675 }`
		`676`
		`677 return(NULL_IX);`
		`678 }`

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	GCache discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. GCache also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, Comer discloses that "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. At line 333 of gcache.c, calookup() calls the function cagetindex(): `333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {` In addition Comer discloses a means for identifying and removing expired records from the linked list of records: "In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10. At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

		linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list:
		`if (caisold(pcb,pce)) {`
		`++pcb->cb_tos;`
		`caunlink(pcb,ix);`
		`return(NULL_IX);`
		`} else {`
		To the extent that GCache does not disclose this limitation, GCache in combination with Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”) discloses record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed and also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed.
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCache with the means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed disclosed by NRL IPv6. *See, e.g.*, key.c at lines 1396-1563. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the system including a

EXHIBIT D-10

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, <i>GCache: A Generalized Caching Mechanism</i>, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)</th></tr>
<tr><td></td><td></td><td>hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in NRL IPv6 is clearly shown in the chart of NRL IPv6, which is hereby incorporated by reference in its entirety.

Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions.

For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
<pre>
188 struct key_acquirelist {
189 u_int8 type; /* secassoc type to acquire */
190 struct sockaddr_in6 target; /* destination address of
secassoc */
191 u_int32 count; /* number of acquire messages sent */
192 u_long expiretime; /* expiration time for acquire message
*/
193 struct key_acquirelist *next;
194 };
</pre>In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. <i>See</i>, <i>e.g.</i>, key.c at lines 1411, 1430-1459.

Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:</td></tr>
</table>

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		`192 u_long expiretime; /* expiration time for acquire message */`
		
		`Furthermore, key.c discloses` a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, as shown in the code below:
		
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`
		`1449 * This should really be done by a function like key_reaper()`
		`1450 * but until we code key_reaper(), this is a quick and dirty`
		`1451 * hack.`
		`1452 */`
		`1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n"));`
		`1454 prevap->next = ap->next;`
		`1455 KFree(ap);`
		`1456 ap = prevap;`
		`1457 }`
		
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least	GCache discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. GCache also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
records in the linked list.	some expired ones of the records in the accessed linked list of records.	For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here: In cainsert(): `275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL IX)`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		`{`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`
		
		To the extent that GCache does not entirely disclose this limitation, GCache in combination with Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6") discloses record search means including a means for identifying and removing

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		at least some of the expired ones of the records from the linked list when the linked list is accessed and also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed.
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCache with the means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed disclosed by NRL IPv6. *See, e.g.*, key.c at lines 1396-1563. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in NRL IPv6 is clearly shown in the chart of NRL IPv6, which is hereby incorporated by reference in its entirety.
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`188 struct key_acquirelist {`
		`189 u int8 type; /* secassoc type to acquire */`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)

		`190 struct sockaddr in6 target; /* destination address of secassoc */`
		`191 u_int32 count; /* number of acquire messages sent */`
		`192 u_long expiretime; /* expiration time for acquire message */`
		`193 struct key_acquirelist *next;`
		`194 };`
		
		In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. *See*, *e.g.*, key.c at lines 1411, 1430-1459.
		
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:
		
		`192 u_long expiretime; /* expiration time for acquire message */`
		
		Furthermore, key.c discloses a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, as shown in the code below：
		
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`
		`1449 * This should really be done by a function like key_reaper()`
		`1450 * but until we code key_reaper(), this is a quick and dirty`
		`1451 * hack.`
		`1452 */`
		`1453 DPRINTF(IDL MAJOR EVENT,("found an expired`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		`entry...deleting`
		`it!\n"));`
		`1454 prevap->next = ap->next;`
		`1455 KFree(ap);`
		`1456 ap = prevap;`
		`1457 }`
		
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	GCache discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.
		
		For example, Comer discloses dynamically determining whether to delete one record or zero records from the accessed linked list of records:
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink() and returns. If the matching record is not expired, the code returns the index of the matched record:
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb tos;`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")

		668 caunlink(pcb,ix);
		669 return(NULL_IX);
		670 } else {
		671 return(ix);
		672 }
		
		In a second example, Comer discloses dynamically determining whether to delete one record or zero records from an accessed list of records:
		
		"Insertion of a new entry into a full cache forces the deletion of the entry that was looked up the least recently." *See* Comer at 3.
		
		At line 275 of gcache.c, cainsert() calls cagetindex() to access a linked list of records and see if a matching entry already exists. If a matching entry does not exist, cainset() calls cagetfree() at line 281 to get a free entry. In cagetfree(), the following code dynamically determines whether to delete one record or zero records:
		
		719 /* if the free list is empty, delete the oldest entry */
		720 if (pcb->cb_freelist == NULL_IX) {
		721 cadeleteold(pcb);
		722 ++pcb->cb_fulls;
		723 }
		
		In addition, GCache combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		As both GCache and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as GCache. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with GCache nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with GCache and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCache with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example,

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both GCache and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining GCache with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine GCache with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in GCache can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining GCache with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120

EXHIBIT D-10

Asserted Claims From **U.S. Pat. No. 5,893,120**		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		patent provides motivations to combine GCache with Thatte. Alternatively, it would also be obvious to combine GCache with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT D-10

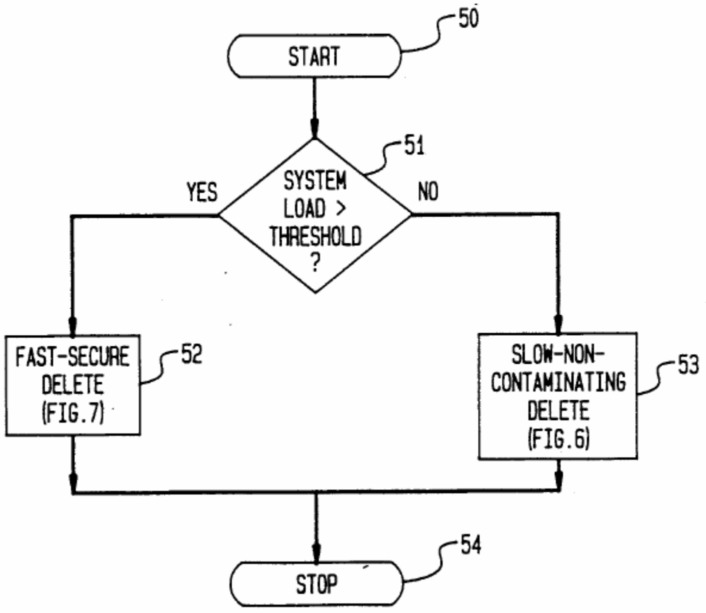
Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both GCache and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in GCache. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with GCache would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with GCache and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine GCache with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both GCache and the Opportunistic Garbage Collection Articles relate to

EXHIBIT D-10

Asserted Claims From **U.S. Pat. No. 5,893,120**		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**
		deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as GCache. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with GCache would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with GCache and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in GCache to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in GCache with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in GCache can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by GCache in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with GCache. For example, both Linux 2.0.1 and GCache describe systems and methods for performing

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110.

EXHIBIT D-10

Asserted Claims From **U.S. Pat. No. 5,893,120**		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, GCache discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. GCache also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, Comer discloses using linked lists to store records, the linked lists chained to a hash table using an external chaining technique: "GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		insertion and deletion." *See* Comer at 3. "GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table" *See* Comer at 5. "GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain." *See* Comer at 6.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

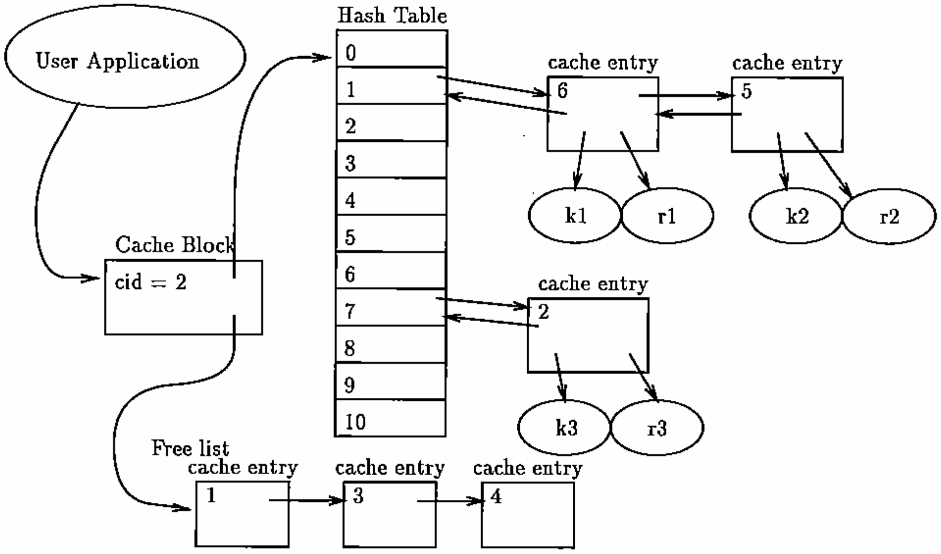
		 User Application
		Hash Table
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		cache entry
		6
		cache entry
		5
		k1
		r1
		k2
		r2
		Cache Block
		cid = 2
		cache entry
		2
		k3
		r3
		Free list
		cache entry
		1
		cache entry
		3
		cache entry
		4
		Figure 1: GCache Data Structures
		
		See also, gcache.c at lines 53-64, defining cacheentry as a linked list, as shown in the code below:
		
		`53 struct cacheentry {`
		`54 ce_status ce_status; /* INUSE or FREE */`
		`55 char *ce_keyptr; /* pointer to the key */`
		`56 tcelen ce_keylen; /* length of the key */`
		`57 char *ce_resptr; /* pointer to the result */`
		`58 tcelen ce_reslen; /* length of the result */`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

		`59 thval ce hash; /* value that was hashed in */`
		`60 ttstamp ce_tsinsert; /* timestamp - time inserted */`
		`61 ttstamp ce_tsaccess; /* timestamp - last access */`
		`62 tceix ce_prev; /* next entry on list */`
		`63 tceix ce_next; /* prev entry on list */`
		`64 };`
		
		Comer discloses storing records in a linked list, for example:
		
		“*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4.
		
		See also, gache.c at lines 241-304, defining cainsert().
		
		Comer discloses providing access to records stored in a linked list, for example:
		
		“*Calookup*() searches for a cached entry matching the key passed as an argument.” *See* Comer at 4.
		
		See also, gcache.c at lines 307-347 and 637-678, defining calookup() and cagetindex().
		
		Comer discloses at least some of the records automatically expiring, for example:
		
		“In a simpler and cleaner design chosen for GCache, each cached entry

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")

		contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		See also, gcache.c at lines 617-634, defining caisold() which determines if the record has expired:
		`617 /*`
		`618 *`
		`===============================================================`
		`=====`
		`619 * caisold - return TRUE if the given entry is "too old"`
		`620 *`
		`===============================================================`
		`=====`
		`621 */`
		`622 LOCAL int caisold(pcb,pce)`
		`623 struct cacheblk *pcb;`
		`624 struct cacheentry *pce;`
		`625 {`
		`626 unsigned now;`
		`627`
		`628 if (pcb->cb_maxlife == 0)`
		`629 return(FALSE);`
		`630`
		`631 gettime(&now);`
		`632`
		`633 return ((now - pce->ce_tsaccess) > pcb->cb_maxlife);`
		`634 }`

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	GCache discloses accessing a linked list of records. GCache also discloses accessing a linked list of records having same hash address.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		For example, Comer discloses accessing a linked list of records having the same hash address:
		"GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion." *See* Comer at 3.
		"GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table" *See* Comer at 5.
		"GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain." *See* Comer at 6.
		See also, gcache.c at lines 637-677, defining cagetindex(), which contains code constituting a "record search means" that uses a hash value to access and traverse a linked list of records having the same hash address:
		`637 /*`
		`638 *`
		`===============================================================`
		`=====`
		`639 * cagetindex - return the index of a matching entry, or`
		`SYSERR`
		`640 * N.B. assumes MUTEX is already held`
		`641 *`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")

		==
		=====
		*/
		LOCAL tceix cagetindex(pcb,pkey,keylen,hash)
		struct cacheblk *pcb;
		char *pkey;
		tcelen keylen;
		thval hash;
		{
		struct cacheentry *pce;
		tceix ix;
		tceix nextix;
		
		++pcb->cb_lookups;
		
		ix = pcb->cb_hash[HASHTOIX(hash,pcb)].he_ix;
		
		while (ix != NULL_IX) {
		pce = &pcb->cb_cache[ix];
		nextix = pce->ce_next;
		
		if ((pce->ce_hash == hash) &&
		(pce->ce_keylen == keylen) &&
		(blkequ(pkey,pce->ce_keyptr,keylen))) {
		/* this is a match */
		++pcb->cb_hits;
		if (caisold(pcb,pce)) {
		++pcb->cb_tos;
		caunlink(pcb,ix);
		return(NULL_IX);
		} else {
		return(ix);
		}
		}
		ix = nextix;
		}

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		`677 return(NULL_IX);`
		`678 }`

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	GCache discloses identifying at least some of the automatically expired ones of the records.
		
		For example, Comer discloses a means for identifying and removing expired records from the linked list of records:
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list:
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`
		
		To the extent that GCache does not disclose this limitation, GCache in combination with Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6") discloses identifying at least some of the automatically expired ones of the

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		records.
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the method disclosed in GCache with the techniques for identifying at least some of the automatically expired ones of the records disclosed by NRL IPv6. *See*, *e.g.*, key.c at lines 1396-1563. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a method that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the method utilizing a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in NRL IPv6 is clearly shown in the chart of NRL IPv6, which is hereby incorporated by reference in its entirety.
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`struct key_acquirelist {`
		`u_int8 type; /* secassoc type to acquire */`
		`struct sockaddr_in6 target; /* destination address of secassoc */`
		`u_int32 count; /* number of acquire messages sent */`
		`u_long expiretime; /* expiration time for acquire message`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

		`*/`
		`193 struct key_acquirelist *next;`
		`194 };`
		
		In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. *See, e.g.*, key.c at lines 1411, 1430-1459.
		
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:
		
		`192 u_long expiretime; /* expiration time for acquire message`
		`*/`
		
		`Furthermore, key.c discloses` a means for identifying at least some of the expired ones of the records, as shown in the code below：
		
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`
		`1449 * This should really be done by a function like`
		`key_reaper()`
		`1450 * but until we code key_reaper(), this is a quick and`
		`dirty`
		`1451 * hack.`
		`1452 */`
		`1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired`
		`entry...deleting`
		`it!\n"));`
		`1454 prevap->next = ap->next;`
		`1455 KFree(ap);`
		`1456 ap = prevap;`
		`1457 }`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	GCache discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed.
		For example:
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list:
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`
		To the extent that GCache does not disclose this limitation, GCache in combination with Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6") discloses removing at least some of the automatically expired records from the

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		linked list when the linked list is accessed.
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the method disclosed in GCache with the techniques for removing at least some of the automatically expired records from the linked list when the linked list is accessed disclosed by NRL IPv6. *See, e.g.*, key.c at lines 1396-1563. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a method that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the method utilizing a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in NRL IPv6 is clearly shown in the chart of NRL IPv6, which is hereby incorporated by reference in its entirety.
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`188 struct key_acquirelist {`
		`189 u_int8 type; /* secassoc type to acquire */`
		`190 struct sockaddr_in6 target; /* destination address of secassoc */`
		`191 u_int32 count; /* number of acquire messages sent */`
		`192 u_long expiretime; /* expiration time for acquire message`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		`*/`
		`193 struct key_acquirelist *next;`
		`194 };`
		
		In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. *See, e.g.*, key.c at lines 1411, 1430-1459.
		
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:
		
		`192 u_long expiretime; /* expiration time for acquire message`
		`*/`
		
		Furthermore, key.c discloses a means for identifying at least some of the expired ones of the records, as shown in the code below:
		
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`
		`1449 * This should really be done by a function like`
		`key_reaper()`
		`1450 * but until we code key_reaper(), this is a quick and`
		`dirty`
		`1451 * hack.`
		`1452 */`
		`1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired`
		`entry...deleting`
		`it!\n"));`
		`1454 prevap->next = ap->next;`
		`1455 KFree(ap);`
		`1456 ap = prevap;`
		`1457 }`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	GCache discloses inserting, retrieving or deleting one of the records from the system following the step of removing.
		For example, Comer discloses means for inserting records:
		"*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4.
		At line 275 of gcache.c, cainsert() utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below.
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and if so, removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")

		return(NULL_IX);
		} else {
		
		After the call to cagetindex() returns, through which the expired entry was removed, cainsert() proceeds to insert a new entry at the head of the list and populates the fields of the structure, as shown in the code below:
		
		if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		/* use the old one */
		caclear(pcb,ixnew);
		pce = &pcb->cb_cache[ixnew];
		} else {
		/* get a free cacheentry */
		ixnew = cagetfree(pcb);
		pce = &pcb->cb_cache[ixnew];
		
		/* ... and put it at the head of the list */
		pce->ce_prev = 0;
		pce->ce_next = phe->he_ix;
		pcb->cb_cache[phe->he_ix].ce_prev = ixnew;
		phe->he_ix = ixnew;
		}
		
		pce->ce_status = CE_INUSE;
		pce->ce_hash = hash;
		pce->ce_keyptr = cagetmem(keylen);
		pce->ce_keylen = keylen;
		blkcopy(pce->ce_keyptr,pkey,keylen);
		pce->ce_resptr = cagetmem(reslen);
		pce->ce_reslen = reslen;
		blkcopy(pce->ce_resptr,pres,reslen);
		gettime(&pce->ce_tsinsert);
		pce->ce_tsaccess = pce->ce_tsinsert;

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		In a second example, caunlink(), which is called by other functions including caremove() and cagetindex(), removes a record from the linked list by modifying the values of ce_next and ce_prev in the records to which it was linked and then deletes the data stored in a record and frees memory by calling caclear(), as shown in the code below:

```
pce = &pcb->cb_cache[ix];
hash = pce->ce_hash;
phe = &pcb->cb_hash[HASHTOIX(hash,pcb)];

if (pce->ce_prev == NULL_IX)
phe->he_ix = pce->ce_next;
else
pcb->cb_cache[pce->ce_prev].ce_next = pce->ce_next;

pcb->cb_cache[pce->ce_next].ce_prev = pce->ce_prev;

caclear(pcb,ix);
```

To the extent that GCache does not disclose this limitation, GCache in combination with Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6") discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed.

One of ordinary skill in the art would be motivated to, and would understand how to, combine the method disclosed in GCache with the identifying at least some of the automatically expired ones of the records disclosed by NRL IPv6. *See, e.g.*, key.c at lines 1396-1563. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a method that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

		motivated to combine the linked list of NRL IPv6 with the method utilizing a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in NRL IPv6 is clearly shown in the chart of NRL IPv6, which is hereby incorporated by reference in its entirety.
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`struct key_acquirelist {`
		`u_int8 type; /* secassoc type to acquire */`
		`struct sockaddr_in6 target; /* destination address of secassoc */`
		`u_int32 count; /* number of acquire messages sent */`
		`u_long expiretime; /* expiration time for acquire message */`
		`struct key_acquirelist *next;`
		`};`
		In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. *See, e.g.*, key.c at lines 1411, 1430-1459.
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		192 u_long expiretime; /* expiration time for acquire message */ Furthermore, key.c discloses a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, as shown in the code below: 1445 } else if (ap->expiretime < time.tv_sec) { 1446 /* 1447 * Since we're already looking at the list, we may as 1448 * well delete expired entries as we scan through the list. 1449 * This should really be done by a function like key_reaper() 1450 * but until we code key_reaper(), this is a quick and dirty 1451 * hack. 1452 */ 1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n")); 1454 prevap->next = ap->next; 1455 KFree(ap); 1456 ap = prevap; 1457 } *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of	GCache discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, Comer discloses dynamically determining whether to delete one record or zero records from the accessed linked list of records:

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**

the records to remove when the linked list is accessed.	the records to remove when the linked list is accessed.	"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink() and returns. If the matching record is not expired, the code returns the index of the matched record:
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`
		`671 return(ix);`
		`672 }`
		
		In a second example, Comer discloses dynamically determining whether to delete one record or zero records from an accessed linked list of records:
		
		"Insertion of a new entry into a full cache forces the deletion of the entry that was looked up the least recently." *See* Comer at 3.
		
		At line 275 of gcache.c, cainsert() calls cagetindex() to access a linked list of records and see if a matching entry already exists. If a matching entry does not

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**

		exist, cainset() calls cagetfree() at line 281 to get a free entry. In cagetfree(), the following code dynamically determines whether to delete one record or zero records:
		
		`719 /* if the free list is empty, delete the oldest entry */`
		`720 if (pcb->cb_freelist == NULL_IX) {`
		`721 cadeleteold(pcb);`
		`722 ++pcb->cb_fulls;`
		`723 }`
		
		In addition, GCache combined with Dirks, Thatte, the ’663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.
		
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.
		
		For example, as summarized in Dirks,
		
		each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both GCache and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as GCache. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with GCache nothing more than

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with GCache and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCache with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both GCache and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining GCache with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**
		Further, one of ordinary skill in the art would be motivated to combine GCache with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in GCache can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining GCache with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine GCache with Thatte. Alternatively, it would also be obvious to combine GCache with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-10

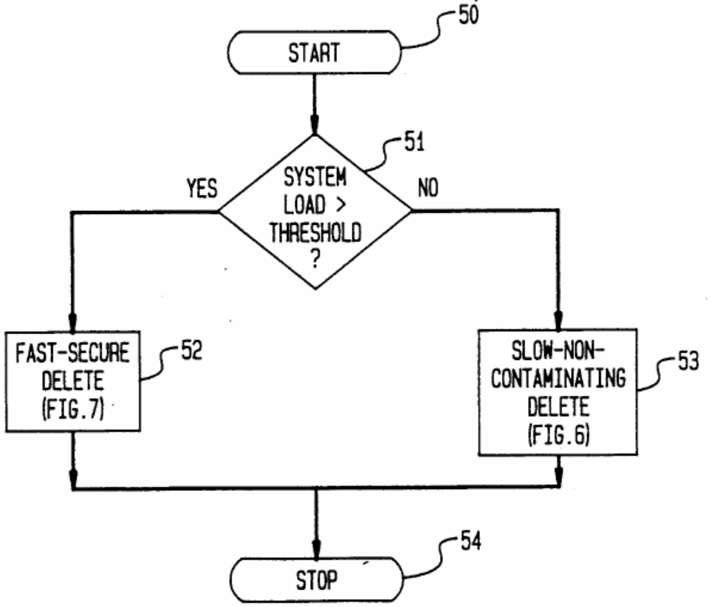
Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a

EXHIBIT D-10

Asserted Claims From **U.S. Pat. No. 5,893,120**		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both GCache and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in GCache. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with GCache would be nothing more than the predictable use of prior art elements according to

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with GCache and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine GCache with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id*. If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both GCache and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as GCache. Moreover, one of ordinary skill in the art

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with GCache would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with GCache and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in GCache to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in GCache with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”)**
		of records to solve a number of potential problems. For example, the removal of expired records described in GCache can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by GCache in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with GCache. For example, both Linux 2.0.1 and GCache describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		**gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")**
		`RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can

EXHIBIT D-10

Asserted Claims From U.S. Pat. No. 5,893,120		gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache")
		remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, NRL IPv6 discloses an information storage and retrieval system. For example, NRL IPv6 discloses a linked list of automatically expiring data. *See, e.g.*, struct_keyacquirelist defined in key.h at lines 188-194. *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	NRL IPv6 discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. NRL IPv6 in combination with Robert L. Kruse, Data Structures & Program Design (Prentice Hall 1987) (hereinafter "Kruse") discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with the hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address disclosed by Kruse. *See, e.g.*, Kruse at 206-208. For example, since NRL IPv6, as discussed below, utilizes a linked list for storing records and Kruse discloses attaching or chaining linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the

[1] Available as of August 1995.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		hash table using external chaining of linked lists disclosed by Kruse. Also, it is common practice among those of skill in the art to utilize techniques disclosed in textbooks such as Kruse in order to design and implement systems. The disclosure of these claim elements in Kruse is clearly shown in the chart of Kruse, which is hereby incorporated by reference in its entirety.
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with Kruse would be nothing more than the predictable use of prior art elements according to their established functions.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`188 struct key_acquirelist {`
		`189 u_int8 type; /* secassoc type to acquire */`
		`190 struct sockaddr_in6 target; /* destination address of secassoc */`
		`191 u_int32 count; /* number of acquire messages sent */`
		`192 u_long expiretime; /* expiration time for acquire message */`
		`193 struct key_acquirelist *next;`
		`194 };`
		In addition, key.c discloses traversing key_acquirelist and accessing entries stored therein. *See*, *e.g.*, key.c at lines 1411, 1430-1459.
		Also, key.c and key.h disclose automatically expiring records. For example,

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

		the key_acquirelist structure defined in key.h contains the following code:
		
		`u_long expiretime; /* expiration time for acquire message */`
		
		For example, key.c checks to see if a record has expired using the above-described field in key_acquirelist, as shown in the code below:
		
		`} else if (ap->expiretime < time.tv_sec) {`
		`/*`
		`* Since we're already looking at the list, we may as`
		`* well delete expired entries as we scan through the list.`
		`* This should really be done by a function like key_reaper()`
		`* but until we code key_reaper(), this is a quick and dirty`
		`* hack.`
		`*/`
		`DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n"));`
		`prevap->next = ap->next;`
		`KFree(ap);`
		`ap = prevap;`
		`}`
		
		Further, Kruse discloses hash tables with external chaining. *See, e.g.*, Kruse at 206-208. One of ordinary skill in the art would be motivated to, and would understand how to, combine the systems and methods of NRL IPv6 with the systems and methods of using hash tables with external chaining disclosed by Kruse.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194. To the extent that NRL IPv6 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring and also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache,

EXHIBIT D-11

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)**[1]
		which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses using linked lists to store records, the linked lists chained to a hash table using an external chaining technique: “GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion.” *See* Comer at 3. “GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table” *See* Comer at 5. “GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain.” *See* Comer at 6.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

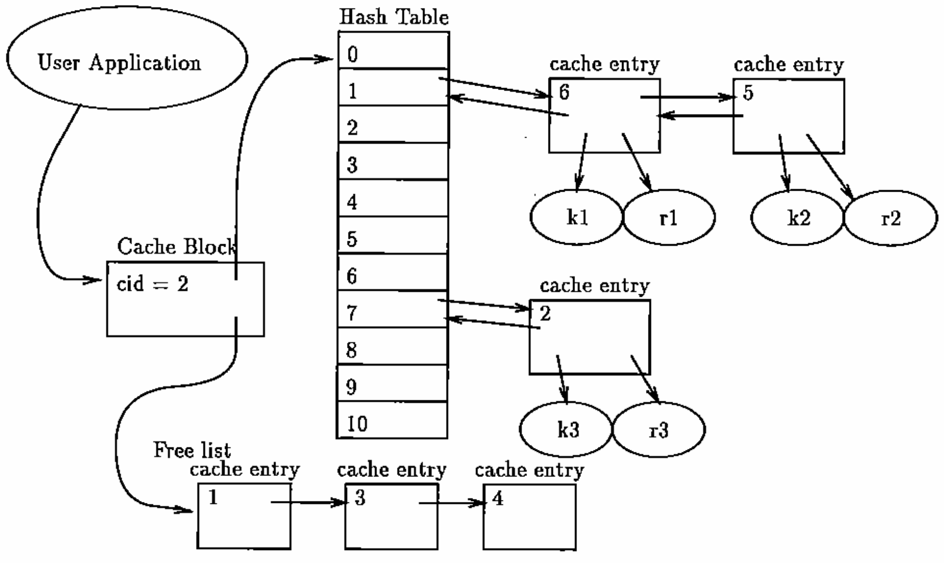
		
		 User Application
		Hash Table
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		cache entry
		6
		cache entry
		5
		k1
		r1
		k2
		r2
		Cache Block
		cid = 2
		cache entry
		2
		k3
		r3
		Free list
		cache entry
		1
		cache entry
		3
		cache entry
		4
		Figure 1: GCache Data Structures
		
		See also, gcache.c at lines 53-64, defining cacheentry as a linked list, as shown in the code below:
		
		`53 struct cacheentry {`
		`54 ce_status ce_status; /* INUSE or FREE */`
		`55 char *ce_keyptr; /* pointer to the key */`
		`56 tcelen ce_keylen; /* length of the key */`
		`57 char *ce_resptr; /* pointer to the result */`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		58 tcelen ce_reslen; /* length of the result */ 59 thval ce_hash; /* value that was hashed in */ 60 ttstamp ce_tsinsert; /* timestamp - time inserted */ 61 ttstamp ce_tsaccess; /* timestamp - last access */ 62 tceix ce_prev; /* next entry on list */ 63 tceix ce_next; /* prev entry on list */ 64 }; Comer discloses storing records in a linked list, for example: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gache.c at lines 241-304, defining cainsert(). Comer discloses providing access to records stored in a linked list, for example: "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 307-347 and 637-678, defining calookup() and cagetindex(). Comer discloses at least some of the records automatically expiring, for example:

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)**[1]

		“In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.” *See* Comer at 10.
		
		See also, gcache.c at lines 617-634, defining caisold() which determines if the record has expired:
		
		`617 /*`
		`618 * ====================================================================`
		`619 * caisold - return TRUE if the given entry is "too old"`
		`620 * ====================================================================`
		`621 */`
		`622 LOCAL int caisold(pcb,pce)`
		`623 struct cacheblk *pcb;`
		`624 struct cacheentry *pce;`
		`625 {`
		`626 unsigned now;`
		`627`
		`628 if (pcb->cb_maxlife == 0)`
		`629 return(FALSE);`
		`630`
		`631 gettime(&now);`
		`632`
		`633 return ((now - pce->ce_tsaccess) > pcb->cb_maxlife);`
		`634 }`

[1b] a record search means	[5b] a record search means	The combination of NRL IPv6 and Kruse discloses a record search means

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

utilizing a search key to access the linked list,	utilizing a search key to access a linked list of records having the same hash address,	utilizing a search key to access the linked list. The combination NRL IPv6 and Kruse also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.
		
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		
		`struct key_acquirelist {`
		`u_int8 type; /* secassoc type to acquire */`
		`struct sockaddr_in6 target; /* destination address of secassoc */`
		`u_int32 count; /* number of acquire messages sent */`
		`u_long expiretime; /* expiration time for acquire message */`
		`struct key_acquirelist *next;`
		`};`
		
		In addition, key.c discloses traversing key_acquirelist—accessing records stored therein--to search for matching record., as shown in the code below:
		
		`struct key_acquirelist *ap, *prevap;`
		`.`
		`.`
		`.`
		`prevap = key_acquirelist;`
		`for(ap = key_acquirelist->next; ap; ap = ap->next) {`
		`if (addrpart_equal(dst, (struct sockaddr *)&(ap->target)) &&`
		`(etype == ap->type)) {`
		`DPRINTF(IDL_MAJOR_EVENT,("acquire message previously sent!\n"));`
		`if (ap->expiretime < time.tv_sec) {`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)**[1]

		`1436 DPRINTF(IDL_MAJOR_EVENT,("acquire message has expired!\n"));`
		`1437 ap->count = 0;`
		`1438 break;`
		`1439 }`
		
		Further, Kruse discloses hash tables and hash tables with external chaining. *See, e.g.*, Kruse at 198-208. One of ordinary skill in the art would be motivated to, and would understand how to, combine the systems and methods of NRL IPv6 with the systems and methods of using hash tables with external chaining disclosed by Kruse. In such a combination, one of ordinary skill in the art would recognize that a hash key is used to access a list of records having the same hash address (a linked list chained to a hash bucket).
		
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
		
		To the extent that NRL IPv6 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses a record search means utilizing a search key to access the linked list, and also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.
		
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with the a hashing means to provide access to records stored in a memory of the system and using an external

EXHIBIT D-11

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]</th></tr>
<tr><td></td><td></td><td>chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. <i>See, e.g.</i>, Comer at 3-10. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety.

Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions.

For example, Comer discloses utilizing a search key to access a linked list of records having the same hash address. In the quoted text below, the use of hash value to determine a hash table with an attached linked list of records having the same hash address is an example of “utilizing a search key to access a linked list.”

“GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion.” <i>See</i> Comer at 3.

“GCache keeps all <i>cacheentry</i> structures for an active cache either on the free</td></tr>
</table>

EXHIBIT D-11

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]</th></tr>
<tr><td></td><td></td><td>

list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table” *See* Comer at 5.

“GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain.” *See* Comer at 6.

See also, gcache.c at lines 637-677, defining cagetindex(), which contains code constituting a “record search means” that uses a hash value to access and traverse a linked list of records having the same hash address:

<pre>
637 /*
638 *
===
=====
639 * cagetindex - return the index of a matching entry, or
SYSERR
640 * N.B. assumes MUTEX is already held
641 *
===
=====
642 */
643 LOCAL tceix cagetindex(pcb,pkey,keylen,hash)
644 struct cacheblk *pcb;
645 char *pkey;
646 tcelen keylen;
647 thval hash;
648 {
649 struct cacheentry *pce;
</pre>

</td></tr>
</table>

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		tceix ix;
		tceix nextix;
		
		++pcb->cb_lookups;
		
		ix = pcb->cb_hash[HASHTOIX(hash,pcb)].he_ix;
		
		while (ix != NULL_IX) {
		pce = &pcb->cb_cache[ix];
		nextix = pce->ce_next;
		
		if ((pce->ce_hash == hash) &&
		(pce->ce_keylen == keylen) &&
		(blkequ(pkey,pce->ce_keyptr,keylen))) {
		/* this is a match */
		++pcb->cb_hits;
		if (caisold(pcb,pce)) {
		++pcb->cb_tos;
		caunlink(pcb,ix);
		return(NULL_IX);
		} else {
		return(ix);
		}
		}
		ix = nextix;
		}
		
		return(NULL_IX);
		}

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the	NRL IPv6 discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. NRL IPv6 also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

records from the linked list when the linked list is accessed, and	linked list of records when the linked list is accessed, and	accessed.
		
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		
		`188 struct key_acquirelist {`
		`189 u_int8 type; /* secassoc type to acquire */`
		`190 struct sockaddr_in6 target; /* destination address of secassoc */`
		`191 u_int32 count; /* number of acquire messages sent */`
		`192 u_long expiretime; /* expiration time for acquire message */`
		`193 struct key_acquirelist *next;`
		`194 };`
		
		In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. *See, e.g.*, key.c at lines 1411, 1430-1459.
		
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:
		
		`192 u_long expiretime; /* expiration time for acquire message */`
		
		`Furthermore, key.c discloses` a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, as shown in the code below：
		
		`1445 } else if (ap->expiretime < time.tv_sec) {`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		1446 /* 1447 * Since we're already looking at the list, we may as 1448 * well delete expired entries as we scan through the list. 1449 * This should really be done by a function like key_reaper() 1450 * but until we code key_reaper(), this is a quick and dirty 1451 * hack. 1452 */ 1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n")); 1454 prevap->next = ap->next; 1455 KFree(ap); 1456 ap = prevap; 1457 } *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	NRL IPv6 discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. NRL IPv6 also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. The "means, utilizing the record search means" limitation is met, for example, by a function that calls key_acquire(). At line 1835 of key.c, for example, key_acquire() is called by the function getassocbysocket(). In addition, the function key_acquire() in key.c contains a *for* loop beginning

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120	Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

at line 1431. Within the *for* loop, the code starting at the *elseif* at line 1445 and ending at line 1457, modifies a pointer in an element of the linked list such that it removes the expired item from the linked list and then calls KFree(). After KFree() is called, control returns to the *for* loop which, unless it has reached the end of the linked list, will access the next record in the list. If the record has not been previously sent and has expired, it will be removed and deleted in the code starting at the *elseif* at line 1445 and ending at line 1457. Finally, after the loop described above completes, key_acquire() contains the following code to insert an entry into key_acquirelist:

```
1542 /*
1543 * Update the acquirelist
1544 */
1545 if (success) {
1546 if (!ap) {
1547 DPRINTF(IDL_MAJOR_EVENT,("Adding new entry in
acquirelist\n"));
1548 K_Malloc(ap, struct key_acquirelist *, sizeof(struct
key_acquirelist));
1549 if (ap == 0)
1550 return(success ? 0 : -1);
1551 bzero((char *)ap, sizeof(struct key_acquirelist));
1552 bcopy((char *)dst, (char *)&(ap->target), dst->sa_len);
1553 ap->type = etype;
1554 ap->next = key_acquirelist->next;
1555 key_acquirelist->next = ap;
1556 }
1557 DPRINTF(IDL_EVENT,("Updating acquire counter and
expiration
time\n"));
1558 ap->count++;
1559 ap->expiretime = time.tv_sec + maxacquiretime;
```

EXHIBIT D-11

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)**[1]
		`1560 }` *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194. To the extent that NRL IPv6 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter “gcache.c”) and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter “Comer”) (collectively hereinafter “GCache”) discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache,

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove().

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	NRL IPv6 discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.
		For example, in key.c each time the else if statement at line 1445 is executed, it dynamically determines the maximum number of records to remove—one or zero—based on whether the record has expired.
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`
		`1449 * This should really be done by a function like`
		`key_reaper()`
		`1450 * but until we code key_reaper(), this is a quick and`
		`dirty`
		`1451 * hack.`
		`1452 */`
		`1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired`
		`entry...deleting`
		`it!\n"));`
		`1454 prevap->next = ap->next;`
		`1455 KFree(ap);`
		`1456 ap = prevap;`
		`1457 }`
		Further, NRL IPv6 combined with Dirks, Thatte, the '663 patent and/or the

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both NRL IPv6 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as NRL IPv6. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with NRL IPv6 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with NRL IPv6 and would

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both NRL IPv6 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine NRL IPv6 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in NRL IPv6 can be burdensome on the system, adding to the system's load and slowing down the system's

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		processing. One of ordinary skill in the art would recognize that combining NRL IPv6 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine NRL IPv6 with Thatte. Alternatively, it would also be obvious to combine NRL IPv6 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]**
		no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-11

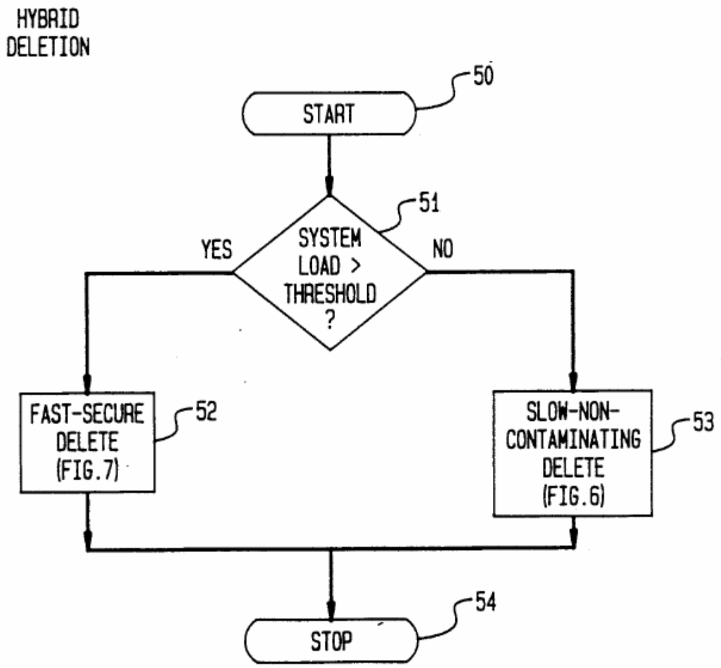
Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64,

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both NRL IPv6 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in NRL IPv6. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		combining the '663 patent's deletion decision procedure with NRL IPv6 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with NRL IPv6 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine NRL IPv6 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged;

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both NRL IPv6 and the Opportunistic Garbage Collection Articles relate to

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as NRL IPv6. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with NRL IPv6 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with NRL IPv6 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in NRL IPv6 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in NRL IPv6 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in NRL IPv6 can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194. To the extent that dynamically determining a maximum number of expired records is not disclosed by NRL IPv6 in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with NRL IPv6. For example, both Linux 2.0.1 and NRL IPv6 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list.

EXHIBIT D-11

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]**
		See Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least	To the extent the preamble is a limitation, NRL IPv6 discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. The combination of NRL IPv6 and Kruse discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)**[1]

steps of:	some of the records automatically expiring, the method comprising the steps of:	One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address as disclosed by Kruse. *See, e.g.*, Kruse at 206-208. For example, since NRL IPv6, as discussed below, utilizes a linked list for storing records and Kruse discloses attaching or chaining linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the hash table using external chaining of linked lists disclosed by Kruse. Also, it is common practice among those of skill in the art to utilize techniques disclosed in textbooks such as Kruse in order to design and implement systems. The disclosure of these claim elements in Kruse is clearly shown in the chart of Kruse, which is hereby incorporated by reference in its entirety.
		Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with Kruse would be nothing more than the predictable use of prior art elements according to their established functions.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`struct key_acquirelist {`
		`u_int8 type; /* secassoc type to acquire */`
		`struct sockaddr_in6 target; /* destination address of secassoc */`
		`u_int32 count; /* number of acquire messages sent */`
		`u_long expiretime; /* expiration time for acquire message`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		*/ 193 struct key_acquirelist *next; 194 }; In addition, key.c discloses traversing key_acquirelist and accessing entries stored therein. *See*, *e.g.*, key.c at lines 1411, 1430-1459. Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code: 192 u_long expiretime; /* expiration time for acquire message */ For example, key.c checks to see if a record has expired using the above-described field in key_acquirelist, as shown in the code below: 1445 } else if (ap->expiretime < time.tv_sec) { 1446 /* 1447 * Since we're already looking at the list, we may as 1448 * well delete expired entries as we scan through the list. 1449 * This should really be done by a function like key_reaper() 1450 * but until we code key_reaper(), this is a quick and dirty 1451 * hack. 1452 */ 1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n")); 1454 prevap->next = ap->next; 1455 KFree(ap); 1456 ap = prevap; 1457 }

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		Further, Kruse discloses hash tables with external chaining. *See, e.g.*, Kruse at 206-208. One of ordinary skill in the art would be motivated to, and would understand how to, combine the systems and methods of NRL IPv6 with the systems and methods of using hash tables with external chaining disclosed by Kruse. *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194. To the extent that the preamble is a limitation and to the extent that NRL IPv6 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, and discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. One of ordinary skill in the art would be motivated to, and would understand how to, combine the method disclosed in NRL IPv6 with the method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a method that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the method utilizing a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses using linked lists to store records, the linked lists chained to a hash table using an external chaining technique: "GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion." *See* Comer at 3. "GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table" *See* Comer at 5.

EXHIBIT D-11

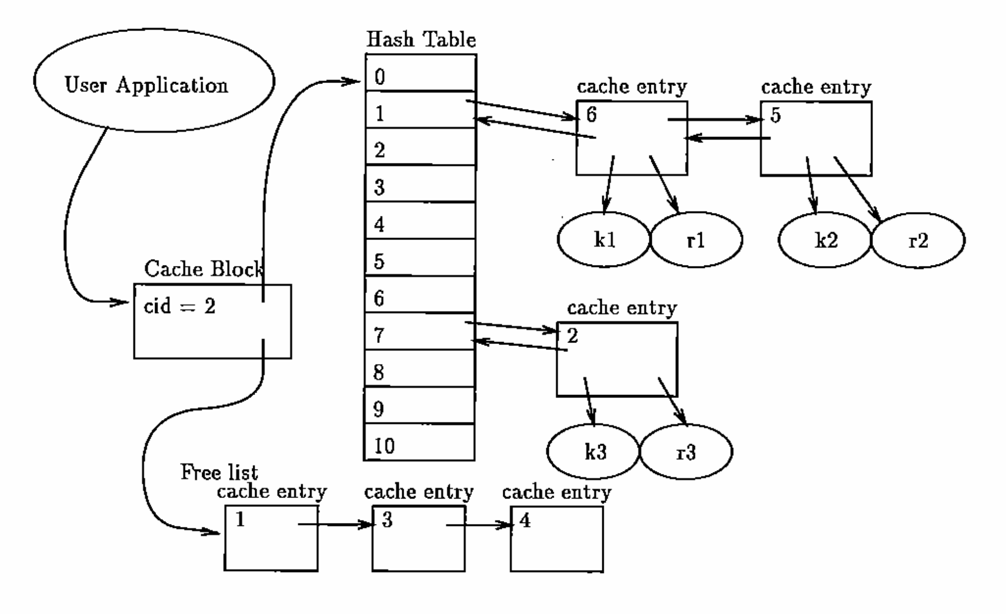
Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]**
		"GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain." *See* Comer at 6.  User Application Hash Table 0 1 2 3 4 5 6 7 8 9 10 cache entry 6 cache entry 5 k1 r1 k2 r2 Cache Block cid = 2 cache entry 2 k3 r3 Free list cache entry 1 cache entry 3 cache entry 4 Figure 1: GCache Data Structures *See also*, gcache.c at lines 53-64, defining cacheentry as a linked list, as shown in the code below:

EXHIBIT D-11

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]</th></tr>
<tr><td></td><td></td><td><pre>
53 struct cacheentry {
54 ce_status ce_status; /* INUSE or FREE */
55 char *ce_keyptr; /* pointer to the key */
56 tcelen ce_keylen; /* length of the key */
57 char *ce_resptr; /* pointer to the result */
58 tcelen ce_reslen; /* length of the result */
59 thval ce_hash; /* value that was hashed in */
60 ttstamp ce_tsinsert; /* timestamp - time inserted */
61 ttstamp ce_tsaccess; /* timestamp - last access */
62 tceix ce_prev; /* next entry on list */
63 tceix ce_next; /* prev entry on list */
64 };
</pre>
<p>Comer discloses storing records in a linked list, for example:</p>
<p>"<i>Cainsert</i>() inserts a new mapping, key => res, into the cache." <i>See</i> Comer at 4.</p>
<p><i>See also</i>, gache.c at lines 241-304, defining cainsert().</p>
<p>Comer discloses providing access to records stored in a linked list, for example:</p>
<p>"<i>Calookup</i>() searches for a cached entry matching the key passed as an argument." <i>See</i> Comer at 4.</p>
<p><i>See also</i>, gcache.c at lines 307-347 and 637-678, defining calookup() and</p></td></tr>
</table>

EXHIBIT D-11

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]</th></tr>
<tr><td></td><td></td><td>

cagetindex().

Comer discloses at least some of the records automatically expiring, for example:

“In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.” *See* Comer at 10.

See also, gcache.c at lines 617-634, defining caisold() which determines if the record has expired:

```
617 /*
618 *
==============================================================
=====
619 * caisold - return TRUE if the given entry is "too old"
620 *
==============================================================
=====
621 */
622 LOCAL int caisold(pcb,pce)
623 struct cacheblk *pcb;
624 struct cacheentry *pce;
625 {
626 unsigned now;
627
628 if (pcb->cb_maxlife == 0)
629 return(FALSE);
630
```

</td></tr>
</table>

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		631 gettime(&now);
		632
		633 return ((now - pce->ce_tsaccess) > pcb->cb_maxlife);
		634 }

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	NRL IPv6 discloses accessing a linked list of records. The combination of NRL IPv6 and Kruse discloses accessing a linked list of records having same hash address.
		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		188 struct key_acquirelist {
		189 u_int8 type; /* secassoc type to acquire */
		190 struct sockaddr_in6 target; /* destination address of secassoc */
		191 u_int32 count; /* number of acquire messages sent */
		192 u_long expiretime; /* expiration time for acquire message */
		193 struct key_acquirelist *next;
		194 };
		In addition, key.c discloses traversing key_acquirelist—accessing records stored therein--to search for matching record, as shown in the code below:
		1411 struct key_acquirelist *ap, *prevap;
		.
		.
		.
		1430 prevap = key_acquirelist;
		1431 for(ap = key_acquirelist->next; ap; ap = ap->next) {
		1432 if (addrpart_equal(dst, (struct sockaddr *)&(ap->target))

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		&&
		1433 (etype == ap->type)) {
		1434 DPRINTF(IDL_MAJOR_EVENT,("acquire message previously sent!\n"));
		1435 if (ap->expiretime < time.tv_sec) {
		1436 DPRINTF(IDL_MAJOR_EVENT,("acquire message has expired!\n"));
		1437 ap->count = 0;
		1438 break;
		1439 }
		
		Further, Kruse discloses hash tables and hash tables with external chaining. *See, e.g.*, Kruse at 198-208. One of ordinary skill in the art would be motivated to, and would understand how to, combine the systems and methods of NRL IPv6 with the systems and methods of using hash tables with external chaining disclosed by Kruse. In such a combination, one of ordinary skill in the art would recognize that a hash key is used to access a list of records having the same hash address (a linked list chained to a hash bucket).
		
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
		
		To the extent that NRL IPv6 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses accessing a linked list of records, and also discloses accessing a linked list of records having same hash address.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the method disclosed in NRL IPv6 with the method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a method that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the method utilizing a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses accessing a linked list of records having the same hash address: "GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion." *See* Comer at 3.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		"GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of bytes in the key buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table" *See* Comer at 5.
		"GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain." *See* Comer at 6.
		See also, gcache.c at lines 637-677, defining cagetindex(), which contains code constituting a "record search means" that uses a hash value to access and traverse a linked list of records having the same hash address:
		`637 /*`
		`638 *`
		`===============================================================`
		`=====`
		`639 * cagetindex - return the index of a matching entry, or`
		`SYSERR`
		`640 * N.B. assumes MUTEX is already held`
		`641 *`
		`===============================================================`
		`=====`
		`642 */`
		`643 LOCAL tceix cagetindex(pcb,pkey,keylen,hash)`
		`644 struct cacheblk *pcb;`
		`645 char *pkey;`
		`646 tcelen keylen;`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

		thval hash;
		{
		struct cacheentry *pce;
		tceix ix;
		tceix nextix;
		
		++pcb->cb_lookups;
		
		ix = pcb->cb_hash[HASHTOIX(hash,pcb)].he_ix;
		
		while (ix != NULL_IX) {
		pce = &pcb->cb_cache[ix];
		nextix = pce->ce_next;
		
		if ((pce->ce_hash == hash) &&
		(pce->ce_keylen == keylen) &&
		(blkequ(pkey,pce->ce_keyptr,keylen))) {
		/* this is a match */
		++pcb->cb_hits;
		if (caisold(pcb,pce)) {
		++pcb->cb_tos;
		caunlink(pcb,ix);
		return(NULL_IX);
		} else {
		return(ix);
		}
		}
		ix = nextix;
		}
		
		return(NULL_IX);
		}

[3b] identifying at least some of the automatically expired ones of the records,	[7b] identifying at least some of the automatically expired ones of the records,	NRL IPv6 discloses identifying at least some of the automatically expired ones of the records.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

and		For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list:
		`188 struct key_acquirelist {`
		`189 u_int8 type; /* secassoc type to acquire */`
		`190 struct sockaddr_in6 target; /* destination address of secassoc */`
		`191 u_int32 count; /* number of acquire messages sent */`
		`192 u_long expiretime; /* expiration time for acquire message */`
		`193 struct key_acquirelist *next;`
		`194 };`
		In addition, key.c, within the function key_acquire(), discloses traversing key_acquirelist and accessing entries stored therein. *See*, *e.g.*, key.c at lines 1411, 1430-1459.
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:
		`192 u_long expiretime; /* expiration time for acquire message */`
		`Furthermore, key.c discloses` a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, as shown in the code below：
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		1449 * This should really be done by a function like key_reaper() 1450 * but until we code key_reaper(), this is a quick and dirty 1451 * hack. 1452 */ 1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n")); 1454 prevap->next = ap->next; 1455 KFree(ap); 1456 ap = prevap; 1457 } *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	NRL IPv6 discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, as defined in key.h and shown in the code below, the key_acquirelist structure is a linked list: 188 struct key_acquirelist { 189 u_int8 type; /* secassoc type to acquire */ 190 struct sockaddr_in6 target; /* destination address of secassoc */ 191 u_int32 count; /* number of acquire messages sent */ 192 u_long expiretime; /* expiration time for acquire message */ 193 struct key_acquirelist *next; 194 }; In addition, key.c, within the function key_acquire(), discloses traversing

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		key_acquirelist and accessing entries stored therein. *See, e.g.*, key.c at lines 1411, 1430-1459.
		
		Also, key.c and key.h disclose automatically expiring records. For example, the key_acquirelist structure defined in key.h contains the following code:
		
		`192 u_long expiretime; /* expiration time for acquire message */`
		
		Furthermore, key.c discloses a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, as shown in the code below:
		
		`1445 } else if (ap->expiretime < time.tv_sec) {`
		`1446 /*`
		`1447 * Since we're already looking at the list, we may as`
		`1448 * well delete expired entries as we scan through the list.`
		`1449 * This should really be done by a function like key_reaper()`
		`1450 * but until we code key_reaper(), this is a quick and dirty`
		`1451 * hack.`
		`1452 */`
		`1453 DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting it!\n"));`
		`1454 prevap->next = ap->next;`
		`1455 KFree(ap);`
		`1456 ap = prevap;`
		`1457 }`
		
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	NRL IPv6 discloses inserting, retrieving or deleting one of the records from the system following the step of removing.
		
		For example, the function key_acquire() in key.c contains a *for* loop beginning at line 1431. Within the *for* loop, the code starting at the *elseif* at line 1445 and ending at line 1457, modifies a pointer in an element of the linked list such that it removes the expired item from the linked list and then calls KFree(). After KFree() is called, control returns to the *for* loop which, unless it has reached the end of the linked list, will retrieve the next record in the list. If the record has not been previously sent and has expired, it will be removed and deleted in the code starting at the *elseif* at line 1445 and ending at line 1457. Finally, after the loop described above completes, key_acquire() contains the following code to insert an entry into key_acquirelist:
		
		`1542 /*`
		`1543 * Update the acquirelist`
		`1544 */`
		`1545 if (success) {`
		`1546 if (!ap) {`
		`1547 DPRINTF(IDL_MAJOR_EVENT,("Adding new entry in acquirelist\n"));`
		`1548 K_Malloc(ap, struct key_acquirelist *, sizeof(struct key_acquirelist));`
		`1549 if (ap == 0)`
		`1550 return(success ? 0 : -1);`
		`1551 bzero((char *)ap, sizeof(struct key_acquirelist));`
		`1552 bcopy((char *)dst, (char *)&(ap->target), dst->sa_len);`
		`1553 ap->type = etype;`
		`1554 ap->next = key_acquirelist->next;`
		`1555 key_acquirelist->next = ap;`
		`1556 }`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		`1557 DPRINTF(IDL EVENT,("Updating acquire counter and expiration time\n"));`
		`1558 ap->count++;`
		`1559 ap->expiretime = time.tv_sec + maxacquiretime;`
		`1560 }`
		
		See also, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.
		
		To the extent that NRL IPv6 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses discloses inserting, retrieving or deleting one of the records from the system following the step of removing.
		
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the method disclosed in NRL IPv6 with the method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See*, *e.g.*, Comer at 3-10. For example, since NRL IPv6 utilizes a linked list for storing records and GCache discloses a method that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of NRL IPv6 with the method utilizing a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. At line 275 of gcache.c, cainsert() utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. "In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10. At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and if so, removes the expired record

EXHIBIT D-11

<table>
<tr>
<th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th>
<th>Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]</th>
</tr>
<tr>
<td></td>
<td></td>
<td>

from the linked list using caunlink():

```
if (caisold(pcb,pce)) {
++pcb->cb_tos;
caunlink(pcb,ix);
return(NULL_IX);
} else {
```

After the call to cagetindex() returns, through which the expired entry was removed, cainsert() proceeds to insert a new entry at the head of the list and populates the fields of the structure, as shown in the code below:

```
if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
/* use the old one */
caclear(pcb,ixnew);
pce = &pcb->cb_cache[ixnew];
} else {
/* get a free cacheentry */
ixnew = cagetfree(pcb);
pce = &pcb->cb_cache[ixnew];

/* ... and put it at the head of the list */
pce->ce_prev = 0;
pce->ce_next = phe->he_ix;
pcb->cb_cache[phe->he_ix].ce_prev = ixnew;
phe->he_ix = ixnew;
}

pce->ce_status = CE_INUSE;
pce->ce_hash = hash;
pce->ce_keyptr = cagetmem(keylen);
pce->ce_keylen = keylen;
```

</td>
</tr>
</table>

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]

		295 blkcopy(pce->ce_keyptr,pkey,keylen);
		296 pce->ce_resptr = cagetmem(reslen);
		297 pce->ce_reslen = reslen;
		298 blkcopy(pce->ce_resptr,pres,reslen);
		299 gettime(&pce->ce_tsinsert);
		300 pce->ce_tsaccess = pce->ce_tsinsert;
		
		In a second example, caunlink(), which is called by other functions including caremove() and cagetindex(), removes a record from the linked list by modifying the values of ce_next and ce_prev in the records to which it was linked and then deletes the data stored in a record and frees memory by calling caclear(), as shown in the code below:
		
		750 pce = &pcb->cb_cache[ix];
		751 hash = pce->ce_hash;
		752 phe = &pcb->cb_hash[HASHTOIX(hash,pcb)];
		753
		754 if (pce->ce_prev == NULL_IX)
		755 phe->he_ix = pce->ce_next;
		756 else
		757 pcb->cb_cache[pce->ce_prev].ce_next = pce->ce_next;
		758
		759 pcb->cb_cache[pce->ce_next].ce_prev = pce->ce_prev;
		760
		761 caclear(pcb,ix);

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of	NRL IPv6 discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, in key.c each time the else if statement at line 1445 is executed, it dynamically determines the maximum number of records to remove—one or

<u>**EXHIBIT D-11**</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")**[1]

the records to remove when the linked list is accessed.	the records to remove when the linked list is accessed.	zero—based on whether the record has expired.
		`} else if (ap->expiretime < time.tv_sec) {`
		`/*`
		`* Since we're already looking at the list, we may as`
		`* well delete expired entries as we scan through the list.`
		`* This should really be done by a function like`
		`key_reaper()`
		`* but until we code key_reaper(), this is a quick and`
		`dirty`
		`* hack.`
		`*/`
		`DPRINTF(IDL_MAJOR_EVENT,("found an expired`
		`entry...deleting`
		`it!\n"));`
		`prevap->next = ap->next;`
		`KFree(ap);`
		`ap = prevap;`
		`}`
		In addition, NRL IPv6 combined with Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both NRL IPv6 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		implementations such as NRL IPv6. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with NRL IPv6 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with NRL IPv6 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in NRL IPv6 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]
		Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both NRL IPv6 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining NRL IPv6 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine NRL IPv6 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in NRL IPv6 can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining NRL IPv6 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine NRL IPv6 with Thatte.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		Alternatively, it would also be obvious to combine NRL IPv6 with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT D-11

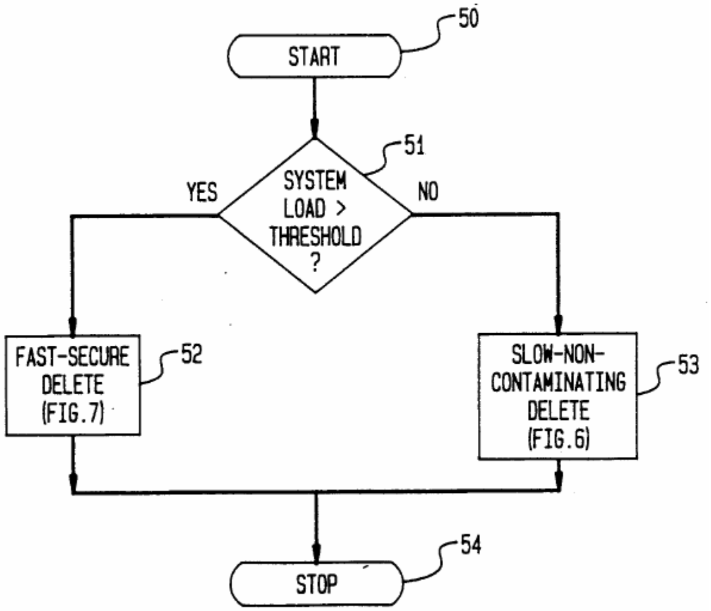
Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter “NRL IPv6”)[1]**
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 YES SYSTEM LOAD > THRESHOLD ? NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both NRL IPv6 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in NRL IPv6. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with NRL IPv6 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with NRL IPv6 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine NRL IPv6 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part:

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both NRL IPv6 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as NRL IPv6. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with NRL IPv6 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with NRL IPv6 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		modify the system disclosed in NRL IPv6 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in NRL IPv6 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in NRL IPv6 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. *See also*, key.c at lines 1396-1563, 1768-1845; key.h at lines 188-194.

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")**[1]
		To the extent that dynamically determining a maximum number of expired records is not disclosed by NRL IPv6 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with NRL IPv6. For example, both Linux 2.0.1 and NRL IPv6 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1`

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")[1]
		than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete,

EXHIBIT D-11

Asserted Claims From U.S. Pat. No. 5,893,120		**Naval Research Laboratories ipv6-dist-domestic\sys.common\netinet6\key.c and key.h (hereinafter "NRL IPv6")**[1]
		such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-12

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, LINUX 2.0.1 discloses an information storage and retrieval system. For example, LINUX 2.0.1 includes a hash table `ip_rt_hash_table`. The hash table is an information storage and retrieval system.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	LINUX 2.0.1 discloses a linked list to store and provide access to records stored in a memory of the system. LINUX 2.0.1 also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, the hash table is an array of pointers to `rtable` structures (i.e., records). See line 151. Each `rtable` structure contains an `rt_next` field, which is a pointer to another `rtable` structure. See /include/net/route.h, line 67. Accordingly, `rtable` structures can be linked to form linked lists. LINUX 2.0.1 discloses that at least some of the records automatically expire. The `rtable` structure includes an `rt_lastuse` field. Functions in route.c use a record's `rt_lastuse` field to determine whether the record has automatically expired. See /include/net/route.h, line 74 and analysis below.

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	LINUX 2.0.1 discloses a record search means utilizing a search key to access the linked list. LINUX 2.0.1 also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, the hash table contains an array of linked lists of `rtable` structures. The hash table is accessed using a search key. Specifically, route.c includes a function `rt_cache_add`. Lines 1299-1385. The function `rt_cache_add` includes a first code set (line 1356 and lines 1365-1383). The first code set uses the search key `hash` to access a linked list at the `hash` index of the hash table. See lines 1345 and 1356.
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	LINUX 2.0.1 discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. LINUX 2.0.1 also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. For example, the function `rt_cache_add` includes a first code set, as described above. The first code set includes a second code set (lines 1365-1383). The second code set defines a loop that iterates through a linked list to remove "automatically expired" records. Specifically, line 1369 determines whether the record has expired when more than a particular amount of time has passed since the record was last used.

Asserted Claims from U.S. Pat. No. 5,893,120		Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("Linux 2.0.1") alone and in combination
		Also, Linux 2.0.1 removes expired records from the linked list, as claimed. For example, line 1372 adjusts the pointers to de-link the expired record from the linked list. And line 1378 invokes the `rt_free` function. The `rt_free` function frees the memory allocated to the expired record. See lines 881-905. The function `rt_cache_add` includes code that inserts a new record into the linked list. Lines 1356-1357. Inserting a new record into the linked list is one way of accessing the linked list. The code to insert the new record into the linked list and the second code set are executed in the same invocation of the function `rt_cache_add`. Hence, the first code set includes means (i.e., the second code set) for identifying and removing expired records from a linked list when the function `rt_cache_add` accesses the linked list.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] meals[sic], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	Linux 2.0.1 discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. Linux 2.0.1 also discloses meals[sic], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. For example, the function `rt_cache_add` utilizes the first code set (i.e., the record search means) as described above. Furthermore, the function

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		rt_cache_add accesses the linked list. See lines 1356-1359 (inserting a record into the linked list). In the same invocation (i.e., at the same time), the function rt_cache_add performs the first code set to remove at least some of the expired ones of the records in the linked list as described above. The function rt_cache_add also retrieves records from the system. See lines 1347-1354 (looping through and printing the rt_dst element of each record in the linked list). In addition, the function rt_cache_add deletes records from the system (i.e., hash table). The function rt_cache_add invokes the function rt_garbage_collect. Line 1432. The function rt_garbage_collect invokes a function rt_garbage_collect_1. Line 1293. The function rt_garbage_collect_1 loops through each of the linked lists in the hash table and removes records from the linked lists. See lines 1122-1138. The function rt_garbage_collect_1 can remove expired ones of the records in the linked list plus other records in the hash table. See lines 1122-1138. Hence, by invoking the function rt_garbage_collect, the function rt_cache_add deletes records from the system. Because the function rt_cache_add utilizes the first code set, inserts records into the system, retrieves records from the system, and deletes records from the system, the function rt_cache_add is a means utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time (i.e., in the same invocation of the

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		function `rt_cache_add`), removes at least some expired ones of the records in the accessed linked list of records. To the extent that Linux 2.0.1 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 2.0.1 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since Linux 2.0.1 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of Linux 2.0.1 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining Linux 2.0.1 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4.

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination

		See also, gcache.c at lines 355-376, defining caremove().
		
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination

		caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1, further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5, further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	LINUX 2.0.1 discloses means for dynamically determining maximum number of expired ones of the records to remove in the accessed linked list of records. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. Line 1359. When the function `rt_cache_add` removes an expired record from a linked list in the hash table, the function `rt_cache_add` decrements the variable `rt_cache_size`. Line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table. Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1` in route.c. The function `rt_garbage_collect_1` loops through each of the linked lists in the hash table `ip_rt_hash_table`. See lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is

Asserted Claims from U.S. Pat. No. 5,893,120		**Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("Linux 2.0.1") alone and in combination**
		expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. See lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired records for the function `rt_garbage_collect_1` to remove from the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. See lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. Line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. See lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. See lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. See line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. See lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. Line 1122.

Asserted Claims from U.S. Pat. No. 5,893,120		Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("Linux 2.0.1") alone and in combination
		The variable expire is initially one half of a fixed timeout value `RT_CACHE_TIMEOUT`. Line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. See line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. Line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. Line 1122. Rather, the

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list. Thus, by determining at line 1341 whether to invoke the function `rt_garbage_collect`, the function `rt_cache_add` dynamically determines the maximum number of records that can be removed from a linked list in the hash table. Bedrock's proposed claim constructions assert that this element corresponds to portions of application software, user access software, or operating system software… that perform the function of determining, based upon one or more factors existing at the time the record search means is invoked, maximum number of record for the search means to remove in the linked list of records. Furthermore, Bedrock points to col. 6, line 56 – col. 7, line 15 of the '120 Patent which "describe code that chooses among removal options at the time the record search means is invoked by the caller, thus sometimes removing all expired records, at other times removing some but not all of them, and yet at other times choosing to remove none of them, or the equivalent thereof." (Emphasis added).

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		The total number of records in the hash table is a factor existing at the time the function `rt_cache_add` is invoked. See lines 1299-1340 (not changing the variable `rt_cache_size`). By determining at line 1341 whether to invoke the function `rt_garbage_collect`, the function `rt_cache_add` chooses among removal options at the time the function `rt_cache_add` is invoked by a caller of the function `rt_cache_add`. The different removal options (i.e., using the function `rt_garbage_collect_1` or not using the function `rt_garbage_collect_1`) can remove different numbers of records from a linked list in the hash table. Further, Linux 2.0.1 combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or

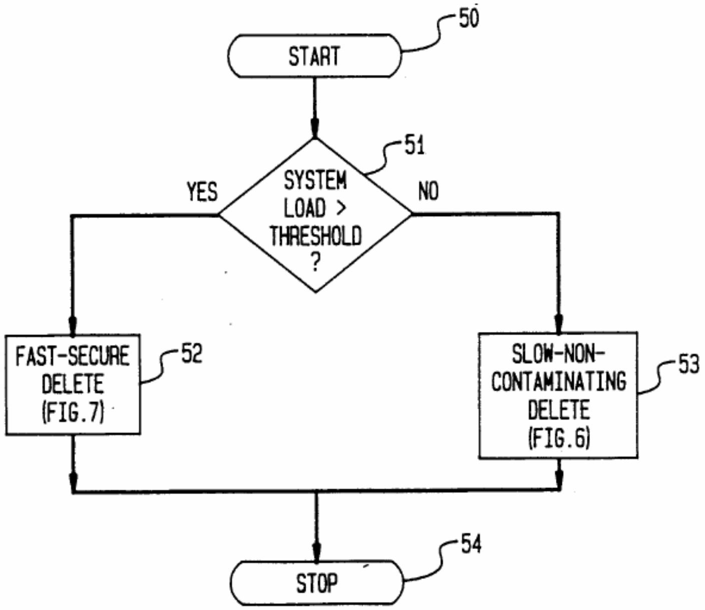
Asserted Claims from **U.S. Pat. No. 5,893,120**		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where:

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 2.0.1 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 2.0.1. Moreover, one of

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with Linux 2.0.1 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 2.0.1 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.\` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 2.0.1 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte,

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 2.0.1 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 2.0.1 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 2.0.1 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in Linux 2.0.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining Linux 2.0.1 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 2.0.1 with Thatte.

Asserted Claims from U.S. Pat. No. 5,893,120		Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("Linux 2.0.1") alone and in combination
		Alternatively, it would also be obvious to combine Linux 2.0.1 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The ’663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5. FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5.

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 2.0.1 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 2.0.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with Linux 2.0.1 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Linux 2.0.1 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 2.0.1 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part:

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both Linux 2.0.1 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 2.0.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 2.0.1 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 2.0.1 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system.

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 2.0.1 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Linux 2.0.1 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 2.0.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15.

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, LINUX 2.0.1 discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. To the extent the preamble is a limitation, LINUX 2.0.1 also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, LINUX 2.0.1 includes a hash table `ip_rt_hash_table`. The hash table is an information storage and retrieval system. The hash table uses linked lists to store and provide access to records. The hash table is an array of pointers to `rtable` structures. See line 151. Each rtable structure includes an `rt_next` element that can point to another `rtable` element. See /include/net/route.h, line 67. In this way, linked lists of `rtable` structures are formed by setting the `rt_next` elements to point to other `rtable` structures. Route.c includes a function `rt_cache_add`. Lines 1299-1385. The function `rt_cache_add` accesses the route table using a search key `hash`. See lines 1345 and 1356. The function `rt_cache_add` uses an external chaining technique to store records with the same hash address. See lines 1345, 1356, and 1357.

Asserted Claims from U.S. Pat. No. 5,893,120		Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("Linux 2.0.1") alone and in combination
		Each `rtable` structure includes an `rt_lastuse` field. Functions in route.c, such as the function `rt_cache_add`, use a record's `rt_lastuse` field to determine whether the record has automatically expired. See /include/net/route.h, line 74 and route.c, line 1369.
accessing the linked list of records,	accessing a linked list of records having same hash address,	Linux 2.0.1 discloses accessing the linked list of records. Linux 2.0.1 also discloses accessing a linked list of records having same hash address. For example, the hash table is an array of linked lists of `rtable` structures. The function `rt_cache_add` accesses the route table using a search key `hash`. See lines 1345 and 1356. The function `rt_cache_add` includes code that inserts a new record into the linked list. Lines 1356-1357. Inserting a record into a linked list is one possible way to access the linked list.
identifying at least some of the automatically expired ones of the records, and	identifying at least some of the automatically expired ones of the records,	Linux 2.0.1 discloses identifying at least some of the automatically expired ones of the records. For example, the function `rt_cache_add` iterates through the linked list to identifying "automatically expired" records in the linked list. Lines 1365-1383.
removing at least some of the automatically expired	removing at least some of the automatically expired	Linux 2.0.1 discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed.

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
records from the linked list when the linked list is accessed.	records from the linked list when the linked list is accessed, and	For example, the function rt_cache_add includes a while loop. Lines 1365-1383. The while loop iterates through a linked list to remove "automatically expired" records. Specifically, line 1369 determines whether the record has expired when more than a particular amount of time has passed since the record was last used. Also, LINUX 2.0.1 removes expired records from the linked list, as claimed. For example, line 1372 adjusts the pointers to de-link the expired record from the linked list. And line 1378 invokes the rt_free function. The rt_free function frees the memory allocated to the expired record. See lines 881-905. The function rt_cache_add includes code that inserts a new record into the linked list. Lines 1356-1357. Inserting a record into a linked list is one possible way to access the linked list. The code to insert the new record into the linked list and the while loop are executed in the same invocation of the function rt_cache_add. Hence, the function rt_cache_add includes means (i.e., the while loop) for removing expired records from a linked list when the function rt_cache_add inserts a new record into (i.e., accesses) the linked list.
	inserting, retrieving or deleting one of the records	LINUX 2.0.1 discloses inserting, retrieving or deleting one of the records from the system following the step of removing.

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
	from the system following the step of removing.	For example, the loop beginning at line 1365 iterates through the records in the previously-accessed linked list, and line 1369 identifies whether a particular record has expired. Depending on claim construction, line 1372 and/or line 1378 removes the expired record from the linked list, and line 894 deletes the expired record from memory. Further, the while loop of lines 1365 to 1383 is checking for duplicate records at the same time that it is checking for automatically expired records. See lines 1370 and 1378. The while loop may delete a duplicate record following the removal of at least one of the automatically expired records from the linked list because the while loop of lines 1365 to 1383 does not break after an automatically expired record is removed.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	LINUX 2.0.1 discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. Line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. Line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`.

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. See lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. See lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. See lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. Line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. Line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. See lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. See lines 1120-1131. For each record in a linked list, the function

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz (“LINUX 2.0.1”) alone and in combination
		`rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. See line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. See lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. Line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. Line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. See line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock’s proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are “expired” records. That is, the records removed by the function `rt_garbage_collect_1` are data items which after a limited time or after the occurrence of some event become

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. Line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. Line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list. Thus, by determining at line 1341 whether to invoke the function `rt_garbage_collect`, the function `rt_cache_add` dynamically determines the maximum number of records that can be removed from a linked list in the hash table.

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		Bedrock's proposed claim constructions assert that this element corresponds to portions of application software, user access software, or operating system software… that perform the function of determining, based upon one or more factors existing at the time the record search means is invoked, maximum number of record for the search means to remove in the linked list of records. Furthermore, Bedrock points to col. 6, line 56 – col. 7, line 15 of the '120 Patent which "describe code that chooses among removal options at the time the record search means is invoked by the caller, thus sometimes removing all expired records, at other times removing some but not all of them, and yet at other times choosing to remove none of them, or the equivalent thereof." (Emphasis added). The total number of records in the hash table is a factor existing at the time the function `rt_cache_add` is invoked. See lines 1299-1340 (not changing the variable `rt_cache_size`). By determining at line 1341 whether to invoke the function `rt_garbage_collect`, the function `rt_cache_add` chooses among removal options at the time the function `rt_cache_add` is invoked by a caller of the function `rt_cache_add`. The different removal options (i.e., using the function `rt_garbage_collect_1` or not using the function `rt_garbage_collect_1`) can remove different numbers of records from a linked list in the hash table. Further, Linux 2.0.1 combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress

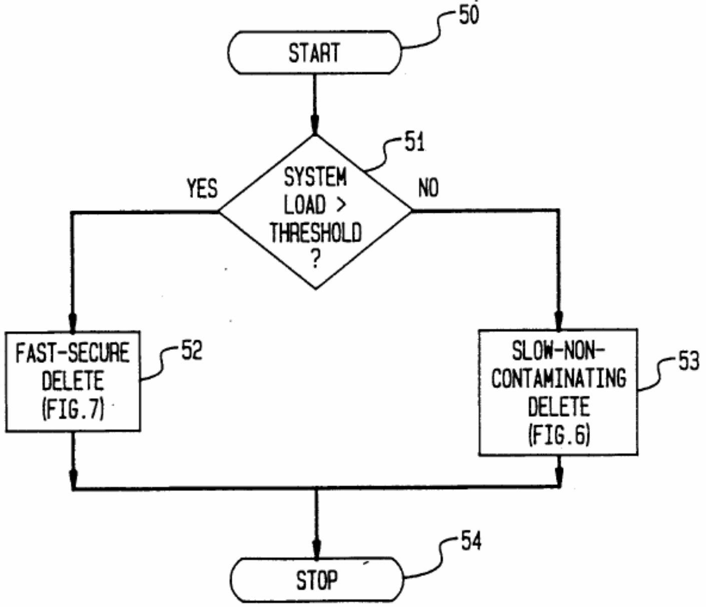
Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz (“LINUX 2.0.1”) alone and in combination
		(Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT$_i$ on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that

Asserted Claims from U.S. Pat. No. 5,893,120		**Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("Linux 2.0.1") alone and in combination**
		all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both Linux 2.0.1 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as Linux 2.0.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with Linux 2.0.1 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with Linux 2.0.1 and

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in Linux 2.0.1 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both Linux 2.0.1 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining Linux 2.0.1 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine Linux 2.0.1 with Thatte and recognize the benefits of doing so. For

Asserted Claims from U.S. Pat. No. 5,893,120		**Linux 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz (“Linux 2.0.1”) alone and in combination**
		example, the removal of expired records described in Linux 2.0.1 can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining Linux 2.0.1 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine Linux 2.0.1 with Thatte. Alternatively, it would also be obvious to combine Linux 2.0.1 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz (“LINUX 2.0.1”) alone and in combination**
		than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both Linux 2.0.1 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in Linux 2.0.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		patent's deletion decision procedure with Linux 2.0.1 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with Linux 2.0.1 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine Linux 2.0.1 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is

Asserted Claims from U.S. Pat. No. 5,893,120		**LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination**
		to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32.

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		As both Linux 2.0.1 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as Linux 2.0.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with Linux 2.0.1 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with Linux 2.0.1 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in Linux 2.0.1 to dynamically determine the maximum number of expired records to remove in the accessed linked list

Asserted Claims from U.S. Pat. No. 5,893,120		LINUX 2.0.1 net\ipv4\route.c, available at http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.1.tar.gz ("LINUX 2.0.1") alone and in combination
		of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in Linux 2.0.1 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in Linux 2.0.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15.

EXHIBIT D-13

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, GCC 2.7.2.1 discloses an information storage and retrieval system. For example, GCC 2.7.2.1 includes a file "alloca.c". "alloca.c" is "an implementation of the PWB library alloca function, which is used to allocate space off the run-time stack so that it is automatically reclaimed upon procedure exit…" alloca.c, lines 4-6. The run-time stack is an information storage and retrieval system.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	GCC 2.7.2.1 discloses a linked list to store and provide access to records stored in a memory of the system. GCC 2.7.2.1 also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, GCC 2.7.2.1 includes a `header` structure. alloca, lines 142-150. The `header` structure includes a pointer to a next `header` structure. alloca.c, line 147. The pointer in the `header` structure is used to chain together (i.e., form a linked list of) `header` structures. See alloca.c, lines 131-132. GCC 2.7.2.1 discloses that at least some of the blocks automatically expire. For example, GCC 2.7.2.1 provides that "garbage" is reclaimed. In GCC

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		2.7.2.1, "garbage" is "defined as all alloca'd storage that was allocated from deeper in the stack than currently (sic)." alloca.c, lines 173-174. Bedrock's proposed construction of the term "automatically expiring" is "data items which, after a limited period of time or after the occurrence of some event, become obsolete, such that their presence is no longer needed or desired." Under Bedrock's proposed construction, "alloca'd storage" can become "expired" if the "alloca'd storage" is no longer needed or desired after the occurrence of some event. In GCC 2.7.2.1, "alloca'd storage" is no longer needed or desired after the "alloca'd storage" becomes deeper in the stack that the current depth. GCC 2.7.2.1 includes a file "genattrtab.c." Genattrtab.c provides for "a hash table for sharing RTL and strings." See genattrtab.c, lines 456 and 481. Genattrtab.c further provides that each hash table slot is a bucket containing a chain of `attr_hash` structures. See genattrtab.c, line 458 and lines 462-471. Each `attr_hash` structure includes a pointer to a next `attr_hash` structure in a bucket. genattrtab.c, line 464. Genattrtab.c further provides for using an external chaining technique to store the records with same hash address. See, e.g., genattrtab.c, line 500. It would have been obvious to one of skill in the art to combine genattrtab.c with alloca.c. For example, both source files are within the GCC software package and disclose using a known technique to a known system to yield a predictable result.

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination**
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	GCC 2.7.2.1 discloses a record search means utilizing a search key to access the linked list. GCC 2.7.2.1 includes a file "genattrtab.c." Genattrtab.c provides for "a hash table for sharing RTL and strings." See genattrtab.c, lines 456 and 481. Genattrtab.c further provides that each hash table slot is a bucket containing a chain of `attr_hash` structures. See genattrtab.c, line 458 and lines 462-471. Each `attr_hash` structure includes a pointer to a next `attr_hash` structure in a bucket. genattrtab.c, line 464. Genattrtab.c further provides for using a hash code to access a linked list in the hash table. genattrtab.c, line 500. Under Bedrock's proposed construction, the "record search means" is "corresponding portions of the application software, user access software or operating system software… that perform the function of record searching utilizing a search key to access the linked list." Under Bedrock's proposed construction, the "search key" can be a hash code. Thus, GCC 2.7.2.1 discloses utilizing a record search means (i.e., software in the genattrtab.c file) that utilizes a search key (i.e., the hash code) to access a linked list of records (i.e., the chain of `attr_hash` structures) having the same hash address (i.e., being in the same bucket of the hash table).

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination**
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	CGG 2.7.21 discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. For example, the alloca.c file in GCC 2.7.2.1 includes a function `alloca`. When invoked, the function `alloca` creates a new `header` structure and adds the new `header` structure to a linked list of `header` structures. alloca.c, lines 206-215. In this way, the function `alloca` accesses the linked list of `header` structures. When invoked, the function `alloca` also performs a loop. The loop scans through the linked list of `header` structures to identify and remove “expired” `header` structures from the linked list of `header` structures. alloca.c, lines 183-201. As described above, `header` structures can be considered to be “expired” when the `header` structures have depths that are greater than the current depth. Hence, GCC 2.7.2.1 discloses a means (i.e., the loop) for identifying and removing at least some of the expired ones of the records (i.e., the `header` structures having depths that are greater than the current depth) from the linked list (i.e., the linked list of `header` structures) when the linked list is accessed. GCC 2.7.2.1 does not disclose the function `alloca` uses a search key to access the linked list of `header` structures. As described above, the portion of GCC 2.7.2.1 in genattrtab.c discloses the use of a search key to access a

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		linked list of attr_hash structures. Thus, one could take the concept of using of a search key to access a linked list, as taught in the genattrtab.c portion of GCC 2.7.2.1, and place it into the function alloca in the alloca.c portion of GCC 2.7.2.1. The result would be a record search means that utilizes a search key to access the linked list and that includes a means for identifying and removing at least some of the expired ones of the records form the linked list when the linked list is accessed.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] meals[sic], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	GCC 2.7.2.1 discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. GCC 2.7.2.1 also discloses meals[sic], utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. As described above, the function alloca utilizes the loop (i.e., a means for identifying and removing at least some of the expired ones of the records in the linked list). This loop could easily be modified to access the linked list from a hash table. The function alloca also inserts header structures into the linked list. alloca.c, lines 212-213. Inserting the header structure into the linked list is one possible way of accessing the linked list. When invoked, the function alloca inserts the header structure into the linked list and also executes the loop. Hence, the function alloca is a

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		means, utilizing the loop, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records from the linked list. The function `alloca` could be augmented with the teachings of LINUX 2.0.1 to derive a means that utilizes the loop, inserts records into the linked list, and also retrieves and deletes records from the linked list. LINUX 2.0.1 includes a file route.c. The file route.c includes a function `rt_cache_add`. The function `rt_cache_add` retrieves records from a linked list. See route.c, lines 1347-1354 (looping through and printing the `rt_dst` element of each record in the linked list). One could easily adapt the loop described in lines 1347-1354 of route.c for use in the function `alloca`. Debugging the function `alloca` might be one possible motivation for adapting the loop described in lines 1347-1354 of route.c for use in the function `alloca`. This example motivation is provided in line 1346 of route.c ("`#if RT_CACHE_DEBUG >= 2`"). The function `rt_cache_add` also deletes records from a linked list. The function `rt_cache_add` invokes the function `rt_garbage_collect`. route.c, line 1432. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. route.c, line 1293. The function `rt_garbage_collect_1` removes records from a linked list. See route.c, lines 1122-1138. The function `rt_garbage_collect_1` can remove expired records in the linked list plus other records in the linked list. See route.c, lines 1122-1138. Hence, by invoking the function `rt_garbage_collect`, the function `rt_cache_add` deletes records and

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		removes expired records. One could add this functionality to the function `alloca` to remove other records from the linked list that are not in use. To the extent that GCC 2.7.2.1 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCC 2.7.2.1 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since GCC 2.7.2.1 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of GCC 2.7.2.1 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache,

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining GCC 2.7.2.1 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: “*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). “*Calookup*() searches for a cached entry matching the key passed as an argument.” *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). “*Caremove*() removes the cached entry whose key is given, if one exists.” *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove().

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination

		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination

		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1, further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5, further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	GCC 2.7.2.1 combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach

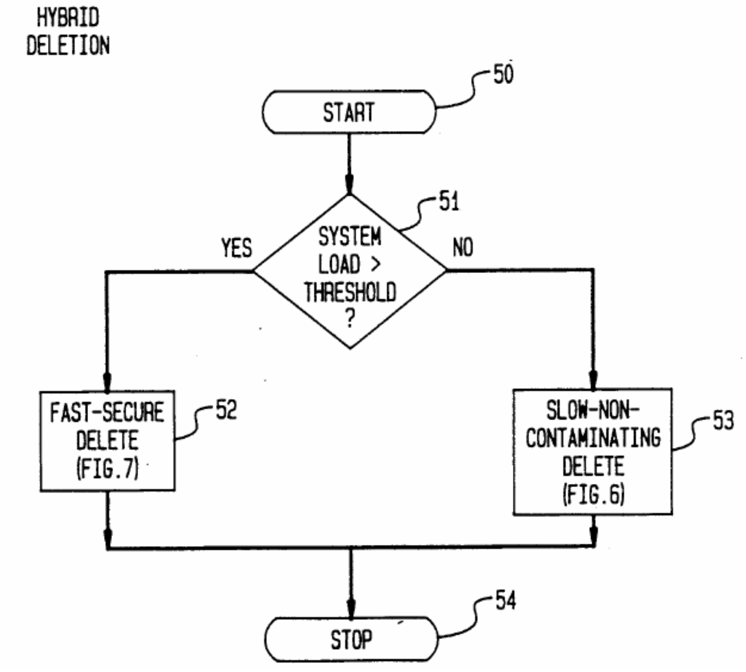
Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks:

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination**
		Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both GCC 2.7.2.1 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as GCC 2.7.2.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with GCC 2.7.2.1 nothing more than the predictable use

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with GCC 2.7.2.1 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCC 2.7.2.1 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both GCC 2.7.2.1 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining GCC 2.7.2.1 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine GCC 2.7.2.1 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in GCC 2.7.2.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining GCC 2.7.2.1 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine GCC 2.7.2.1 with Thatte. Alternatively, it would also be obvious to combine GCC 2.7.2.1 with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.*

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both GCC 2.7.2.1 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in GCC 2.7.2.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with GCC 2.7.2.1 would be nothing more than the predictable use of prior art elements according to their

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with GCC 2.7.2.1 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine GCC 2.7.2.1 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both GCC 2.7.2.1 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		in other hash table implementations such as GCC 2.7.2.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with GCC 2.7.2.1 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with GCC 2.7.2.1 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in GCC 2.7.2.1 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in GCC 2.7.2.1 with the fundamental

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in GCC 2.7.2.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by GCC 2.7.2.1 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with GCC 2.7.2.1. For example, both Linux 2.0.1 and GCC 2.7.2.1 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination**
		result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1`

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list. GCC 2.7.2.1
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least	To the extent the preamble is a limitation, GCC 2.7.2.1 discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. To the extent the preamble is a limitation, GCC 2.7.2.1d also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
steps of:	some of the records automatically expiring, the method comprising the steps of:	For example, GCC 2.7.2.1 includes a file "alloca.c". "alloca.c" is "an implementation of the PWB library alloca function, which is used to allocate space off the run-time stack so that it is automatically reclaimed upon procedure exit…" alloca.c, lines 4-6. The run-time stack is an information storage and retrieval system. GCC 2.7.2.1 discloses a linked list to store and provide access to records stored in a memory of the system. For example, GCC 2.7.2.1 includes a `header` structure. alloca, lines 142-150. The `header` structure includes a pointer to a next `header` structure. alloca.c, line 147. The pointer in the `header` structure is used to chain together (i.e., form a linked list of) `header` structures. See alloca.c, lines 131-132. GCC 2.7.2.1 discloses that at least some of the blocks automatically expire. For example, GCC 2.7.2.1 provides that "garbage" is reclaimed. In GCC 2.7.2.1, "garbage" is "defined as all alloca'd storage that was allocated from deeper in the stack than currently." alloca.c, lines 173-174. Bedrock's proposed construction of the term "automatically expiring" is "data items which, after a limited period of time or after the occurrence of some event, become obsolete, such that their presence is no longer needed or desired."

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		Under Bedrock's proposed construction, "alloca'd storage" can become "expired" if the "alloca'd storage" is no longer needed or desired after the occurrence of some event. In GCC 2.7.2.1, "alloca'd storage" is no longer needed or desired after the "alloca'd storage" becomes deeper in the stack that the current depth. GCC 2.7.2.1 includes a file "genattrtab.c." Genattrtab.c provides for "a hash table for sharing RTL and strings." See genattrtab.c, lines 456 and 481. Genattrtab.c further provides that each hash table slot is a bucket containing a chain of `attr_hash` structures. See genattrtab.c, line 458 and lines 462-471. Each `attr_hash` structure includes a pointer to a next `attr_hash` structure in a bucket. genattrtab.c, line 464. Genattrtab.c further provides for using an external chaining technique to store the records with same hash address. See, e.g., genattrtab.c, line 500. It would have been obvious to one of skill in the art to combine genattrtab.c with alloca.c. For example, both source files are within the GCC software package and disclose using a known technique to a known system to yield a predictable result.
accessing the linked list of records,	accessing a linked list of records having same hash address,	GCC 2.7.2.1 discloses accessing the linked list of records. GCC 2.7.2.1 also discloses accessing a linked list of records having same hash address. For example, alloca.c includes a function `alloca`. alloca.c, lines 162-221. The function `alloca` adds a `header` structure to a linked list of `header` structures. alloca.c, lines 212-213. Adding a `header` structure to a linked

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		list is one way of accessing the linked list. Bedrock's proposed claim constructions provide that "means for accessing the linked list" correspond to software which provides the **insert**, retrieve, or delete record capability illustrated in the flowchart of FIG. 5, FIG. 6, or FIG. 7…" (emphasis added). Hence, under Bedrock's proposed claim constructions, the function `alloca` is software that "accesses" a linked list. GCC 2.7.2.1 includes a file "genattrtab.c." Genattrtab.c provides for "a hash table for sharing RTL and strings." See genattrtab.c, lines 456 and 481. Genattrtab.c further provides that each hash table slot is a bucket containing a chain of `attr_hash` structures. See genattrtab.c, line 458 and lines 462-471. Each `attr_hash` structure includes a pointer to a next `attr_hash` structure in a bucket. genattrtab.c, line 464. Genattrtab.c further provides for using an external chaining technique to store the records with same hash address. See, e.g., genattrtab.c, line 500. It would have been obvious to one of skill in the art to combine genattrtab.c with alloca.c. For example, both source files are within the GCC software package and disclose using a known technique to a known system to yield a predictable result.
identifying at least some of the automatically expired ones of the records, and	identifying at least some of the automatically expired ones of the records,	GCC 2.7.2.1 identifies at least some of the automatically expired ones of the records. When invoked, the function `alloca` performs a loop. The loop scans through the linked list of `header` structures to identify "expired" `header`

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		structures from the linked list of header structures. alloca.c, lines 183-201. As described above, header structures can be considered to be "expired" when the header structures have depths that are greater than the current depth.
removing at least some of the automatically expired records from the linked list when the linked list is accessed.	removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	GCC 2.7.2.1 removes at least some of the automatically expired records from the linked list when the linked list is accessed. As mentioned above, the function alloca performs a loop. The loop scans through the linked list of header structures to remove "expired" header structures from the linked list of header structures. alloca.c, lines 183-201. Bedrock's proposed claim construction provides that at least some of the expired ones of the records from the linked list when the linked list is accessed for a purpose other than garbage collection. The function alloca performs the loop when invoked. The function alloca does not insert a header structure in the linked list of header structures for the purpose of garbage collection. Rather, the function alloca is used to allocate space off the run-time stack so that it is automatically reclaimed upon procedure exit. See alloca.c, lines 4-6.
	inserting, retrieving or deleting one of the records from the system following the step of removing.	GCC 2.7.2.1 discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, the function alloca removes header structures from the linked list in lines 183-194.

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		The function `alloca` inserts a `header` structure into the linked list in lines 212-213. Hence, the function `alloca` inserts the `header` structure into the linked list following the step of removing `header` structures from the linked list.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	GCC 2.7.2.1 combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in

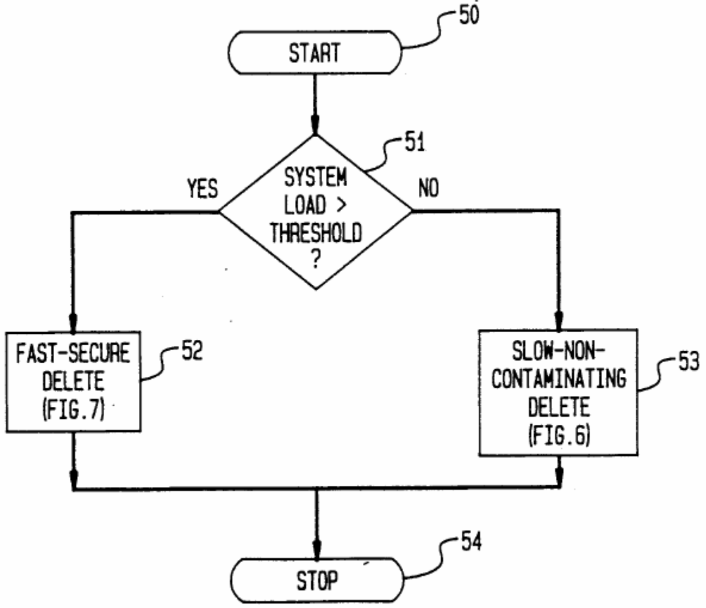
Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination**
		the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30.

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both GCC 2.7.2.1 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as GCC 2.7.2.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with GCC 2.7.2.1 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with GCC 2.7.2.1 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in GCC 2.7.2.1 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both GCC 2.7.2.1 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining GCC 2.7.2.1 with Thatte would be nothing

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine GCC 2.7.2.1 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in GCC 2.7.2.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining GCC 2.7.2.1 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine GCC 2.7.2.1 with Thatte. Alternatively, it would also be obvious to combine GCC 2.7.2.1 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination**
		involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.*

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both GCC 2.7.2.1 and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in GCC 2.7.2.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with GCC 2.7.2.1 would be nothing more than the predictable use of prior art elements according to their

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with GCC 2.7.2.1 and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine GCC 2.7.2.1 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination**
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both GCC 2.7.2.1 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		in other hash table implementations such as GCC 2.7.2.1. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with GCC 2.7.2.1 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with GCC 2.7.2.1 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in GCC 2.7.2.1 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in GCC 2.7.2.1 with the fundamental

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in GCC 2.7.2.1 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by GCC 2.7.2.1 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with GCC 2.7.2.1. For example, both Linux 2.0.1 and GCC 2.7.2.1 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1`

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ (“GCC 2.7.2.1”) alone and in combination
		accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record’s expiration factor is based on a variable `expire` and the record’s reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at

Asserted Claims from U.S. Pat. No. 5,893,120		**GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination**
		line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the

Asserted Claims from U.S. Pat. No. 5,893,120		GCC 2.7.2.1, available at http://gcc-uk.internet.bs/old-releases/gcc-2/ ("GCC 2.7.2.1") alone and in combination
		maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, MK84 discloses an information storage and retrieval system. For example, MK84 discloses a hash table with queues, which are doubly-linked lists, of automatically-expiring records. *See, e.g.*, net_io.c, at 479-87: 479 struct net_hash_entry { 480 queue_chain_t chain; /* list of entries with same hval */ 481 #define he_next chain.next 482 #define he_prev chain.prev 483 ipc_port_t rcv_port; /* destination port */ 484 int rcv_qlimit; /* qlimit for the port */ 485 unsigned int keys[N_NET_HASH_KEYS]; 486 }; 487 typedef struct net_hash_entry *net_hash_entry_t;
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	MK84 discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. MK84 also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, MK84 discloses using a queue, which is a doubly-linked list, as well as an external chaining technique to store the records with same hash address. *See, e.g.*, net_io.c, at 479-87: 479 struct net_hash_entry { 480 queue_chain_t chain; /* list of entries with same hval */ 481 #define he_next chain.next

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**

		482 #define he_prev chain.prev
		483 ipc_port_t rcv_port; /* destination port */
		484 int rcv_qlimit; /* qlimit for the port */
		485 unsigned int keys[N_NET_HASH_KEYS];
		486 };
		487 typedef struct net_hash_entry *net_hash_entry_t;
		
		MK84 also discloses records automatically expiring. For example, the net_filter() function in net_io.c deals with records corresponding to filters that have died. *See, e.g.*, net_io.c, at 969-89:
		
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {
		975 queue_remove(&ifp->if_rcv_port_list, infp,
		976 net_rcv_port_t, chain);
		977 ENQUEUE_DEAD(dead_infp, infp);
		978 continue;
		979 } else {
		980 (void) hash_ent_remove (
		981 ifp,
		982 (net_hash_header_t)infp,
		983 FALSE, /* no longer used */
		984 hash_headp,
		985 entp,
		986 &dead_entp);

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**

		987 continue;
		988 }
		989 }

[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	MK84 discloses a record search means utilizing a search key to access the linked list. MK84 also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.
		For example, MK84 includes functionality to use a pointer to traverse a linked list having the same hash address. Examples of utilizing a search key to access the linked list and utilizing a search key to access a linked list of records having the same address can be found at the queue_remove(), hash_ent_remove(), and remqueue() calls. *See, e.g.*, net_io.c, at 969-89:
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {
		975 queue_remove(&ifp->if_rcv_port_list, infp,
		976 net_rcv_port_t, chain);
		977 ENQUEUE_DEAD(dead_infp, infp);
		978 continue;
		979 } else {
		980 (void) hash_ent_remove (
		981 ifp,
		982 (net_hash_header_t)infp,
		983 FALSE, /* no longer used */
		984 hash_headp,

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		985 entp, 986 &dead_entp); 987 continue; 988 } 989 } The queue_remove() macro is defined at lines 320-44 in queue.h: 320 /* 321 * Macro: queue_remove 322 * Function: 323 * Remove an arbitrary item from the queue. 324 * Header: 325 * void queue_remove(q, qe, type, field) 326 * arguments as in queue_enter 327 */ 328 #define queue_remove(head, elt, type, field) \\ 329 MACRO_BEGIN \\ 330 register queue_entry_t next, prev; \\ 331 \\ 332 next = (elt)->field.next; \\ 333 prev = (elt)->field.prev; \\ 334 \\ 335 if ((head) == next) \\ 336 (head)->prev = prev; \\ 337 else \\ 338 ((type)next)->field.prev = prev; \\ 339 \\ 340 if ((head) == prev) \\

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		341 (head)->next = next; \\
		342 else \\
		343 ((type)prev)->field.next = next; \\
		344 MACRO_END
		
		The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:
		
		2216 /*
		2217 * Removes a hash entry (ENTP) from its queue (HEAD).
		2218 * If the reference count of filter (HP) becomes zero and not USED,
		2219 * HP is removed from ifp->if_rcv_port_list and is freed.
		2220 */
		2221
		2222 boolean_t
		2223 hash_ent_remove (
		2224 struct ifnet *ifp,
		2225 net_hash_header_t hp,
		2226 int used,
		2227 net_hash_entry_t *head,
		2228 net_hash_entry_t entp,
		2229 queue_entry_t *dead_p)
		2230 {
		2231 hp->ref_count--;
		2232
		2233 if (*head == entp) {
		2234
		2235 if (queue_empty((queue_t) entp)) {
		2236 *head = 0;
		2237 ENQUEUE_DEAD(*dead_p, entp);

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		2238 if (hp->ref_count == 0 && !used) {
		2239 remqueue((queue_t) &ifp->if_rcv_port_list,
		2240 (queue_entry_t)hp);
		2241 hp->n_keys = 0;
		2242 return TRUE;
		2243 }
		2244 return FALSE;
		2245 } else {
		2246 *head = (net_hash_entry_t)queue_next((queue_t) entp);
		2247 }
		2248 }
		2249
		2250 remqueue((queue_t)*head, (queue_entry_t)entp);
		2251 ENQUEUE_DEAD(*dead_p, entp);
		2252 return FALSE;
		2253 }
		
		The remqueue() function is defined in queue.c at lines 139-45:
		
		132 /*
		133 * Remove arbitrary element from queue.
		134 * Does not check whether element is on queue - the world
		135 * will go haywire if it isn't.
		136 */
		137
		138 /*ARGSUSED*/
		139 void remqueue(
		140 queue_t que,
		141 register queue_entry_t elt)

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**

		142 {
		143 elt->next->prev = elt->prev;
		144 elt->prev->next = elt->next;
		145 }

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	MK84 discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. MK84 also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed.
		
		For example, MK84 includes the functionality to identify and remove expired records from the linked list when the linked list is accessed. For example, in the net_filter() function in net_io.c, the linked list is accessed for a purpose other than garbage collection. *See, e.g.*, net_io.c, at 942-67:
		
		942 FILTER_ITERATE(ifp, infp, nextfp)
		943 {
		944 entp = (net_hash_entry_t) 0;
		945 if (infp->filter[0] == NETF_BPF) {
		946 ret_count = bpf_do_filter(infp, net_kmsg(kmsg)->packet, count,
		947 net_kmsg(kmsg)->header,
		948 &hash_headp, &entp);
		949 if (entp == (net_hash_entry_t) 0)
		950 dest = infp->rcv_port;
		951 else
		952 dest = entp->rcv_port;
		953 } else {
		954 ret_count = net_do_filter(infp, net_kmsg(kmsg)->packet,

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		count, 955 net_kmsg(kmsg)->header); 956 if (ret_count) 957 ret_count = count; 958 dest = infp->rcv_port; 959 } 960 961 if (ret_count) { 962 963 /* 964 * Make a send right for the destination. 965 */ 966 967 dest = ipc_port_copy_send(dest); The FILTER_ITERATE() macro used at line 942 is defined in net_io.c at lines 516-21: 516 #define FILTER_ITERATE(ifp, fp, nextfp) \\ 517 for ((fp) = (net_rcv_port_t) queue_first(&(ifp)->if_rcv_port_list);\\ 518 !queue_end(&(ifp)->if_rcv_port_list, (queue_entry_t)(fp)); \\ 519 (fp) = (nextfp)) { \\ 520 (nextfp) = (net_rcv_port_t) queue_next(&(fp)->chain); 521 #define FILTER_ITERATE_END } Also, at line 946, the bpf_do_filter() function is called. The bpf_do_filter() function is in net_io.c at 1760-2064. This function performs various operations, depending on the code associated with the packet. One example of is found at lines 1817-23:

EXHIBIT D-14

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>net_filter() in net_io.c from version MK84 of the Mach kernel (1993)
(hereinafter “MK84”) alone and in combination</th></tr>
<tr><td></td><td></td><td>

```
1817            case BPF_RET|BPF_MATCH_IMM:
1818                if (bpf_match ((net_hash_header_t)infp, pc->jt, mem,
1819                        hash_headpp, entpp)) {
1820                    return ((unsigned int)pc->k <= wirelen) ?
1821                                pc->k : wirelen;
1822                }
1823                return 0;
```

At line 1818, the bpf_match() function is called. The bpf_match() function iterates through the hash table and associated linked lists searching for a match. Thus, this is an example of accessing the linked list for a purpose other than garbage collection. *See, e.g.*, net_io.c at 2180-2213:

```
2180 boolean_t
2181 bpf_match (
2182        net_hash_header_t hash,
2183        register int n_keys,
2184        register unsigned int *keys,
2185        net_hash_entry_t **hash_headpp,
2186        net_hash_entry_t *entpp)
2187 {
2188        register net_hash_entry_t head, entp;
2189        register int i;
2190
2191        if (n_keys != hash->n_keys)
2192            return FALSE;
2193
2194        *hash_headpp = &hash->table[bpf_hash(n_keys, keys)];
```

</td></tr>
</table>

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		2195 head = **hash_headpp;
		2196
		2197 if (head == 0)
		2198 return FALSE;
		2199
		2200 HASH_ITERATE (head, entp)
		2201 {
		2202 for (i = 0; i < n_keys; i++) {
		2203 if (keys[i] != entp->keys[i])
		2204 break;
		2205 }
		2206 if (i == n_keys) {
		2207 *entpp = entp;
		2208 return TRUE;
		2209 }
		2210 }
		2211 HASH_ITERATE_END (head, entp)
		2212 return FALSE;
		2213 }
		
		The HASH_ITERATE() and HASH_ITERATE_END() macros are defined at lines 510-513 of net_io.c:
		
		510 #define HASH_ITERATE(head, elt) (elt) = (net_hash_entry_t) (head); do {
		511 #define HASH_ITERATE_END(head, elt) \
		512 (elt) = (net_hash_entry_t) queue_next((queue_entry_t) (elt)); \
		513 } while ((elt) != (head));

EXHIBIT D-14

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>net_filter() in net_io.c from version MK84 of the Mach kernel (1993)
(hereinafter “MK84”) alone and in combination</th></tr>
<tr><td></td><td></td><td>

As shown in the example above, MK84 accesses the linked list of records. MK84 also identifies and removes expired ones of the records when the linked list is accessed. An example of this is in the net_filter() function in MK84. If the filter is dead, then the record in the linked list associated with that record is removed. *See, e.g.*, net_io.c at 961-89:

```
961         if (ret_count) {
962
963             /*
964              * Make a send right for the destination.
965              */
966
967             dest = ipc_port_copy_send(dest);
968             if (!IP_VALID(dest)) {
969                 /*
970                  * This filter is dead.  We remove it from the
971                  * filter list and set it aside for deallocation.
972                  */
973
974                 if (entp == (net_hash_entry_t) 0) {
975                     queue_remove(&ifp->if_rcv_port_list, infp,
976                                 net_rcv_port_t, chain);
977                     ENQUEUE_DEAD(dead_infp, infp);
978                     continue;
979                 } else {
980                     (void) hash_ent_remove (
981                                     ifp,
982                                     (net_hash_header_t)infp,
983                                     FALSE,        /* no longer used */
```

</td></tr>
</table>

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		984 hash_headp, 985 entp, 986 &dead_entp); 987 continue; 988 } 989 } MK84 also discloses deallocating the memory used by the expired records. *See, e.g.*, net_io.c at 1058-64: 1058 /* 1059 * Deallocate dead filters. 1060 */ 1061 if (dead_infp != 0) 1062 net_free_dead_infp(dead_infp); 1063 if (dead_entp != 0) 1064 net_free_dead_entp(dead_entp);
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	MK84 discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. MK84 also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. The “means, utilizing the record search means . . .” limitation is met, for example, by any function calling the net_filter() function. For example, net_deliver() calls net_filter() at net_io.c, line 646: 642 /*

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		643 * Run the packet through the filters,
		644 * getting back a queue of packets to send.
		645 */
		646 net_filter(kmsg, &send_list);
		
		Further, depending on claim construction, lines 2239-40 in net_io.c provide examples of the "deleting" and/or "removing" limitations. This code is called from line 980 in net_io.c in the net_filter() function. An example of the "retrieving" step is line 942. These operations take place within a single function and "at the same time," as claimed.
		
		2233 if (*head == entp) {
		2234
		2235 if (queue_empty((queue_t) entp)) {
		2236 *head = 0;
		2237 ENQUEUE_DEAD(*dead_p, entp);
		2238 if (hp->ref_count == 0 && !used) {
		2239 remqueue((queue_t) &ifp->if_rcv_port_list,
		2240 (queue_entry_t)hp);
		2241 hp->n_keys = 0;
		2242 return TRUE;
		2243 }
		2244 return FALSE;
		2245 } else {
		2246 *head = (net_hash_entry_t)queue_next((queue_t) entp);
		2247 }
		
		Further, it would be obvious for one of skill in the art to modify this code to include a means for inserting records at the same time as removal. For

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		example, the net_set_filter() function in net_io.c includes code for inserting records into the linked list. One of skill in the art would have known to combine this known technique disclosed in the same source code file to a known system, such as the net_filter() function in the MK84 kernel disclosed herein, in order to implement the claim limitation. *See, e.g.*, net_io.c at 1477-1492: 1477 /* Insert my_infp according to priority */ 1478 queue_iterate(&ifp->if_rcv_port_list, infp, net_rcv_port_t, chain) 1479 if (priority > infp->priority) 1480 break; 1481 enqueue_tail((queue_t)&infp->chain, (queue_entry_t)my_infp); 1482 } 1483 1484 if (match != 0) 1485 { /* Insert to hash list */ 1486 net_hash_entry_t *p; 1487 1488 hash_entp->rcv_port = rcv_port; 1489 for (i = 0; i < match->jt; i++) /* match->jt is n_keys */ 1490 hash_entp->keys[i] = match[i+1].k; 1491 p = &((net_hash_header_t)my_infp)-> 1492 table[bpf_hash(match->jt, hash_entp->keys)]; To the extent that MK84 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for

EXHIBIT D-14

Asserted Claims From **U.S. Pat. No. 5,893,120**		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993)** **(hereinafter “MK84”) alone and in combination**
		accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in MK84 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since MK84 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of MK84 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining MK84 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: “*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4.

EXHIBIT D-14

Asserted Claims From **U.S. Pat. No. 5,893,120**		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993)** **(hereinafter "MK84") alone and in combination**

		See also, gcache.c at lines 246-304, defining cainsert().
		"*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4.
		See also, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex().
		"*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4.
		See also, gcache.c at lines 355-376, defining caremove().
		Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned."

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		See Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	6. The information storage and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	MK84 discloses a means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.
		
		For example, the net_filter() function in net_io.c determines whether to remove one or zero elements from the linked list of records. For example, the code at line 968 determines if the filter is dead. If so, then it is to be removed; but if the filter is not dead then it is not to be removed. *See, e.g.*, net_io.c at 968-89:
		
		968 if (!IP_VALID(dest)) {
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {
		975 queue_remove(&ifp->if_rcv_port_list, infp,
		976 net_rcv_port_t, chain);

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		ENQUEUE_DEAD(dead_infp, infp);
		continue;
		} else {
		(void) hash_ent_remove (
		ifp,
		(net_hash_header_t)infp,
		FALSE, /* no longer used */
		hash_headp,
		entp,
		&dead_entp);
		continue;
		}
		}
		
		Further, MK84 combined with Linux 2.0.1, Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.
		
		Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety.
		
		For example, as summarized in Dirks,
		
		each time a VSID is assigned from the free list to a new application or

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30.

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**
		Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both MK84 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with MK84 nothing more than the predictable use of prior art elements according to their established

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with MK84 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in MK84 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both MK84 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining MK84 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine MK84 with Thatte and recognize the benefits of doing so. For example, the removal

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		of expired records described in MK84 can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining MK84 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine MK84 with Thatte. Alternatively, it would also be obvious to combine MK84 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-14

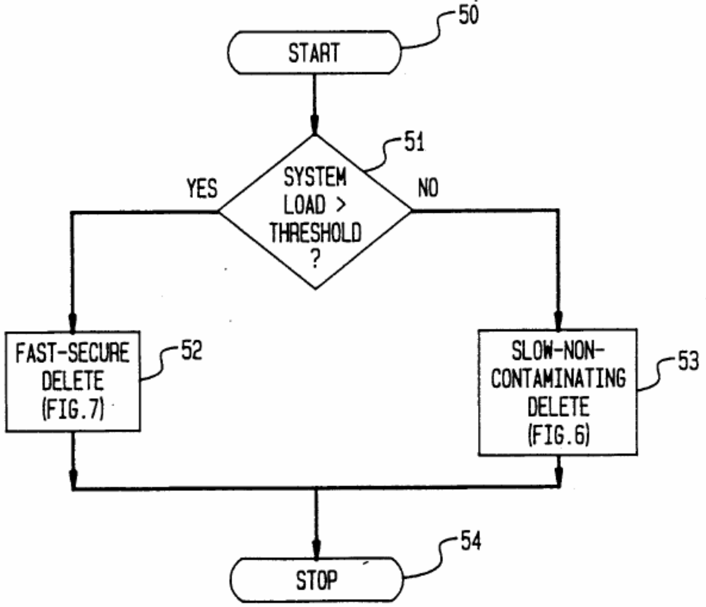
Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		FIG.5 HYBRID DELETION  50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both MK84 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with MK84 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with MK84 and

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine MK84 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both MK84 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with MK84 would be nothing more than the predictable use of prior

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with MK84 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in MK84 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in MK84 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by MK84 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with MK84. For example, both Linux 2.0.1 and MK84 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1`

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, MK84 discloses method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring as well as a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, MK84 discloses a hash table with external chaining using queues, which are doubly-linked lists, of automatically-expiring records. *See, e.g.*, net_io.c, at 479-87: 479 struct net_hash_entry {

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		480 queue_chain_t chain; /* list of entries with same hval */
		481 #define he_next chain.next
		482 #define he_prev chain.prev
		483 ipc_port_t rcv_port; /* destination port */
		484 int rcv_qlimit; /* qlimit for the port */
		485 unsigned int keys[N_NET_HASH_KEYS];
		486 };
		487 typedef struct net_hash_entry *net_hash_entry_t;
		
		MK84 also discloses records automatically expiring. For example, the net_filter() function in net_io.c deals with records corresponding to filters that have died. *See, e.g.*, net_io.c, at 969-89:
		
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {
		975 queue_remove(&ifp->if_rcv_port_list, infp,
		976 net_rcv_port_t, chain);
		977 ENQUEUE_DEAD(dead_infp, infp);
		978 continue;
		979 } else {
		980 (void) hash_ent_remove (
		981 ifp,
		982 (net_hash_header_t)infp,
		983 FALSE, /* no longer used */

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		984 hash_headp,
		985 entp,
		986 &dead_entp);
		987 continue;
		988 }
		989 }

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	MK84 discloses accessing the linked list of records. MK84 also discloses accessing a linked list of records having same hash address.
		
		For example, MK84 includes functionality to use a pointer to traverse a linked list having the same hash address. Examples of accessing a linked list of records and accessing a linked list of records having same hash address can be found at the queue_remove(), hash_ent_remove(), and remqueue() calls. *See, e.g.*, net_io.c, at 969-89:
		
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {
		975 queue_remove(&ifp->if_rcv_port_list, infp,
		976 net_rcv_port_t, chain);
		977 ENQUEUE_DEAD(dead_infp, infp);
		978 continue;
		979 } else {
		980 (void) hash_ent_remove (
		981 ifp,
		982 (net_hash_header_t)infp,

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
 983                                   FALSE,      /* no longer used */
 984                                   hash_headp,
 985                                   entp,
 986                                   &dead_entp);
 987                          continue;
 988                      }
 989                  }
```

The queue_remove() macro is defined at lines 320-44 in queue.h:

```
 320 /*
 321  *     Macro:        queue_remove
 322  *     Function:
 323  *          Remove an arbitrary item from the queue.
 324  *     Header:
 325  *          void queue_remove(q, qe, type, field)
 326  *               arguments as in queue_enter
 327  */
 328 #define queue_remove(head, elt, type, field)            \
 329     MACRO_BEGIN                                         \
 330         register queue_entry_t  next, prev;             \
 331                                                         \
 332         next = (elt)->field.next;                       \
 333         prev = (elt)->field.prev;                       \
 334                                                         \
 335         if ((head) == next)                             \
 336                 (head)->prev = prev;                    \
 337         else                                            \
 338                 ((type)next)->field.prev = prev;        \
```

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**

		339 \\
		340 if ((head) == prev) \\
		341 (head)->next = next; \\
		342 else \\
		343 ((type)prev)->field.next = next; \\
		344 MACRO_END
		
		The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:
		
		2216 /*
		2217 * Removes a hash entry (ENTP) from its queue (HEAD).
		2218 * If the reference count of filter (HP) becomes zero and not USED,
		2219 * HP is removed from ifp->if_rcv_port_list and is freed.
		2220 */
		2221
		2222 boolean_t
		2223 hash_ent_remove (
		2224 struct ifnet *ifp,
		2225 net_hash_header_t hp,
		2226 int used,
		2227 net_hash_entry_t *head,
		2228 net_hash_entry_t entp,
		2229 queue_entry_t *dead_p)
		2230 {
		2231 hp->ref_count--;
		2232
		2233 if (*head == entp) {
		2234
		2235 if (queue_empty((queue_t) entp)) {

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		2236 *head = 0;
		2237 ENQUEUE_DEAD(*dead_p, entp);
		2238 if (hp->ref_count == 0 && !used) {
		2239 remqueue((queue_t) &ifp->if_rcv_port_list,
		2240 (queue_entry_t)hp);
		2241 hp->n_keys = 0;
		2242 return TRUE;
		2243 }
		2244 return FALSE;
		2245 } else {
		2246 *head = (net_hash_entry_t)queue_next((queue_t) entp);
		2247 }
		2248 }
		2249
		2250 remqueue((queue_t)*head, (queue_entry_t)entp);
		2251 ENQUEUE_DEAD(*dead_p, entp);
		2252 return FALSE;
		2253 }
		
		The remqueue() function is defined in queue.c at lines 139-45:
		
		132 /*
		133 * Remove arbitrary element from queue.
		134 * Does not check whether element is on queue - the world
		135 * will go haywire if it isn't.
		136 */
		137
		138 /*ARGSUSED*/
		139 void remqueue(

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		140 queue_t que,
		141 register queue_entry_t elt)
		142 {
		143 elt->next->prev = elt->prev;
		144 elt->prev->next = elt->next;
		145 }

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	MK84 discloses identifying at least some of the automatically expired ones of the records.
		
		For example, MK84 accesses the linked list of records and identifies and removes expired ones of the records when the linked list is accessed. An example of this is in the net_filter() function in MK84. If the filter is dead, then the record in the linked list associated with that record is removed. *See*, *e.g.*, net_io.c at 961-89:
		
		961 if (ret_count) {
		962
		963 /*
		964 * Make a send right for the destination.
		965 */
		966
		967 dest = ipc_port_copy_send(dest);
		968 if (!IP_VALID(dest)) {
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		975 queue_remove(&ifp->if_rcv_port_list, infp,
		976 net_rcv_port_t, chain);
		977 ENQUEUE_DEAD(dead_infp, infp);
		978 continue;
		979 } else {
		980 (void) hash_ent_remove (
		981 ifp,
		982 (net_hash_header_t)infp,
		983 FALSE, /* no longer used */
		984 hash_headp,
		985 entp,
		986 &dead_entp);
		987 continue;
		988 }
		989 }
		
		Further, the functions called at lines 975 and 980 identify expired ones of the records.

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	MK84 discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed.
		
		For example, in the net_filter() function in net_io.c, the linked list is accessed for a purpose other than garbage collection. *See, e.g.*, net_io.c, at 942-67:
		
		942 FILTER_ITERATE(ifp, infp, nextfp)
		943 {
		944 entp = (net_hash_entry_t) 0;
		945 if (infp->filter[0] == NETF_BPF) {
		946 ret_count = bpf_do_filter(infp, net_kmsg(kmsg)->packet,

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**

		count,
		947 net_kmsg(kmsg)->header,
		948 &hash_headp, &entp);
		949 if (entp == (net_hash_entry_t) 0)
		950 dest = infp->rcv_port;
		951 else
		952 dest = entp->rcv_port;
		953 } else {
		954 ret_count = net_do_filter(infp, net_kmsg(kmsg)->packet,
		count,
		955 net_kmsg(kmsg)->header);
		956 if (ret_count)
		957 ret_count = count;
		958 dest = infp->rcv_port;
		959 }
		960
		961 if (ret_count) {
		962
		963 /*
		964 * Make a send right for the destination.
		965 */
		966
		967 dest = ipc_port_copy_send(dest);
		
		The FILTER_ITERATE() macro used at line 942 is defined in net_io.c at lines 516-21:
		
		516 #define FILTER_ITERATE(ifp, fp, nextfp) \\
		517 for ((fp) = (net_rcv_port_t) queue_first(&(ifp)->if_rcv_port_list);\\

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		518 !queue_end(&(ifp)->if_rcv_port_list, (queue_entry_t)(fp)); \\ 519 (fp) = (nextfp)) { \\ 520 (nextfp) = (net_rcv_port_t) queue_next(&(fp)->chain); 521 #define FILTER_ITERATE_END } Also, at line 946, the bpf_do_filter() function is called. The bpf_do_filter() function is in net_io.c at 1760-2064. This function performs various operations, depending on the code associated with the packet. One example of is found at lines 1817-23: 1817 case BPF_RET|BPF_MATCH_IMM: 1818 if (bpf_match ((net_hash_header_t)infp, pc->jt, mem, 1819 hash_headpp, entpp)) { 1820 return ((unsigned int)pc->k <= wirelen) ? 1821 pc->k : wirelen; 1822 } 1823 return 0; At line 1818, the bpf_match() function is called. The bpf_match() function iterates through the hash table and associated linked lists searching for a match. Thus, this is an example of accessing the linked list for a purpose other than garbage collection. *See, e.g.*, net_io.c at 2180-2213: 2180 boolean_t 2181 bpf_match (2182 net_hash_header_t hash, 2183 register int n_keys, 2184 register unsigned int *keys, 2185 net_hash_entry_t **hash_headpp,

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		net_hash_entry_t *entpp)
		{
		register net_hash_entry_t head, entp;
		register int i;
		
		if (n_keys != hash->n_keys)
		    return FALSE;
		
		*hash_headpp = &hash->table[bpf_hash(n_keys, keys)];
		head = **hash_headpp;
		
		if (head == 0)
		    return FALSE;
		
		HASH_ITERATE (head, entp)
		{
		    for (i = 0; i < n_keys; i++) {
		        if (keys[i] != entp->keys[i])
		            break;
		    }
		    if (i == n_keys) {
		        *entpp = entp;
		        return TRUE;
		    }
		}
		HASH_ITERATE_END (head, entp)
		return FALSE;
		}

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		The HASH_ITERATE() and HASH_ITERATE_END() macros are defined at lines 510-513 of net_io.c:
		
		510 #define HASH_ITERATE(head, elt) (elt) = (net_hash_entry_t) (head); do {
		511 #define HASH_ITERATE_END(head, elt) \\
		512 (elt) = (net_hash_entry_t) queue_next((queue_entry_t) (elt)); \\
		513 } while ((elt) != (head));
		
		As shown in the example above, MK84 accesses the linked list of records. MK84 also identifies and removes expired ones of the records when the linked list is accessed. An example of this is in the net_filter() function in MK84. If the filter is dead, then the record in the linked list associated with that record is removed. *See, e.g.*, net_io.c at 961-89:
		
		961 if (ret_count) {
		962
		963 /*
		964 * Make a send right for the destination.
		965 */
		966
		967 dest = ipc_port_copy_send(dest);
		968 if (!IP_VALID(dest)) {
		969 /*
		970 * This filter is dead. We remove it from the
		971 * filter list and set it aside for deallocation.
		972 */
		973
		974 if (entp == (net_hash_entry_t) 0) {

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		975 queue_remove(&ifp->if_rcv_port_list, infp, 976 net_rcv_port_t, chain); 977 ENQUEUE_DEAD(dead_infp, infp); 978 continue; 979 } else { 980 (void) hash_ent_remove (981 ifp, 982 (net_hash_header_t)infp, 983 FALSE, /* no longer used */ 984 hash_headp, 985 entp, 986 &dead_entp); 987 continue; 988 } 989 } MK84 also discloses deallocating the memory used by the expired records. *See*, *e.g.*, net_io.c at 1058-64: 1058 /* 1059 * Deallocate dead filters. 1060 */ 1061 if (dead_infp != 0) 1062 net_free_dead_infp(dead_infp); 1063 if (dead_entp != 0) 1064 net_free_dead_entp(dead_entp);
	[7d] inserting, retrieving or deleting one of the records from the system	MK84 discloses inserting, retrieving or deleting one of the records from the system following the step of removing.

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		**net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**

	following the step of removing.	For example, MK84 includes functionality to use retrieve records from the linked list to determine whether the ordering of the filters is wrong, and to adjust priority values. Depending on claim construction, this takes place after the step of removing from the linked list. *See, e.g.*, net_io.c at 1026-45:
		1026 /*
		1027 * See if ordering of filters is wrong
		1028 */
		1029 if (infp->priority >= NET_HI_PRI) {
		1030 prevfp = (net_rcv_port_t) queue_prev(&infp->chain);
		1031 /*
		1032 * If infp is not the first element on the queue,
		1033 * and the previous element is at equal priority
		1034 * but has a lower count, then promote infp to
		1035 * be in front of prevfp.
		1036 */
		1037 if ((queue_t)prevfp != &ifp->if_rcv_port_list &&
		1038 infp->priority == prevfp->priority) {
		1039 /*
		1040 * Threshold difference to prevent thrashing
		1041 */
		1042 if (net_filter_queue_reorder
		1043 && (100 + prevfp->rcv_count < rcount))
		1044 reorder_queue(&prevfp->chain, &infp->chain);
		1045 }

4. The method according to claim 3 further including the step of dynamically determining maximum	8. The method according to claim 7 further including the step of dynamically determining maximum	MK84 discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, the net_filter() function in net_io.c determines whether to remove

EXHIBIT D-14

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>net_filter() in net_io.c from version MK84 of the Mach kernel (1993)
(hereinafter “MK84”) alone and in combination</th></tr>
<tr><td>number of expired ones of the records to remove when the linked list is accessed.</td><td>number of expired ones of the records to remove when the linked list is accessed.</td><td>

one or zero elements from the linked list of records. For example, the code at line 968 determines if the filter is dead. If so, then it is to be removed; but if the filter is not dead then it is not to be removed. *See*, *e.g.*, net_io.c at 968-89:

```
968         if (!IP_VALID(dest)) {
969             /*
970              * This filter is dead.  We remove it from the
971              * filter list and set it aside for deallocation.
972              */
973
974             if (entp == (net_hash_entry_t) 0) {
975                 queue_remove(&ifp->if_rcv_port_list, infp,
976                             net_rcv_port_t, chain);
977                 ENQUEUE_DEAD(dead_infp, infp);
978                 continue;
979             } else {
980                 (void) hash_ent_remove (
981                                 ifp,
982                                 (net_hash_header_t)infp,
983                                 FALSE,      /* no longer used */
984                                 hash_headp,
985                                 entp,
986                                 &dead_entp);
987                 continue;
988             }
989         }
```

Further, MK84 combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage

</td></tr>
</table>

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both MK84 and Dirks relate to deletion of aged records upon the allocation

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks’ dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks’ deletion decision procedure with MK84 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks’ dynamic determination of the suitable number of entries to examine during each step of the sweeping process with MK84 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in MK84 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both MK84 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining MK84 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine MK84 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining MK84 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine MK84 with Thatte. Alternatively, it would also be obvious to combine MK84 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent:

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-14

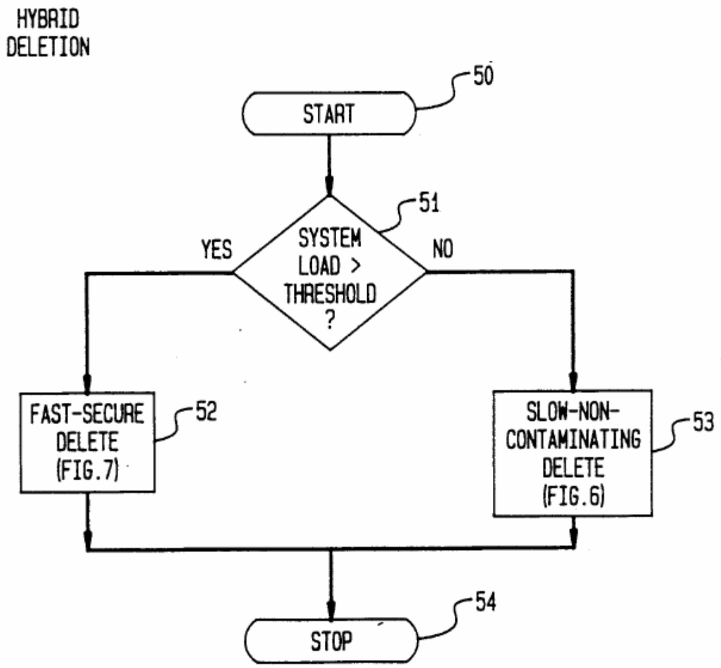
Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON- CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both MK84 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with MK84 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with MK84 and

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine MK84 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both MK84 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with MK84 would be nothing more than the predictable use of prior

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with MK84 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in MK84 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in MK84 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by MK84 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with MK84. For example, both Linux 2.0.1 and MK84 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1`

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux

EXHIBIT D-14

Asserted Claims From U.S. Pat. No. 5,893,120		net_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, MK84 discloses an information storage and retrieval system. For example, MK84 discloses a hash table with queues, which are doubly-linked lists, of automatically-expiring records. *See, e.g.*, net_io.c, at 479-87: 479 struct net_hash_entry { 480 queue_chain_t chain; /* list of entries with same hval */ 481 #define he_next chain.next 482 #define he_prev chain.prev 483 ipc_port_t rcv_port; /* destination port */ 484 int rcv_qlimit; /* qlimit for the port */ 485 unsigned int keys[N_NET_HASH_KEYS]; 486 }; 487 typedef struct net_hash_entry *net_hash_entry_t;
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	MK84 discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. MK84 also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, MK84 discloses using a queue, which is a doubly-linked list, as well as an external chaining technique. *See, e.g.*, net_io.c, at 479-87: 479 struct net_hash_entry { 480 queue_chain_t chain; /* list of entries with same hval */ 481 #define he_next chain.next 482 #define he_prev chain.prev

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**

		483 ipc_port_t rcv_port; /* destination port */
		484 int rcv_qlimit; /* qlimit for the port */
		485 unsigned int keys[N_NET_HASH_KEYS];
		486 };
		487 typedef struct net_hash_entry *net_hash_entry_t;
		
		MK84 also discloses records automatically expiring. For example, the net_set_filter() function in net_io.c deals with records corresponding to filters that have invalid ports. *See, e.g.*, net_io.c, at 1381-1416:
		
		1381 for (i = 0; i < NET_HASH_SIZE; i++) {
		1382 head = &((net_hash_header_t) infp)->table[i];
		1383 if (*head == 0)
		1384 continue;
		1385
		1386 /*
		1387 * Check each hash entry to make sure the
		1388 * destination port is still valid. Remove
		1389 * any invalid entries.
		1390 */
		1391 entp = *head;
		1392 do {
		1393 nextentp = (net_hash_entry_t) entp->he_next;
		1394
		1395 /* checked without
		1396 ip_lock(entp->rcv_port) */
		1397 if (entp->rcv_port == rcv_port
		1398 || !IP_VALID(entp->rcv_port)
		1399 || !ip_active(entp->rcv_port)) {

<u>**EXHIBIT D-15**</u>

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		1400 1401 ret = hash_ent_remove (ifp, 1402 (net_hash_header_t)infp, 1403 (my_infp == infp), 1404 head, 1405 entp, 1406 &dead_entp); 1407 if (ret) 1408 goto hash_loop_end; 1409 } 1410 1411 entp = nextentp; 1412 /* While test checks head since hash_ent_remove 1413 might modify it. 1414 */ 1415 } while (*head != 0 && entp != *head); 1416 }
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	MK84 discloses a record search means utilizing a search key to access the linked list. MK84 also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. For example, MK84 includes functionality to use a pointer to traverse a linked list having the same hash address. *See, e.g.*, net_io.c, at 1381-1428: 1360 /* 1361 * Look for an existing filter on the same reply port. 1362 * Look for filters with dead ports (for GC). 1363 * Look for a filter with the same code except KEY insns.

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
 */

simple_lock(&ifp->if_rcv_port_list_lock);

FILTER_ITERATE(ifp, infp, nextfp)
{
        if (infp->rcv_port == MACH_PORT_NULL) {
                if (match != 0
                    && infp->priority == priority
                    && my_infp == 0
                    && (infp->filter_end - infp->filter) == filter_count
                    && bpf_eq((bpf_insn_t)infp->filter,
                              (bpf_insn_t)filter, filter_bytes))
                        {
                                my_infp = infp;
                        }

                for (i = 0; i < NET_HASH_SIZE; i++) {
                        head = &((net_hash_header_t) infp)->table[i];
                        if (*head == 0)
                                continue;

                        /*
                         * Check each hash entry to make sure the
                         * destination port is still valid.  Remove
                         * any invalid entries.
                         */
                        entp = *head;
                        do {
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		nextentp = (net_hash_entry_t) entp->he_next;
		
		/* checked without
		ip_lock(entp->rcv_port) */
		if (entp->rcv_port == rcv_port
		|| !IP_VALID(entp->rcv_port)
		|| !ip_active(entp->rcv_port)) {
		
		ret = hash_ent_remove (ifp,
		(net_hash_header_t)infp,
		(my_infp == infp),
		head,
		entp,
		&dead_entp);
		if (ret)
		goto hash_loop_end;
		}
		
		entp = nextentp;
		/* While test checks head since hash_ent_remove
		might modify it.
		*/
		} while (*head != 0 && entp != *head);
		}
		hash_loop_end:
		;
		
		} else if (infp->rcv_port == rcv_port
		|| !IP_VALID(infp->rcv_port)

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
1422                    || !ip_active(infp->rcv_port)) {
1423                    /* Remove the old filter from list */
1424                    remqueue(&ifp->if_rcv_port_list, (queue_entry_t)infp);
1425                    ENQUEUE_DEAD(dead_infp, infp);
1426            }
1427    }
1428    FILTER_ITERATE_END
```

The FILTER_ITERATE() macro used at line 942 is defined in net_io.c at lines 516-21:

```
516 #define FILTER_ITERATE(ifp, fp, nextfp) \
517         for ((fp) = (net_rcv_port_t) queue_first(&(ifp)->if_rcv_port_list);\
518              !queue_end(&(ifp)->if_rcv_port_list, (queue_entry_t)(fp));    \
519              (fp) = (nextfp)) {                                            \
520                 (nextfp) = (net_rcv_port_t) queue_next(&(fp)->chain);
521 #define FILTER_ITERATE_END }
```

The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:

```
2216 /*
2217  * Removes a hash entry (ENTP) from its queue (HEAD).
2218  * If the reference count of filter (HP) becomes zero and not USED,
2219  * HP is removed from ifp->if_rcv_port_list and is freed.
2220  */
2221
2222 boolean_t
2223 hash_ent_remove (
2224    struct ifnet        *ifp,
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
    net_hash_header_t   hp,
    int                 used,
    net_hash_entry_t    *head,
    net_hash_entry_t    entp,
    queue_entry_t       *dead_p)
{
        hp->ref_count--;

        if (*head == entp) {

                if (queue_empty((queue_t) entp)) {
                        *head = 0;
                        ENQUEUE_DEAD(*dead_p, entp);
                        if (hp->ref_count == 0 && !used) {
                                remqueue((queue_t) &ifp->if_rcv_port_list,
                                         (queue_entry_t)hp);
                                hp->n_keys = 0;
                                return TRUE;
                        }
                        return FALSE;
                } else {
                        *head = (net_hash_entry_t)queue_next((queue_t) entp);
                }
        }

        remqueue((queue_t)*head, (queue_entry_t)entp);
        ENQUEUE_DEAD(*dead_p, entp);
        return FALSE;
}
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**

[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	MK84 discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. MK84 also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed.
		For example, MK84 includes the functionality to identify and remove expired records from the linked list when the linked list is accessed. For example, in the net_set_filter() function in net_io.c, the linked list is accessed for a purpose other than garbage collection. *See, e.g.*, net_io.c, at 1360-1428:
		1360 /*
		1361 * Look for an existing filter on the same reply port.
		1362 * Look for filters with dead ports (for GC).
		1363 * Look for a filter with the same code except KEY insns.
		1364 */
		1365
		1366 simple_lock(&ifp->if_rcv_port_list_lock);
		1367
		1368 FILTER_ITERATE(ifp, infp, nextfp)
		1369 {
		1370 if (infp->rcv_port == MACH_PORT_NULL) {
		1371 if (match != 0
		1372 && infp->priority == priority
		1373 && my_infp == 0
		1374 && (infp->filter_end - infp->filter) == filter_count
		1375 && bpf_eq((bpf_insn_t)infp->filter,
		1376 (bpf_insn_t)filter, filter_bytes))

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		{
		my_infp = infp;
		}
		
		for (i = 0; i < NET_HASH_SIZE; i++) {
		head = &((net_hash_header_t) infp)->table[i];
		if (*head == 0)
		continue;
		
		/*
		* Check each hash entry to make sure the
		* destination port is still valid. Remove
		* any invalid entries.
		*/
		entp = *head;
		do {
		nextentp = (net_hash_entry_t) entp->he_next;
		
		/* checked without
		ip_lock(entp->rcv_port) */
		if (entp->rcv_port == rcv_port
		|| !IP_VALID(entp->rcv_port)
		|| !ip_active(entp->rcv_port)) {
		
		ret = hash_ent_remove (ifp,
		(net_hash_header_t)infp,
		(my_infp == infp),
		head,
		entp,

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		1406 &dead_entp); 1407 if (ret) 1408 goto hash_loop_end; 1409 } 1410 1411 entp = nextentp; 1412 /* While test checks head since hash_ent_remove 1413 might modify it. 1414 */ 1415 } while (*head != 0 && entp != *head); 1416 } 1417 hash_loop_end: 1418 ; 1419 1420 } else if (infp->rcv_port == rcv_port 1421 || !IP_VALID(infp->rcv_port) 1422 || !ip_active(infp->rcv_port)) { 1423 /* Remove the old filter from list */ 1424 remqueue(&ifp->if_rcv_port_list, (queue_entry_t)infp); 1425 ENQUEUE_DEAD(dead_infp, infp); 1426 } 1427 } 1428 FILTER_ITERATE_END The FILTER_ITERATE() macro used at line 942 is defined in net_io.c at lines 516-21: 516 #define FILTER_ITERATE(ifp, fp, nextfp) \\ 517 for ((fp) = (net_rcv_port_t) queue_first(&(ifp)->if_rcv_port_list);\\

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
 518             !queue_end(&(ifp)->if_rcv_port_list, (queue_entry_t)(fp));    \
 519             (fp) = (nextfp)) {                                         \
 520               (nextfp) = (net_rcv_port_t) queue_next(&(fp)->chain);
 521 #define FILTER_ITERATE_END }
```

As shown in the example above, MK84 accesses the linked list of records. MK84 also identifies and removes expired ones of the records when the linked list is accessed. An example of this is in the net_set_filter() function in MK84. If the record's non-matching rcv_port is invalid or inactive, then the record in the linked list is removed. *See*, *e.g.*, net_io.c at 1386-1418:

```
1386                    /*
1387                     * Check each hash entry to make sure the
1388                     * destination port is still valid.  Remove
1389                     * any invalid entries.
1390                     */
1391                    entp = *head;
1392                    do {
1393                            nextentp = (net_hash_entry_t) entp->he_next;
1394
1395                            /* checked without
1396                               ip_lock(entp->rcv_port) */
1397                            if (entp->rcv_port == rcv_port
1398                                || !IP_VALID(entp->rcv_port)
1399                                || !ip_active(entp->rcv_port)) {
1400
1401                                    ret = hash_ent_remove (ifp,
1402                                        (net_hash_header_t)infp,
1403                                        (my_infp == infp),
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
1404                                 head,
1405                                 entp,
1406                                 &dead_entp);
1407                             if (ret)
1408                                 goto hash_loop_end;
1409                         }
1410
1411                         entp = nextentp;
1412                 /* While test checks head since hash_ent_remove
1413                    might modify it.
1414                  */
1415                 } while (*head != 0 && entp != *head);
1416             }
1417         hash_loop_end:
1418             ;
```

The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:

```
2216 /*
2217  * Removes a hash entry (ENTP) from its queue (HEAD).
2218  * If the reference count of filter (HP) becomes zero and not USED,
2219  * HP is removed from ifp->if_rcv_port_list and is freed.
2220  */
2221
2222 boolean_t
2223 hash_ent_remove (
2224     struct ifnet        *ifp,
2225     net_hash_header_t   hp,
2226     int                 used,
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		net_hash_entry_t *head,
		net_hash_entry_t entp,
		queue_entry_t *dead_p)
		{
		    hp->ref_count--;
		
		    if (*head == entp) {
		
		        if (queue_empty((queue_t) entp)) {
		            *head = 0;
		            ENQUEUE_DEAD(*dead_p, entp);
		            if (hp->ref_count == 0 && !used) {
		                remqueue((queue_t) &ifp->if_rcv_port_list,
		                    (queue_entry_t)hp);
		                hp->n_keys = 0;
		                return TRUE;
		            }
		            return FALSE;
		        } else {
		            *head = (net_hash_entry_t)queue_next((queue_t) entp);
		        }
		    }
		
		    remqueue((queue_t)*head, (queue_entry_t)entp);
		    ENQUEUE_DEAD(*dead_p, entp);
		    return FALSE;
		}

[1d] means, utilizing the	[5d] mea[n]s, utilizing the	MK84 discloses means, utilizing the record search means, for accessing the

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. MK84 also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. The "means, utilizing the record search means . . ." limitation is met, for example, by any function calling the net_set_filter() function. For example, se_setinput() calls net_set_filter() at lance.c, line 1568: 1556 /* 1557 * Install new filter. 1558 * Nothing special needs to be done here. 1559 */ 1560 io_return_t 1561 se_setinput(1562 int dev, 1563 ipc_port_t receive_port, 1564 int priority, 1565 filter_t *filter, 1566 natural_t filter_count) 1567 { 1568 return net_set_filter(&se_softc[dev]->is_if, 1569 receive_port, priority, 1570 filter, filter_count); 1571 } For example, depending on claim construction, lines 2250-51 in net_io.c meets the "deleting" and/or "removing" limitations. Note that this code is called

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination</th></tr>
<tr><td></td><td></td><td>from line 1401 in net_io.c in the net_filter() function.

<pre>
2233   if (*head == entp) {
2234
2235   if (queue_empty((queue_t) entp)) {
2236   *head = 0;
2237   ENQUEUE_DEAD(*dead_p, entp);
2238   if (hp->ref_count == 0 && !used) {
2239   remqueue((queue_t) &ifp->if_rcv_port_list,
2240   (queue_entry_t)hp);
2241   hp->n_keys = 0;
2242   return TRUE;
2243   }
2244   return FALSE;
2245   } else {
2246   *head = (net_hash_entry_t)queue_next((queue_t) entp);
2247   }
2248   }
2249
2250   remqueue((queue_t)*head, (queue_entry_t)entp);
2251   ENQUEUE_DEAD(*dead_p, entp);
</pre>

An example of deleting and removal “at the same time” is the loop at lines 1392-1415. During a single complete execution of all the iterations of the loop, the code at line 1397 may determine that the rcv_port of a record matches the rcv_port passed in and delete the record accordingly, and in a second iteration of the loop, the code at line 1398 or 1399 may determine that a record with a non-matching rcv_port has expired and then remove the expired record.</td></tr>
</table>

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
/*
 * Look for an existing filter on the same reply port.
 * Look for filters with dead ports (for GC).
 * Look for a filter with the same code except KEY insns.
 */
…
                        /*
                         * Check each hash entry to make sure the
                         * destination port is still valid.  Remove
                         * any invalid entries.
                         */
                        entp = *head;
                        do {
                                nextentp = (net_hash_entry_t) entp->he_next;

                                /* checked without
                                   ip_lock(entp->rcv_port) */
                                if (entp->rcv_port == rcv_port
                                    || !IP_VALID(entp->rcv_port)
                                    || !ip_active(entp->rcv_port)) {

                                        ret = hash_ent_remove (ifp,
                                            (net_hash_header_t)infp,
                                            (my_infp == infp),
                                            head,
                                            entp,
                                            &dead_entp);
                                        if (ret)
                                                goto hash_loop_end;
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		1409 } 1410 1411 entp = nextentp; 1412 /* While test checks head since hash_ent_remove 1413 might modify it. 1414 */ 1415 } while (*head != 0 && entp != *head); 1416 } An example of the “inserting” step is at net_io.c, lines 1477-1492. This operation and the example of “deleting” and/or “removal” limitations at lines 2050-51 and discussed above, occur within a single function and “at the same time” as claimed. 1477 /* Insert my_infp according to priority */ 1478 queue_iterate(&ifp->if_rcv_port_list, infp, net_rcv_port_t, chain) 1479 if (priority > infp->priority) 1480 break; 1481 enqueue_tail((queue_t)&infp->chain, (queue_entry_t)my_infp); 1482 } 1483 1484 if (match != 0) 1485 { /* Insert to hash list */ 1486 net_hash_entry_t *p; 1487 1488 hash_entp->rcv_port = rcv_port; 1489 for (i = 0; i < match->jt; i++) /* match->jt is n_keys */ 1490 hash_entp->keys[i] = match[i+1].k; 1491 p = &((net_hash_header_t)my_infp)->

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		`1492                    table[bpf_hash(match->jt, hash_entp->keys)];` As discussed above the code at lines 1392-1416 is an example of deleting and removal during the same complete execution of a do loop. It would be obvious to one of ordinary skill in the art to modify the code to retrieve but not delete a record, thereby retrieving a record, such as a record having a matching rcv_port, and removing an expired record "at the same time." To the extent that MK84 does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in MK84 with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since MK84 utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of MK84 with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**
		elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining MK84 with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: “*Cainsert*() inserts a new mapping, key => res, into the cache.” *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). “*Calookup*() searches for a cached entry matching the key passed as an argument.” *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). “*Caremove*() removes the cached entry whose key is given, if one exists.” *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		from the list as described below. The individual calls of cagetindex() are listed here:
		In cainsert():
		`275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In calookup():
		`333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		In caremove():
		`370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {`
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		`666 if (caisold(pcb,pce)) {`
		`667 ++pcb->cb_tos;`
		`668 caunlink(pcb,ix);`
		`669 return(NULL_IX);`
		`670 } else {`

2. The information storage	6. The information storage	MK84 discloses a means for dynamically determining maximum number for

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
and retrieval system according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	and retrieval system according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	the record search means to remove in the accessed linked list of records. For example, the net_set_filter() function in net_io.c determines whether to remove one or zero elements from the linked list of records. For example, the code at lines 1398 and 1399 determine if the record has expired. If so, then it is to be removed; but if the has not expired then it is not to be removed. *See*, *e.g.*, net_io.c at 1397-1409: 1397 if (entp->rcv_port == rcv_port 1398 || !IP_VALID(entp->rcv_port) 1399 || !ip_active(entp->rcv_port)) { 1400 1401 ret = hash_ent_remove (ifp, 1402 (net_hash_header_t)infp, 1403 (my_infp == infp), 1404 head, 1405 entp, 1406 &dead_entp); 1407 if (ret) 1408 goto hash_loop_end; 1409 } Further, MK84 combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both MK84 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with MK84 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with MK84 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in MK84 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both MK84 and Thatte teach a system of data storage

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		and retrieval, one of ordinary skill in the art would recognize that the result of combining MK84 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine MK84 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining MK84 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine MK84 with Thatte. Alternatively, it would also be obvious to combine MK84 with the '663 patent. Disclosure of these claim elements in the '663 patent is clearly shown in the chart of the '663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the '663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-15

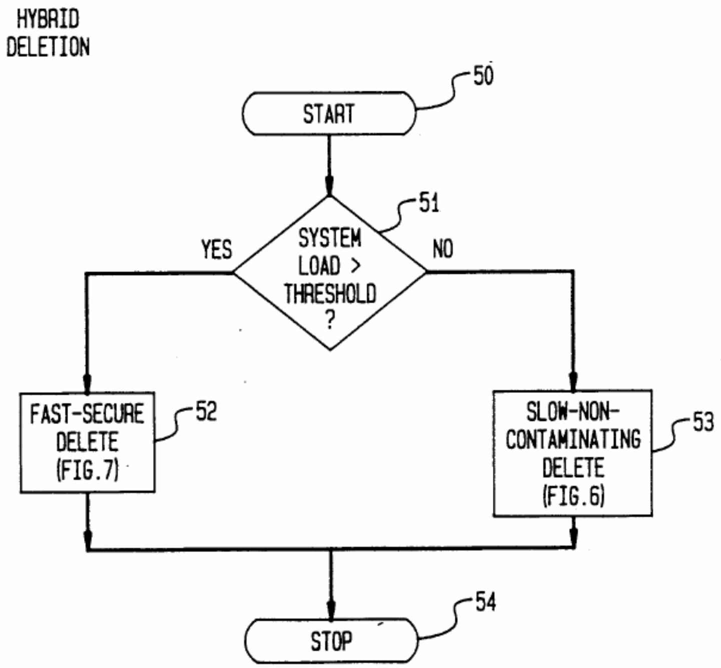
Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both MK84 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with MK84 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with MK84 and

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine MK84 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both MK84 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with MK84 would be nothing more than the predictable use of prior

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with MK84 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in MK84 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in MK84 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by MK84 in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with MK84. For example, both Linux 2.0.1 and MK84 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function rt_cache_add automatically increments an integer variable rt_cache_size. *See* Linux 2.0.1, route.c at line 1359. When the function rt_cache_add removes an expired record, the function rt_cache_add decrements the variable rt_cache_size. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable rt_cache_size indicates the number of records in the hash table (i.e., ip_rt_hash_table). Because the function rt_cache_add automatically increments and decrements the variable rt_cache_size, the variable rt_cache_size is determined dynamically. Furthermore, Linux 2.0.1 includes the function rt_garbage_collect_1. The function rt_garbage_collect_1 loops through each of the linked lists in the ip_rt_hash_table global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function rt_garbage_collect_1

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.
3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, MK84 discloses method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring as well as a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. For example, MK84 discloses a hash table with external chaining using queues, which are doubly-linked lists, of automatically-expiring records. *See, e.g.*, net_io.c, at 479-87: 479 struct net_hash_entry {

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		480 queue_chain_t chain; /* list of entries with same hval */
		481 #define he_next chain.next
		482 #define he_prev chain.prev
		483 ipc_port_t rcv_port; /* destination port */
		484 int rcv_qlimit; /* qlimit for the port */
		485 unsigned int keys[N_NET_HASH_KEYS];
		486 };
		487 typedef struct net_hash_entry *net_hash_entry_t;
		
		MK84 also discloses records automatically expiring. For example, the net_set_filter() function in net_io.c deals with records corresponding to filters that have invalid ports. *See*, *e.g.*, net_io.c, at 1381-1416:
		
		1381 for (i = 0; i < NET_HASH_SIZE; i++) {
		1382 head = &((net_hash_header_t) infp)->table[i];
		1383 if (*head == 0)
		1384 continue;
		1385
		1386 /*
		1387 * Check each hash entry to make sure the
		1388 * destination port is still valid. Remove
		1389 * any invalid entries.
		1390 */
		1391 entp = *head;
		1392 do {
		1393 nextentp = (net_hash_entry_t) entp->he_next;
		1394
		1395 /* checked without

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		1396 ip_lock(entp->rcv_port) */ 1397 if (entp->rcv_port == rcv_port 1398 || !IP_VALID(entp->rcv_port) 1399 || !ip_active(entp->rcv_port)) { 1400 1401 ret = hash_ent_remove (ifp, 1402 (net_hash_header_t)infp, 1403 (my_infp == infp), 1404 head, 1405 entp, 1406 &dead_entp); 1407 if (ret) 1408 goto hash_loop_end; 1409 } 1410 1411 entp = nextentp; 1412 /* While test checks head since hash_ent_remove 1413 might modify it. 1414 */ 1415 } while (*head != 0 && entp != *head); 1416 }
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	MK84 discloses accessing the linked list of records. MK84 also discloses accessing a linked list of records having same hash address. For example, MK84 includes functionality to use a pointer to traverse a linked list having the same hash address. *See, e.g.*, net_io.c, at 1381-1428: 1360 /*

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
	 * Look for an existing filter on the same reply port.
	 * Look for filters with dead ports (for GC).
	 * Look for a filter with the same code except KEY insns.
	 */

	simple_lock(&ifp->if_rcv_port_list_lock);

	FILTER_ITERATE(ifp, infp, nextfp)
	{
		if (infp->rcv_port == MACH_PORT_NULL) {
			if (match != 0
			    && infp->priority == priority
			    && my_infp == 0
			    && (infp->filter_end - infp->filter) == filter_count
			    && bpf_eq((bpf_insn_t)infp->filter,
					(bpf_insn_t)filter, filter_bytes))
				{
					my_infp = infp;
				}

			for (i = 0; i < NET_HASH_SIZE; i++) {
				head = &((net_hash_header_t) infp)->table[i];
				if (*head == 0)
					continue;

				/*
				 * Check each hash entry to make sure the
				 * destination port is still valid.  Remove
				 * any invalid entries.
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

		*/
		entp = *head;
		do {
		nextentp = (net_hash_entry_t) entp->he_next;
		
		/* checked without
		ip_lock(entp->rcv_port) */
		if (entp->rcv_port == rcv_port
		|| !IP_VALID(entp->rcv_port)
		|| !ip_active(entp->rcv_port)) {
		
		ret = hash_ent_remove (ifp,
		(net_hash_header_t)infp,
		(my_infp == infp),
		head,
		entp,
		&dead_entp);
		if (ret)
		goto hash_loop_end;
		}
		
		entp = nextentp;
		/* While test checks head since hash_ent_remove
		might modify it.
		*/
		} while (*head != 0 && entp != *head);
		}
		hash_loop_end:
		;

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

```
 1419
 1420            } else if (infp->rcv_port == rcv_port
 1421                      || !IP_VALID(infp->rcv_port)
 1422                      || !ip_active(infp->rcv_port)) {
 1423                   /* Remove the old filter from list */
 1424                   remqueue(&ifp->if_rcv_port_list, (queue_entry_t)infp);
 1425                   ENQUEUE_DEAD(dead_infp, infp);
 1426            }
 1427     }
 1428     FILTER_ITERATE_END
```

The FILTER_ITERATE() macro used at line 942 is defined in net_io.c at lines 516-21:

```
 516 #define FILTER_ITERATE(ifp, fp, nextfp) \
 517         for ((fp) = (net_rcv_port_t) queue_first(&(ifp)->if_rcv_port_list);\
 518              !queue_end(&(ifp)->if_rcv_port_list, (queue_entry_t)(fp));    \
 519              (fp) = (nextfp)) {                                            \
 520                 (nextfp) = (net_rcv_port_t) queue_next(&(fp)->chain);
 521 #define FILTER_ITERATE_END }
```

The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:

```
 2216 /*
 2217  * Removes a hash entry (ENTP) from its queue (HEAD).
 2218  * If the reference count of filter (HP) becomes zero and not USED,
 2219  * HP is removed from ifp->if_rcv_port_list and is freed.
 2220  */
 2221
```

EXHIBIT D-15

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>net_set_filter() in net_io.c from version MK84 of the Mach kernel
(1993) (hereinafter “MK84”) alone and in combination</th></tr>
<tr><td></td><td></td><td><pre>
boolean_t
hash_ent_remove (
    struct ifnet   *ifp,
    net_hash_header_t   hp,
    int   used,
    net_hash_entry_t   *head,
    net_hash_entry_t   entp,
    queue_entry_t   *dead_p)
{
        hp->ref_count--;

        if (*head == entp) {

                if (queue_empty((queue_t) entp)) {
                        *head = 0;
                        ENQUEUE_DEAD(*dead_p, entp);
                        if (hp->ref_count == 0 && !used) {
                                remqueue((queue_t) &ifp->if_rcv_port_list,
                                         (queue_entry_t)hp);
                                hp->n_keys = 0;
                                return TRUE;
                        }
                        return FALSE;
                } else {
                        *head = (net_hash_entry_t)queue_next((queue_t) entp);
                }
        }

        remqueue((queue_t)*head, (queue_entry_t)entp);
</pre></td></tr>
</table>

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		2251 ENQUEUE_DEAD(*dead_p, entp); 2252 return FALSE; 2253 } The remqueue() function is defined in queue.c at lines 139-45: 132 /* 133 * Remove arbitrary element from queue. 134 * Does not check whether element is on queue - the world 135 * will go haywire if it isn't. 136 */ 137 138 /*ARGSUSED*/ 139 void remqueue(140 queue_t que, 141 register queue_entry_t elt) 142 { 143 elt->next->prev = elt->prev; 144 elt->prev->next = elt->next; 145 }
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	MK84 discloses identifying at least some of the automatically expired ones of the records. For example, MK84 accesses the linked list of records and identifies and removes expired ones of the records when the linked list is accessed. An example of this is in the net_set_filter() function in MK84. If the record's non-matching rcv_port is invalid or inactive, then the record in the linked list is removed. *See*, *e.g.*, net_io.c at 1386-1418:

EXHIBIT D-15

<table>
<tr><th colspan="2">Asserted Claims From
U.S. Pat. No. 5,893,120</th><th>net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination</th></tr>
<tr><td></td><td></td><td><pre>
/*
 * Check each hash entry to make sure the
 * destination port is still valid. Remove
 * any invalid entries.
 */
entp = *head;
do {
        nextentp = (net_hash_entry_t) entp->he_next;

        /* checked without
           ip_lock(entp->rcv_port) */
        if (entp->rcv_port == rcv_port
            || !IP_VALID(entp->rcv_port)
            || !ip_active(entp->rcv_port)) {

                ret = hash_ent_remove (ifp,
                    (net_hash_header_t)infp,
                    (my_infp == infp),
                    head,
                    entp,
                    &dead_entp);
                if (ret)
                        goto hash_loop_end;
        }

        entp = nextentp;
/* While test checks head since hash_ent_remove
   might modify it.
   */
</pre></td></tr>
</table>

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**

		1415 } while (*head != 0 && entp != *head);
		1416 }
		1417 hash_loop_end:
		1418 ;
		The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:
		2216 /*
		2217 * Removes a hash entry (ENTP) from its queue (HEAD).
		2218 * If the reference count of filter (HP) becomes zero and not USED,
		2219 * HP is removed from ifp->if_rcv_port_list and is freed.
		2220 */
		2221
		2222 boolean_t
		2223 hash_ent_remove (
		2224 struct ifnet *ifp,
		2225 net_hash_header_t hp,
		2226 int used,
		2227 net_hash_entry_t *head,
		2228 net_hash_entry_t entp,
		2229 queue_entry_t *dead_p)
		2230 {
		2231 hp->ref_count--;
		2232
		2233 if (*head == entp) {
		2234
		2235 if (queue_empty((queue_t) entp)) {
		2236 *head = 0;
		2237 ENQUEUE_DEAD(*dead_p, entp);

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		2238 if (hp->ref_count == 0 && !used) {
		2239 remqueue((queue_t) &ifp->if_rcv_port_list,
		2240 (queue_entry_t)hp);
		2241 hp->n_keys = 0;
		2242 return TRUE;
		2243 }
		2244 return FALSE;
		2245 } else {
		2246 *head = (net_hash_entry_t)queue_next((queue_t) entp);
		2247 }
		2248 }
		2249
		2250 remqueue((queue_t)*head, (queue_entry_t)entp);
		2251 ENQUEUE_DEAD(*dead_p, entp);
		2252 return FALSE;
		2253 }

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	MK84 discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, in the net_set_filter() function in net_io.c, the linked list is accessed for a purpose other than garbage collection. *See, e.g.*, net_io.c, at 1360-1428: 1360 /* 1361 * Look for an existing filter on the same reply port. 1362 * Look for filters with dead ports (for GC). 1363 * Look for a filter with the same code except KEY insns. 1364 */

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination

```

simple_lock(&ifp->if_rcv_port_list_lock);

FILTER_ITERATE(ifp, infp, nextfp)
{
        if (infp->rcv_port == MACH_PORT_NULL) {
                if (match != 0
                    && infp->priority == priority
                    && my_infp == 0
                    && (infp->filter_end - infp->filter) == filter_count
                    && bpf_eq((bpf_insn_t)infp->filter,
                              (bpf_insn_t)filter, filter_bytes))
                        {
                                my_infp = infp;
                        }

                for (i = 0; i < NET_HASH_SIZE; i++) {
                        head = &((net_hash_header_t) infp)->table[i];
                        if (*head == 0)
                                continue;

                        /*
                         * Check each hash entry to make sure the
                         * destination port is still valid.  Remove
                         * any invalid entries.
                         */
                        entp = *head;
                        do {
                                nextentp = (net_hash_entry_t) entp->he_next;
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```

                        /* checked without
                           ip_lock(entp->rcv_port) */
                        if (entp->rcv_port == rcv_port
                            || !IP_VALID(entp->rcv_port)
                            || !ip_active(entp->rcv_port)) {

                                ret = hash_ent_remove (ifp,
                                    (net_hash_header_t)infp,
                                    (my_infp == infp),
                                    head,
                                    entp,
                                    &dead_entp);
                                if (ret)
                                        goto hash_loop_end;
                        }

                        entp = nextentp;
                /* While test checks head since hash_ent_remove
                   might modify it.
                   */
                } while (*head != 0 && entp != *head);
        }
    hash_loop_end:
        ;

} else if (infp->rcv_port == rcv_port
           || !IP_VALID(infp->rcv_port)
           || !ip_active(infp->rcv_port)) {
```

EXHIBIT D-15

Asserted Claims From **U.S. Pat. No. 5,893,120**		**net_set_filter() in net_io.c from version MK84 of the Mach kernel** **(1993) (hereinafter "MK84") alone and in combination**
		1423 /* Remove the old filter from list */ 1424 remqueue(&ifp->if_rcv_port_list, (queue_entry_t)infp); 1425 ENQUEUE_DEAD(dead_infp, infp); 1426 } 1427 } 1428 FILTER_ITERATE_END The FILTER_ITERATE() macro used at line 942 is defined in net_io.c at lines 516-21: 516 #define FILTER_ITERATE(ifp, fp, nextfp) \\ 517 for ((fp) = (net_rcv_port_t) queue_first(&(ifp)->if_rcv_port_list);\\ 518 !queue_end(&(ifp)->if_rcv_port_list, (queue_entry_t)(fp)); \\ 519 (fp) = (nextfp)) { \\ 520 (nextfp) = (net_rcv_port_t) queue_next(&(fp)->chain); 521 #define FILTER_ITERATE_END } As shown in the example above, MK84 accesses the linked list of records. MK84 also identifies and removes expired ones of the records when the linked list is accessed. An example of this is in the net_set_filter() function in MK84. If the record's non-matching rcv_port is invalid or inactive, then the record in the linked list is removed. *See, e.g.*, net_io.c at 1386-1418: 1386 /* 1387 * Check each hash entry to make sure the 1388 * destination port is still valid. Remove 1389 * any invalid entries. 1390 */

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

```
        entp = *head;
        do {
                nextentp = (net_hash_entry_t) entp->he_next;

                /* checked without
                   ip_lock(entp->rcv_port) */
                if (entp->rcv_port == rcv_port
                    || !IP_VALID(entp->rcv_port)
                    || !ip_active(entp->rcv_port)) {

                        ret = hash_ent_remove (ifp,
                            (net_hash_header_t)infp,
                            (my_infp == infp),
                            head,
                            entp,
                            &dead_entp);
                        if (ret)
                                goto hash_loop_end;
                }

                entp = nextentp;
        /* While test checks head since hash_ent_remove
           might modify it.
           */
        } while (*head != 0 && entp != *head);
    }
hash_loop_end:
    ;
```

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**

		The hash_ent_remove() function is defined at lines 2222-53 in net_io.c:
		2216 /*
		2217 * Removes a hash entry (ENTP) from its queue (HEAD).
		2218 * If the reference count of filter (HP) becomes zero and not USED,
		2219 * HP is removed from ifp->if_rcv_port_list and is freed.
		2220 */
		2221
		2222 boolean_t
		2223 hash_ent_remove (
		2224 struct ifnet *ifp,
		2225 net_hash_header_t hp,
		2226 int used,
		2227 net_hash_entry_t *head,
		2228 net_hash_entry_t entp,
		2229 queue_entry_t *dead_p)
		2230 {
		2231 hp->ref_count--;
		2232
		2233 if (*head == entp) {
		2234
		2235 if (queue_empty((queue_t) entp)) {
		2236 *head = 0;
		2237 ENQUEUE_DEAD(*dead_p, entp);
		2238 if (hp->ref_count == 0 && !used) {
		2239 remqueue((queue_t) &ifp->if_rcv_port_list,
		2240 (queue_entry_t)hp);
		2241 hp->n_keys = 0;
		2242 return TRUE;

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		2243 }
		2244 return FALSE;
		2245 } else {
		2246 *head = (net_hash_entry_t)queue_next((queue_t) entp);
		2247 }
		2248 }
		2249
		2250 remqueue((queue_t)*head, (queue_entry_t)entp);
		2251 ENQUEUE_DEAD(*dead_p, entp);
		2252 return FALSE;
		2253 }

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	MK84 discloses inserting, retrieving or deleting one of the records from the system following the step of removing. For example, depending on claim construction, MK84 includes functionality to delete following the step of removing. During a single complete execution of all the iterations of the loop at lines 1392-1412, the code at line 1398 or 1399 may determine that a record with a non-matching rcv_port has expired and then remove the expired record by calling hash_ent_remove at line 1401, and then in a second iteration of the loop, the code at line 1397 may determine that the rcv_port of a record matches the rcv_port passed into the function and delete the record accordingly. 1360 /* 1361 * Look for an existing filter on the same reply port. 1362 * Look for filters with dead ports (for GC). 1363 * Look for a filter with the same code except KEY insns. 1364 */

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination

		…
		/*
		 * Check each hash entry to make sure the
		 * destination port is still valid. Remove
		 * any invalid entries.
		 */
		entp = *head;
		do {
		    nextentp = (net_hash_entry_t) entp->he_next;
		
		    /* checked without
		       ip_lock(entp->rcv_port) */
		    if (entp->rcv_port == rcv_port
		        || !IP_VALID(entp->rcv_port)
		        || !ip_active(entp->rcv_port)) {
		
		        ret = hash_ent_remove (ifp,
		            (net_hash_header_t)infp,
		            (my_infp == infp),
		            head,
		            entp,
		            &dead_entp);
		        if (ret)
		            goto hash_loop_end;
		    }
		
		    entp = nextentp;
		/* While test checks head since hash_ent_remove
		   might modify it.

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		1414 */ 1415 } while (*head != 0 && entp != *head); 1416 } An example of “inserting” is at net_io.c, lines 1477-1492. This operation follows the step of removal at lines 1398-1406. 1477 /* Insert my_infp according to priority */ 1478 queue_iterate(&ifp->if_rcv_port_list, infp, net_rcv_port_t, chain) 1479 if (priority > infp->priority) 1480 break; 1481 enqueue_tail((queue_t)&infp->chain, (queue_entry_t)my_infp); 1482 } 1483 1484 if (match != 0) 1485 { /* Insert to hash list */ 1486 net_hash_entry_t *p; 1487 1488 hash_entp->rcv_port = rcv_port; 1489 for (i = 0; i < match->jt; i++) /* match->jt is n_keys */ 1490 hash_entp->keys[i] = match[i+1].k; 1491 p = &((net_hash_header_t)my_infp)-> 1492 table[bpf_hash(match->jt, hash_entp->keys)];
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of	MK84 discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. For example, the net_set_filter() function in net_io.c determines whether to remove one or zero elements from the linked list of records. For example, the

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
the records to remove when the linked list is accessed.	the records to remove when the linked list is accessed.	code at lines 1398 and 1399 determine if the record has expired. If so, then it is to be removed; but if the has not expired then it is not to be removed. *See*, *e.g.*, net_io.c at 1397-1409: 1397 if (entp->rcv_port == rcv_port 1398 || !IP_VALID(entp->rcv_port) 1399 || !ip_active(entp->rcv_port)) { 1400 1401 ret = hash_ent_remove (ifp, 1402 (net_hash_header_t)infp, 1403 (my_infp == infp), 1404 head, 1405 entp, 1406 &dead_entp); 1407 if (ret) 1408 goto hash_loop_end; 1409 } Further, MK84 combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined

EXHIBIT D-15

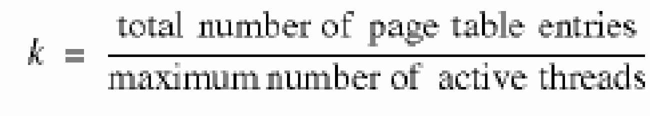
Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		number of entries that are swept is identified as *k*, where:  $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both MK84 and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with MK84 nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with MK84 and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in MK84 with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both MK84 and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining MK84 with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine MK84 with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system’s load and slowing down the system’s processing. One of ordinary skill in the art would recognize that combining MK84 with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine MK84 with Thatte. Alternatively, it would also be obvious to combine MK84 with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		4,996,663 to Nemes at 2:24-34 ("The '663 patent"). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as "deleted" and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-15

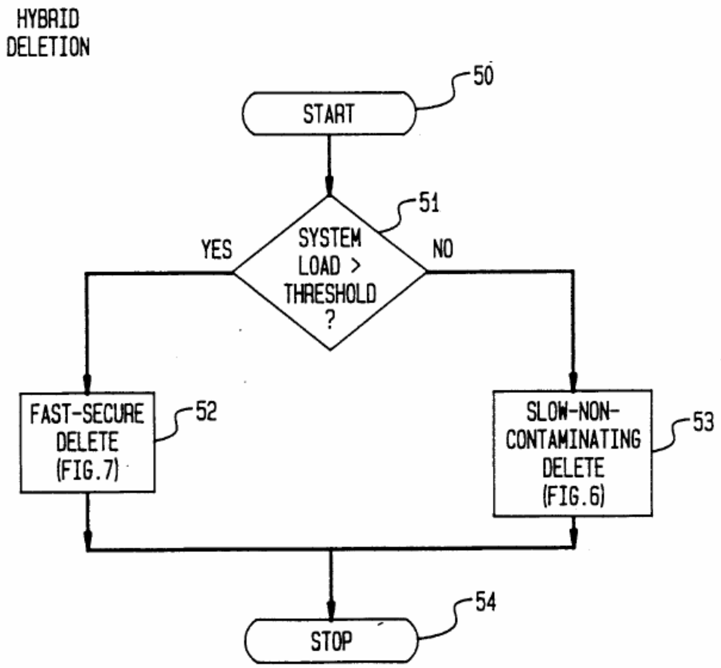
Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		FIG.5  HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both MK84 and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with MK84 would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with MK84 and

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination
		would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine MK84 with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100.

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		> This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* > If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both MK84 and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as MK84. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with MK84 would be nothing more than the predictable use of prior

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with MK84 and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in MK84 to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in MK84 with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in MK84 can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by MK84 in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with MK84. For example, both Linux 2.0.1 and MK84 describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1`

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter “MK84”) alone and in combination**
		accesses the linked list. When the function rt_garbage_collect_1 identifies a record that is expired, the function rt_garbage_collect_1 decrements the variable rt_cache_size and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable rt_cache_size can represent a dynamically determined maximum number of expired ones of the records to remove when function rt_garbage_collect_1 accesses the linked list. Furthermore, the function rt_cache_add determines whether the number of records in the hash table exceeds a predetermined threshold RT_CACHE_SIZE_MAX. If the number of records in the hash table exceeds the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_cache_add invokes a function rt_garbage_collect. *See* Linux 2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record’s last use time plus the record’s expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record’s last use time plus the record’s expiration factor is less than the current time, the

EXHIBIT D-15

Asserted Claims From **U.S. Pat. No. 5,893,120**		**net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination**
		function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux

EXHIBIT D-15

Asserted Claims From U.S. Pat. No. 5,893,120		net_set_filter() in net_io.c from version MK84 of the Mach kernel (1993) (hereinafter "MK84") alone and in combination
		2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	To the extent the preamble is a limitation, makepsres.c discloses an information storage and retrieval system.
		
		For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.
		
		`typedef struct _t_UPRResource {`
		`  char *name;`
		`  char *file;`
		`  char *category;`
		`  int found;`
		`  int noPrefix;`
		`  struct _t_UPRResource *next;`
		`} UPRResource;`
		
		`UPRResource *UPRresources[HASHSIZE];`
		ll. 442-451.
		
		Information may be retrieved from a hash table by calculating a hash key, using the hash key to index a hash table to access a linked list, and by using a while loop to traverse the linked list and access the information. *See*, *e.g.*, ll. 502-539:
		
		hash = Hash(resource->file);
		current = previous = UPRresources[hash];
		
		while (current != NULL) {
		comparison = strcmp (current->file, resource->file);

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

```
if (comparison > 0) break;

if (comparison == 0) {
        if (noPrefix) break;

        if (strcmp(current->name, resource->name) != 0 ||
            strcmp(current->category, resource->category) != 0) { /* Same */
            if (strcmp(current->category, "mkpsresPrivate") == 0 &&
                    strcmp(current->name, "NONRESOURCE") == 0) {

                    /* Replace "NONRESOURCE" entry with resource one */
                    free(current->name);
                    current->name = resource->name;
                    free(current->category);
                    current->category = resource->category;
                    free(resource->file);
                    free (resource);
                    return;
            }
            fprintf(stderr,
                    "%s:  Warning:  file %s identified as different resources\n",
                    program, resource->file);
            fprintf(stderr, "            Using %s\n", current->category);
        }
        free (resource->name);
        free (resource->file);
        free (resource->category);
        free (resource);
```

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		return;
		}
		previous = current;
		current = current->next;
		}
		
		See also, makepsres.c ll. 1-2324.

[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	makepsres.c discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. makepsres.c also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.
		
		For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.
		
		`typedef struct _t_UPRResource {`
		`  char *name;`
		`  char *file;`
		`  char *category;`
		`  int found;`
		`  int noPrefix;`
		`  struct _t_UPRResource *next;`
		`} UPRResource;`

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		`UPRResource *UPRresources[HASHSIZE];`
		ll. 442-451.
		
		The function AddUPRResource walks through the linked list in the hash table, to determine if the entries in that linked list are a “NONRESOURCE” entry, and if so, replaces those entries with a resource entry. *See* ll.458-561. A NONRESOURCE entry is an expired entry, which is removed, before it is replaced with a resource entry.
		
		`while (current != NULL) {`
		`comparison = strcmp (current->file, resource->file);`
		`if (comparison > 0) break;`
		
		`if (comparison == 0) {`
		`if (noPrefix) break;`
		
		`if (strcmp(current->name, resource->name) != 0 ||`
		`strcmp(current->category, resource->category) != 0) { /* Same */`
		`if (strcmp(current->category, "mkpsresPrivate") == 0 &&`
		`strcmp(current->name, "NONRESOURCE") == 0) {`
		
		`/* Replace "NONRESOURCE" entry with resource one */`

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

```
            free(current->name);
            current->name = resource->name;
            free(current->category);
            current->category = resource-
>category;
            free(resource->file);
            free (resource);
            return;
           }
           fprintf(stderr,
            "%s:  Warning:  file %s identified
as different resources\n",
            program, resource->file);
           fprintf(stderr, "            Using
%s\n", current->category);
       }
       free (resource->name);
       free (resource->file);
       free (resource->category);
       free (resource);
       return;
      }
      previous = current;
      current = current->next;
  }

  if (UPRresources[hash] == NULL) {
      UPRresources[hash] = resource;
```

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		`resource->next = NULL;`
		
		`} else if (current == NULL) {`
		`resource->next = NULL;`
		`previous->next = resource;`
		`} else {`
		`resource->next = current;`
		
		`if (current == UPRresources[hash]) {`
		`UPRresources[hash] = resource;`
		`} else {`
		`previous->next = resource;`
		`}`
		`}`
		ll. 505-556.
		
		See also, makepsres.c ll. 1-2324.

[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	makepsres.c discloses a record search means utilizing a search key to access the linked list. makepsres.c also discloses a record search means utilizing a search key to access a linked list of records having the same hash address.
		
		For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.
		
		`typedef struct _t_UPRResource {`
		`char *name;`
		`char *file;`

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		char *category;
		int found;
		int noPrefix;
		struct _t_UPRResource *next;
		} UPRResource;
		
		UPRResource *UPRresources[HASHSIZE];
		ll. 442-451.
		
		For example, a hash is calculated and used to index a hash table to access a linked list. A while loop is then used to traverse the linked list. *See* ll. 502-539:
		
		hash = Hash(resource->file);
		current = previous = UPRresources[hash];
		
		while (current != NULL) {
		comparison = strcmp (current->file, resource->file);
		if (comparison > 0) break;
		
		if (comparison == 0) {
		if (noPrefix) break;
		
		if (strcmp(current->name, resource->name) != 0 ||
		strcmp(current->category, resource->category) != 0) { /* Same */
		if (strcmp(current->category, "mkpsresPrivate") == 0 &&
		strcmp(current->name, "NONRESOURCE") == 0) {
		
		/* Replace "NONRESOURCE" entry with resource one */

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**

		free(current->name);
		current->name = resource->name;
		free(current->category);
		current->category = resource->category;
		free(resource->file);
		free (resource);
		return;
		}
		fprintf(stderr,
		"%s: Warning: file %s identified as different resources\n",
		program, resource->file);
		fprintf(stderr, " Using %s\n", current->category);
		}
		free (resource->name);
		free (resource->file);
		free (resource->category);
		free (resource);
		return;
		}
		previous = current;
		current = current->next;
		}
		
		See also, makepsres.c ll. 1-2324.

[1c] the record search means including a means	[5c] the record search means including means for	makepsres.c discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	the linked list when the linked list is accessed. makepsres.c also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed.
		For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.
		`typedef struct _t_UPRResource {`
		`  char *name;`
		`  char *file;`
		`  char *category;`
		`  int found;`
		`  int noPrefix;`
		`  struct _t_UPRResource *next;`
		`} UPRResource;`
		`UPRResource *UPRresources[HASHSIZE];`
		ll. 442-451.
		A function, Hash(), calculates a hash key:
		`hash = Hash(resource->file);`
		`current = previous = UPRresources[hash];`
		ll. 502-503
		`int Hash(string)`
		`    char *string;`

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		{
		int hash = 0;
		unsigned char *ch = (unsigned char *) string;
		
		while (1) {
		if (*ch == '\0') return hash % HASHSIZE;
		if (*(ch+1) == '\0') {
		hash += *ch;
		return hash % HASHSIZE;
		}
		hash += *ch++ + (*ch++ << 8);
		}
		}
		ll.244-258.
		
		The hash key is then used to index a hash table to access a linked list. A while loop is then used to traverse the linked list. *See* ll. 502-539:
		
		hash = Hash(resource->file);
		current = previous = UPRresources[hash];
		
		while (current != NULL) {
		comparison = strcmp (current->file, resource->file);
		if (comparison > 0) break;
		
		if (comparison == 0) {
		if (noPrefix) break;

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**

		if (strcmp(current->name, resource->name) != 0 ||
		strcmp(current->category, resource->category) != 0) { /* Same */
		if (strcmp(current->category, "mkpsresPrivate") == 0 &&
		strcmp(current->name, "NONRESOURCE") == 0) {
		
		/* Replace "NONRESOURCE" entry with resource one */
		free(current->name);
		current->name = resource->name;
		free(current->category);
		current->category = resource->category;
		free(resource->file);
		free (resource);
		return;
		}
		fprintf(stderr,
		"%s: Warning: file %s identified as different resources\n",
		program, resource->file);
		fprintf(stderr, " Using %s\n", current->category);
		}
		free (resource->name);
		free (resource->file);
		free (resource->category);
		free (resource);
		return;
		}
		previous = current;
		current = current->next;

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		`}` *See also*, makepsres.c ll. 1-2324.

[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	makepsres.c discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. makepsres.c also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.
		`typedef struct _t_UPRResource {`
		`  char *name;`
		`  char *file;`
		`  char *category;`
		`  int found;`
		`  int noPrefix;`
		`  struct _t_UPRResource *next;`
		`} UPRResource;`
		`UPRResource *UPRresources[HASHSIZE];`
		ll. 442-451.
		A function, Hash(), calculates a hash key:

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		`hash = Hash(resource->file);`
		`current = previous = UPRresources[hash];`
		ll. 502-503
		
		`int Hash(string)`
		`    char *string;`
		`{`
		`    int hash = 0;`
		`    unsigned char *ch = (unsigned char *)`
		`string;`
		
		`    while (1) {`
		`     if (*ch == '\0') return hash % HASHSIZE;`
		`     if (*(ch+1) == '\0') {`
		`         hash += *ch;`
		`         return hash % HASHSIZE;`
		`     }`
		`     hash += *ch++ + (*ch++ << 8);`
		`    }`
		`}`
		ll.244-258.
		
		The hash key is then used to index a hash table to access a linked list. A while loop is then used to traverse the linked list. During traversal of the linked list, a “NONRESOURCE” entry is removed and at the same time a resource entry is inserted in its place. *See* ll. 502-539:
		
		hash = Hash(resource->file);

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

```
current = previous = UPRresources[hash];

while (current != NULL) {
    comparison = strcmp (current->file, resource->file);
    if (comparison > 0) break;

    if (comparison == 0) {
            if (noPrefix) break;

            if (strcmp(current->name, resource->name) != 0 ||
                strcmp(current->category, resource->category) != 0) { /* Same */
                if (strcmp(current->category, "mkpsresPrivate") == 0 &&
                        strcmp(current->name, "NONRESOURCE") == 0) {

                        /* Replace "NONRESOURCE" entry with resource one */
                        free(current->name);
                        current->name = resource->name;
                        free(current->category);
                        current->category = resource->category;
                        free(resource->file);
                        free (resource);
                        return;
                }
                fprintf(stderr,
                        "%s:  Warning:  file %s identified as different resources\n",
                        program, resource->file);
                fprintf(stderr, "           Using %s\n", current->category);
            }
```

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		free (resource->name);
		free (resource->file);
		free (resource->category);
		free (resource);
		return;
		}
		previous = current;
		current = current->next;
		}
		
		See also, makepsres.c ll. 1-2324.
		
		To the extent that makepresres.c does not disclose this limitation, gcache.c from Xinu Operating System for Sparc (1991) (hereinafter "gcache.c") and Douglas Comer and Shawn Ostermann, *GCache: A Generalized Caching Mechanism*, Purdue University (Revised March 1992) (hereinafter "Comer") (collectively hereinafter "GCache") discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list, and also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.
		
		One of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in makepresres.c with the a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring disclosed by GCache. *See, e.g.*, Comer at 3-10. For example, since makepresres.c utilizes a linked list for storing records and GCache discloses a system that attaches or chains linked lists to a hash table for storing records, one of ordinary skill in the art would be motivated to combine the linked list of

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		makepresres.c with the system including a hash table using external chaining of linked lists disclosed by GCache. The disclosure of these claim elements in GCache is clearly shown in the chart of GCache, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, one of ordinary skill in the art would recognize that the result of combining makepresres.c with GCache would be nothing more than the predictable use of prior art elements according to their established functions. For example, Comer discloses means for inserting, retrieving, and deleting records: "*Cainsert*() inserts a new mapping, key => res, into the cache." *See* Comer at 4. *See also*, gcache.c at lines 246-304, defining cainsert(). "*Calookup*() searches for a cached entry matching the key passed as an argument." *See* Comer at 4. *See also*, gcache.c at lines 312-347 and 643-678, defining calookup() and cagetindex(). "*Caremove*() removes the cached entry whose key is given, if one exists." *See* Comer at 4. *See also*, gcache.c at lines 355-376, defining caremove(). Each of the means for inserting, retrieving, and deleting, utilizes a record search means, the function cagetindex(), which removes an expired record from the list as described below. The individual calls of cagetindex() are listed here:

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		In cainsert():
		275 if ((ixnew = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		
		In calookup():
		333 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		
		In caremove():
		370 if ((ix = cagetindex(pcb,pkey,keylen,hash)) != NULL_IX) {
		
		"In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned." *See* Comer at 10.
		
		At lines 655-670 of gcache.c, cagetindex() executes a *while* loop to traverse a linked list attached to a bucket of the hash table, accessing records stored therein. In the subset of that code listed below, cagetindex() utilizes caisold() to identify if a matching record is expired and removes the expired record from the linked list using caunlink():
		
		666 if (caisold(pcb,pce)) {
		667 ++pcb->cb_tos;
		668 caunlink(pcb,ix);
		669 return(NULL_IX);
		670 } else {

2. The information storage and retrieval system	6. The information storage and retrieval system	Makepsres.c combined with Dirks, Thatte, the '663 patent and/or the Opportunistic Garbage Collection Articles discloses an information storage

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
according to claim 1 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	according to claim 5 further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	and retrieval system further including means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT$_i$ on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both makepsres.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as makepsres.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with makepsres.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with makepsres.c and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in makepsres.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both makepsres.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining makepsres.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine makepsres.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in makepsres.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining makepsres.c with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine makepsres.c

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		with Thatte. Alternatively, it would also be obvious to combine makepsres.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT D-16

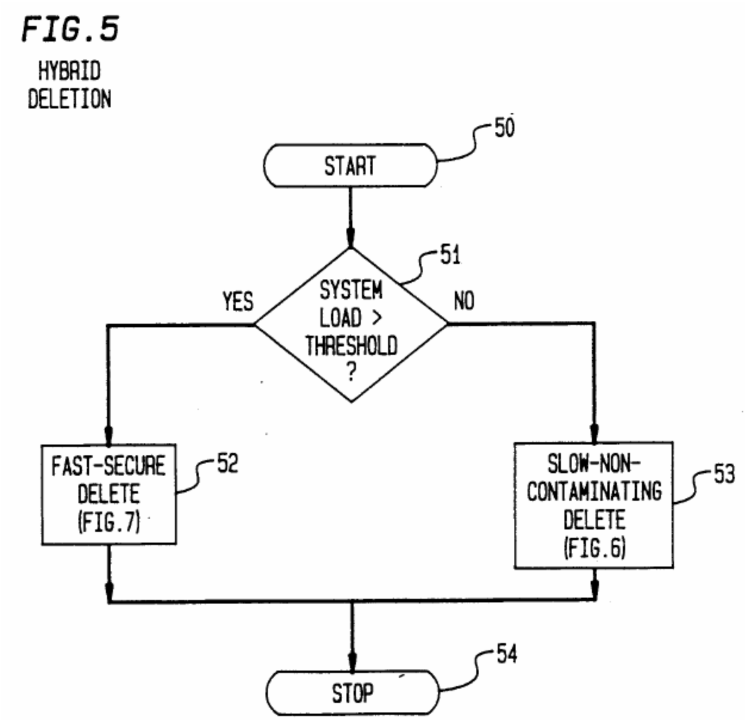
Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5.  FIG.5 HYBRID DELETION 50 START 51 SYSTEM LOAD > THRESHOLD ? YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 54 STOP *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both makepsres.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in makepsres.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		combining the ’663 patent’s deletion decision procedure with makepsres.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with makepsres.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine makepsres.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage*

EXHIBIT D-16

Asserted Claims From **U.S. Pat. No. 5,893,120**		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		Collector at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both makepsres.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as makepsres.c. Moreover, one of ordinary skill in

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles’ deletion decision procedure with makepsres.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with makepsres.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in makepsres.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in makepsres.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example,

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		the removal of expired records described in makepsres.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by makepresres.c in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with makepresres.c. For example, both Linux 2.0.1 and makepresres.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		`rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		2.0.1, route.c at lines 1341-1342. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect invokes a function rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		the maximum number of records that the function `rt_cache_add` can remove from a linked list.

3. A method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	7. A method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	To the extent the preamble is a limitation, makepsres.c discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. makepsres.c also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring.
		For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.
		`typedef struct _t_UPRResource {`
		`  char *name;`
		`  char *file;`
		`  char *category;`
		`  int found;`
		`  int noPrefix;`
		`  struct _t_UPRResource *next;`
		`} UPRResource;`
		`UPRResource *UPRresources[HASHSIZE];`
		ll. 442-451.
		Information may be retrieved from a hash table by calculating a hash key, using the hash key to index a hash table to access a linked list, and by using a

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

while loop to traverse the linked list and access the information:

For example the hash key is calculated by Hash() and the hash key is then used to index into the hash table UPRresources. ll. 502-503:

hash = Hash(resource->file);
current = previous = UPRresources[hash];

The function AddUPRResource walks through the linked list in the hash table, to determine if the entries in that linked list are a “NONRESOURCE” entry, and if so, replaces those entries with a resource entry. *See* ll.458-561. A NONRESOURCE entry is an expired entry, which is removed, before it is replaced with a resource entry.

```
      while (current != NULL) {
            comparison = strcmp (current->file,
    resource->file);
            if (comparison > 0) break;

            if (comparison == 0) {
             if (noPrefix) break;

             if (strcmp(current->name, resource-
    >name) != 0 ||
                 strcmp(current->category, resource-
    >category) != 0) { /* Same */
                 if (strcmp(current->category,
    "mkpsresPrivate") == 0 &&
```

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		`strcmp(current->name, "NONRESOURCE") == 0) {`
		
		`/* Replace "NONRESOURCE" entry with resource one */`
		`free(current->name);`
		`current->name = resource->name;`
		`free(current->category);`
		`current->category = resource->category;`
		`free(resource->file);`
		`free (resource);`
		`return;`
		`}`
		`fprintf(stderr,`
		`"%s:  Warning:  file %s identified as different resources\n",`
		`program, resource->file);`
		`fprintf(stderr, "            Using %s\n", current->category);`
		`}`
		`free (resource->name);`
		`free (resource->file);`
		`free (resource->category);`
		`free (resource);`
		`return;`
		`}`
		`previous = current;`

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		`            current = current->next;`
		`        }`
		
		`        if (UPRresources[hash] == NULL) {`
		`            UPRresources[hash] = resource;`
		`            resource->next = NULL;`
		
		`        } else if (current == NULL) {`
		`            resource->next = NULL;`
		`            previous->next = resource;`
		`        } else {`
		`            resource->next = current;`
		
		`            if (current == UPRresources[hash]) {`
		`             UPRresources[hash] = resource;`
		`            } else {`
		`             previous->next = resource;`
		`            }`
		`        }`
		ll. 505-556.
		
		See also, makepsres.c ll. 1-2324.

[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	makepsres.c discloses accessing a linked list of records. makepsres.c also discloses accessing a linked list of records having same hash address. For example, makepsres.c includes a hash table of linked lists UPRResources, of the size HashSize, which is defined as a size of 2048. See ll.75, 442-451.

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		typedef struct _t_UPRResource {
		char *name;
		char *file;
		char *category;
		int found;
		int noPrefix;
		struct _t_UPRResource *next;
		} UPRResource;
		
		UPRResource *UPRresources[HASHSIZE];
		ll. 442-451.
		
		A function, Hash(), calculates a hash key:
		
		hash = Hash(resource->file);
		current = previous = UPRresources[hash];
		ll. 502-503
		
		int Hash(string)
		char *string;
		{
		int hash = 0;
		unsigned char *ch = (unsigned char *)
		string;
		
		while (1) {
		if (*ch == '\0') return hash % HASHSIZE;

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**

		`if (*(ch+1) == '\0') {`
		`hash += *ch;`
		`return hash % HASHSIZE;`
		`}`
		`hash += *ch++ + (*ch++ << 8);`
		`}`
		`}`
		ll.244-258.
		
		The hash key is then used to index a hash table to access a linked list. A while loop is then used to traverse the linked list. *See* ll. 502-539:
		
		hash = Hash(resource->file);
		current = previous = UPRresources[hash];
		
		while (current != NULL) {
		comparison = strcmp (current->file, resource->file);
		if (comparison > 0) break;
		
		if (comparison == 0) {
		if (noPrefix) break;
		
		if (strcmp(current->name, resource->name) != 0 ||
		strcmp(current->category, resource->category) != 0) { /* Same */
		if (strcmp(current->category, "mkpsresPrivate") == 0 &&
		strcmp(current->name, "NONRESOURCE") == 0) {
		
		/* Replace "NONRESOURCE" entry with resource one */

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		free(current->name);
		current->name = resource->name;
		free(current->category);
		current->category = resource->category;
		free(resource->file);
		free (resource);
		return;
		}
		fprintf(stderr,
		"%s: Warning: file %s identified as different resources\n",
		program, resource->file);
		fprintf(stderr, " Using %s\n", current->category);
		}
		free (resource->name);
		free (resource->file);
		free (resource->category);
		free (resource);
		return;
		}
		previous = current;
		current = current->next;
		}
		See also, makepsres.c ll. 1-2324.

[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	makepsres.c discloses identifying at least some of the automatically expired ones of the records. For example, a hash key is calculated and then used to index a hash table to

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		access a linked list. A while loop is then used to traverse the linked list. During traversal of the linked list, checks are performed to identify a “NONRESOURCE” entry, which is then removed and following the removal a resource entry is inserted in its place. *See* ll. 502-539:
		
		hash = Hash(resource->file);
		  current = previous = UPRresources[hash];
		
		  while (current != NULL) {
		      comparison = strcmp (current->file, resource->file);
		      if (comparison > 0) break;
		
		      if (comparison == 0) {
		             if (noPrefix) break;
		
		             if (strcmp(current->name, resource->name) != 0 ||
		                 strcmp(current->category, resource->category) != 0) { /* Same */
		                 if (strcmp(current->category, "mkpsresPrivate") == 0 &&
		                        strcmp(current->name, "NONRESOURCE") == 0) {
		
		                        /* Replace "NONRESOURCE" entry with resource one */
		                        free(current->name);
		                        current->name = resource->name;
		                        free(current->category);
		                        current->category = resource->category;
		                        free(resource->file);
		                        free (resource);
		                        return;

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**

		}
		fprintf(stderr,
		"%s: Warning: file %s identified as different resources\n",
		program, resource->file);
		fprintf(stderr, " Using %s\n", current->category);
		}
		free (resource->name);
		free (resource->file);
		free (resource->category);
		free (resource);
		return;
		}
		previous = current;
		current = current->next;
		}
		
		See also, makepsres.c ll. 1-2324.

[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	makepsres.c discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. For example, a hash key is calculated and then used to index a hash table to access a linked list. A while loop is then used to traverse the linked list. During traversal of the linked list, a “NONRESOURCE” entry is removed and following the removal a resource entry is inserted in its place. *See* ll. 502-539: hash = Hash(resource->file);

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		see code below

```
current = previous = UPRresources[hash];

while (current != NULL) {
    comparison = strcmp (current->file, resource->file);
    if (comparison > 0) break;

    if (comparison == 0) {
            if (noPrefix) break;

            if (strcmp(current->name, resource->name) != 0 ||
                strcmp(current->category, resource->category) != 0) { /* Same */
                if (strcmp(current->category, "mkpsresPrivate") == 0 &&
                        strcmp(current->name, "NONRESOURCE") == 0) {

                        /* Replace "NONRESOURCE" entry with resource one */
                        free(current->name);
                        current->name = resource->name;
                        free(current->category);
                        current->category = resource->category;
                        free(resource->file);
                        free (resource);
                        return;
                }
                fprintf(stderr,
                        "%s:  Warning:  file %s identified as different resources\n",
                        program, resource->file);
                fprintf(stderr, "           Using %s\n", current->category);
            }
```

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		free (resource->name);
		free (resource->file);
		free (resource->category);
		free (resource);
		return;
		}
		previous = current;
		current = current->next;
		}
		
		See also, makepsres.c ll. 1-2324.

	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	makepsres.c discloses inserting, retrieving or deleting one of the records from the system following the step of removing.
		
		For example, a hash key is calculated and then used to index a hash table to access a linked list. A while loop is then used to traverse the linked list. During traversal of the linked list, a "NONRESOURCE" entry is removed and following the removal a resource entry is inserted in its place. *See* ll. 502-539:
		
		hash = Hash(resource->file);
		current = previous = UPRresources[hash];
		
		while (current != NULL) {
		comparison = strcmp (current->file, resource->file);
		if (comparison > 0) break;

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

```
if (comparison == 0) {
        if (noPrefix) break;

        if (strcmp(current->name, resource->name) != 0 ||
            strcmp(current->category, resource->category) != 0) { /* Same */
            if (strcmp(current->category, "mkpsresPrivate") == 0 &&
                    strcmp(current->name, "NONRESOURCE") == 0) {

                    /* Replace "NONRESOURCE" entry with resource one */
                    free(current->name);
                    current->name = resource->name;
                    free(current->category);
                    current->category = resource->category;
                    free(resource->file);
                    free (resource);
                    return;
            }
            fprintf(stderr,
                    "%s:  Warning:  file %s identified as different resources\n",
                    program, resource->file);
            fprintf(stderr, "            Using %s\n", current->category);
        }
        free (resource->name);
        free (resource->file);
        free (resource->category);
        free (resource);
        return;
}
```

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)

		previous = current;
		current = current->next;
		}
		See also, makepsres.c ll. 1-2324.

4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	Makepsres.c combined with Dirks, Thatte, the ’663 patent and/or the Opportunistic Garbage Collection Articles discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both makepsres.c and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as makepsres.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with makepsres.c nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with makepsres.c and would

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in makepsres.c with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both makepsres.c and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining makepsres.c with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine makepsres.c with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in makepsres.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining makepsres.c with the teachings of Thatte would solve this problem by

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine makepsres.c with Thatte. Alternatively, it would also be obvious to combine makepsres.c with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-16

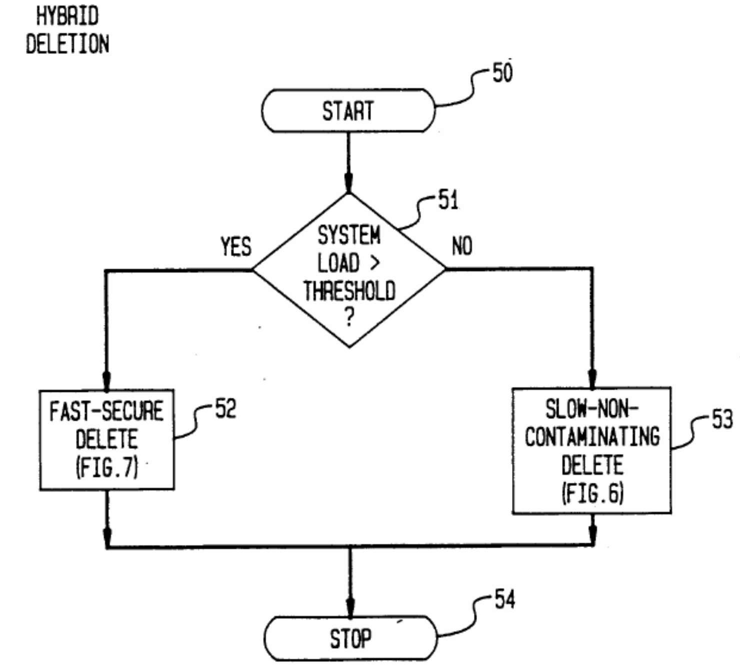
Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		FIG.5  HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the ’663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both makepsres.c and the ’663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in makepsres.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” The ’120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the ’663 patent’s deletion decision procedure with makepsres.c would be nothing more than the predictable use of prior art elements according to their established functions.

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with makepsres.c and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine makepsres.c with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally,* Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both makepsres.c and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as makepsres.c. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with makepsres.c would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with makepsres.c and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in makepsres.c to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in makepsres.c with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in makepsres.c can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the ‘120 patent that “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one.” ‘120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by makepresres.c in combination with Dirks, Thatte, the ‘663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with makepresres.c. For example, both Linux 2.0.1 and makepresres.c describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because

Asserted Claims From U.S. Pat. No. 5,893,120		makepsres.c, distributed as part of Plug And Play Linux distribution (1995)
		the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function

EXHIBIT D-16

Asserted Claims From U.S. Pat. No. 5,893,120		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		rt_garbage_collect_1. *See* Linux 2.0.1, route.c at line 1293. The function rt_garbage_collect_1 loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function rt_garbage_collect_1 looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function rt_garbage_collect_1 determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function rt_garbage_collect_1 removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable expire and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value RT_CACHE_TIMEOUT. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function rt_garbage_collect_1 determines again whether the number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold RT_CACHE_SIZE_MAX, the function rt_garbage_collect_1 halves the variable expire and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the

EXHIBIT D-16

Asserted Claims From **U.S. Pat. No. 5,893,120**		**makepsres.c, distributed as part of Plug And Play Linux distribution (1995)**
		predetermined threshold `RT_CACHE_SIZE_MAX`. Under Bedrock's proposed claim constructions, the records removed by the function `rt_garbage_collect_1` are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function `rt_cache_add` only removes a record from a linked list when the record's last use time plus the fixed timeout value `RT_CACHE_TIMEOUT` is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function `rt_cache_add` can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function `rt_garbage_collect_1` can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function `rt_garbage_collect_1` can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function `rt_garbage_collect_1` can remove from a linked list is different than the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
1. An information storage and retrieval system, the system comprising:	5. An information storage and retrieval system, the system comprising:	On information and belief, all of the techniques claimed by the ‘120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the ’120 Patent. To the extent the preamble is a limitation, on information and belief, the source code for LAT discloses an information storage and retrieval system. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
[1a] a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring,	[5a] a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring,	On information and belief, all of the techniques claimed by the ‘120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the ’120 Patent. On information and belief, the source code for LAT discloses a linked list to store and provide access to records stored in a memory of the system, at least some of the records automatically expiring. On information and belief, the source code for LAT also discloses a hashing means to provide access to records stored in a memory of the system and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
[1b] a record search means utilizing a search key to access the linked list,	[5b] a record search means utilizing a search key to access a linked list of records having the same hash address,	On information and belief, all of the techniques claimed by the ‘120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the ’120 Patent. On information and belief, the source code for LAT discloses a record search means utilizing a search key to access the linked list. On information and belief, the source code for LAT also discloses a record search means utilizing a search key to access a linked list of records having the same hash address. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
[1c] the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed, and	[5c] the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed, and	On information and belief, all of the techniques claimed by the '120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the '120 Patent. On information and belief, the source code for LAT discloses the record search means including a means for identifying and removing at least some of the expired ones of the records from the linked list when the linked list is accessed. On information and belief, the source code for LAT also discloses the record search means including means for identifying and removing at least some expired ones of the records from the linked list of records when the linked list is accessed. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
[1d] means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list.	[5d] mea[n]s, utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records.	On information and belief, all of the techniques claimed by the '120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the '120 Patent. On information and belief, the source code for LAT discloses means, utilizing the record search means, for accessing the linked list and, at the same time, removing at least some of the expired ones of the records in the linked list. On information and belief, the source code for LAT also discloses utilizing the record search means, for inserting, retrieving, and deleting records from the system and, at the same time, removing at least some expired ones of the records in the accessed linked list of records. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
2. The information storage and retrieval system according to claim 1 further including means for	6. The information storage and retrieval system according to claim 5 further including means for	On information and belief, all of the techniques claimed by the '120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the '120 Patent. On information and belief, the source code for LAT discloses, means for dynamically determining maximum

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	dynamically determining maximum number for the record search means to remove in the accessed linked list of records.	number for the record search means to remove in the accessed linked list of records. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced. Furthermore, on information and belief, a person of ordinary skill in the art would have been motivated to combine LAT with the techniques taught by Linux 2.0.1, Dirks, Thatte, the ‘663 patent, and/or the Opportunistic Garbage Collection Articles to disclose means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both LAT and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as LAT. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with LAT nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with LAT and would have

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in LAT with the means for dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both LAT and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining LAT with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine LAT with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in LAT can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining LAT with the teachings of Thatte would solve this problem by dynamically determining how

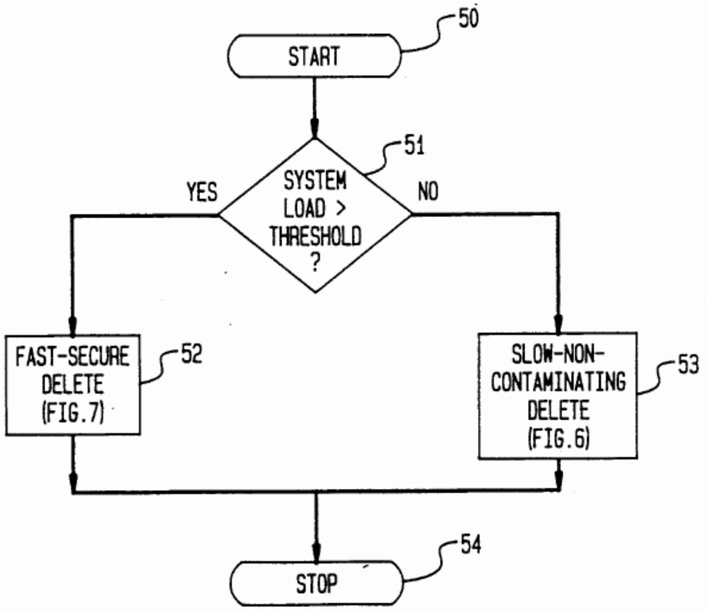
EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine LAT with Thatte. Alternatively, it would also be obvious to combine LAT with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels. *Id.* at 2:42-46. This hybrid deletion is shown in Figure 5.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		FIG.5 HYBRID DELETION  START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both LAT and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in LAT. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663 patent's deletion decision procedure with LAT would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		combined the '663 patent's dynamic decision on whether to perform a deletion based on a systems load as taught by the '663 patent and with LAT and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine LAT with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA '89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32. Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
		responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation’s space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both LAT and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles’ dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as LAT. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the ’120 patent states “[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with LAT would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with LAT and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in LAT to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in LAT with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in LAT can be burdensome on the system, adding to the system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by LAT in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with LAT. For example, both Linux 2.0.1 and LAT describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function `rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, LINUX 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux 2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function `rt_garbage_collect_1` can remove additional records from the linked lists in the hash table. The function `rt_garbage_collect_1` repeats this process until the total number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than the maximum number of records that the function rt_cache_add can remove from a linked list.
3. A method for storing	7. A method for storing	On information and belief, all of the techniques claimed by the '120 patent

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring, the method comprising the steps of:	and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring, the method comprising the steps of:	were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the '120 Patent. To the extent the preamble is a limitation, on information and belief, the source code for LAT discloses a method for storing and retrieving information records using a linked list to store and provide access to the records, at least some of the records automatically expiring. On information and belief, the source code for LAT also discloses a method for storing and retrieving information records using a hashing technique to provide access to the records and using an external chaining technique to store the records with same hash address, at least some of the records automatically expiring. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
[3a] accessing the linked list of records,	[7a] accessing a linked list of records having same hash address,	On information and belief, all of the techniques claimed by the '120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the '120 Patent. On information and belief, the source code for LAT discloses accessing a linked list of records. On information and belief, the source code for LAT also discloses accessing a linked list of records having same hash address. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
[3b] identifying at least some of the automatically expired ones of the records, and	[7b] identifying at least some of the automatically expired ones of the records,	On information and belief, all of the techniques claimed by the '120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the '120 Patent. On information and belief, the source code for LAT discloses identifying at least some of the automatically expired ones of the records. Defendants reserve the right to supplement these

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
		contentions once a complete version of the source code for LAT is produced.
[3c] removing at least some of the automatically expired records from the linked list when the linked list is accessed.	[7c] removing at least some of the automatically expired records from the linked list when the linked list is accessed, and	On information and belief, all of the techniques claimed by the ‘120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the ’120 Patent. On information and belief, the source code for LAT discloses removing at least some of the automatically expired records from the linked list when the linked list is accessed. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
	[7d] inserting, retrieving or deleting one of the records from the system following the step of removing.	On information and belief, all of the techniques claimed by the ‘120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the ’120 Patent. On information and belief, the source code for LAT discloses inserting, retrieving or deleting one of the records from the system following the step of removing. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced.
4. The method according to claim 3 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	8. The method according to claim 7 further including the step of dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed.	On information and belief, all of the techniques claimed by the ‘120 patent were implemented in LAT, which was in public use, sold, and/or offered for sale prior to the filing date of the ‘120 Patent. On information and belief, the source code for LAT discloses dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced. Furthermore, on information and belief, a person of ordinary skill in the art

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		would have been motivated to combine LAT with the techniques taught by Linux 2.0.1, Dirks, Thatte, the '663 patent, and/or the Opportunistic Garbage Collection Articles to disclose dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. Defendants reserve the right to supplement these contentions once a complete version of the source code for LAT is produced. Dirks discloses the management of memory in a computer system and more particularly to the allocation of address space in a virtual memory system, which dynamically determines how many records to sweep/remove upon each allocation. Disclosure of these claim elements in Dirks is clearly shown in Exhibit B-2, which is hereby incorporated by reference in its entirety. For example, as summarized in Dirks, each time a VSID is assigned from the free list to a new application or thread, a fixed number of entries in the page table are scanned to determine whether they have become inactive, by checking them against the VSIDs on the recycle list. Each entry which is identified as being inactive is removed from the page table. After all of the entries in the page table have been examined in this manner, the VSIDs in the recycle list can be transferred to the free list, since all of their associated page table entries will have been removed. This approach thereby guarantees that a predetermined number of VSIDs are always available in the free list without requiring a time-consuming scan of the complete page table at once. U.S. Patent No. 6,119,214 to Dirks at 7:2-14. After [a] new VSID has been allocated, the system checks a flag RFLG

EXHIBIT D-17

Asserted Claims From **U.S. Pat. No. 5,893,120**		**Local Area Transport Protocol (“LAT”) alone and in combination**
		to determine whether a recycle sweep is currently in progress (Step 20). If there is no sweep in progress, i.e. RFLG is not equal to one, a determination is made whether a sweep should be initiated. This is done by checking whether the inactive list is full, i.e. whether it contains x entries (Step 22). If the number of entries I on the inactive list is less than x, no further action is taken, and processing control returns to the operating system (Step 24). If, however, the inactive list is full at this time, the flag RFLG is set (Step 26), the VSIDs on the inactive list are transferred to the recycle list, and an index n is reset to 1 (Step 28). The system then sweeps a predetermined number of page table entries PT_i on the page table, to detect whether any of them are inactive, i.e. their associated VSID is on the recycle list (Step 30). The predetermined number of entries that are swept is identified as *k*, where: $$k = \frac{\text{total number of page table entries}}{\text{maximum number of active threads}}$$ *Id.* at 8:12-30. Dirks discloses that any approach can be employed to determine the number of entries to be examined during each step of the sweeping process. *Id.* at 7:37-40. As stated in Dirks: Any other suitable approach can be employed to determine the number of entries to be examined during each step of the sweeping process. In this regard, it is not necessary that the number of examined entries be fixed for each step. Rather, it might vary from one step to the next. The only criterion is that the number of entries examined on each step be such that all entries in the page table are examined in a determinable amount of time or by the

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		occurrence of a certain event, e.g. by the time the list of free VSIDs is empty. *Id.* at 7:38-46. Thus, Dirks dynamically determines the maximum number of records to sweep/remove by calculating a value *k*. *Id.* at 7:15-46, 7:66-8:56. As both LAT and Dirks relate to deletion of aged records upon the allocation of a new incoming record, one of ordinary skill in the art would have understood how to use Dirks' dynamic decision making process of determining the maximum number of records to sweep/remove in other hash tables implementations such as LAT. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining Dirks' deletion decision procedure with LAT nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined Dirks' dynamic determination of the suitable number of entries to examine during each step of the sweeping process with LAT and would have seen the benefits of doing so. One possible benefit, for example, is saving the system from performing sometimes time-consuming sweeps.` Alternatively, one of ordinary skill in the art would be motivated to, and would understand how to, combine the system disclosed in LAT with the means for

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		dynamically determining maximum number for the record search means to remove in the accessed linked list of records disclosed by Thatte. For example, Thatte discloses a system and method using hash tables and/or linked lists and further discloses means for dynamically determining the maximum number for the record search means to remove in the accessed linked list of records. The disclosure of these claim elements in Thatte is clearly shown in the chart of Thatte, which is hereby incorporated by reference in its entirety. Moreover, one of ordinary skill in the art would recognize that these combinations would improve the similar systems and methods in the same way. Additionally, Ass both LAT and Thatte teach a system of data storage and retrieval, one of ordinary skill in the art would recognize that the result of combining LAT with Thatte would be nothing more than the predictable use of prior art elements according to their established functions. The resulting combination would include the capability to determine the maximum number for the record search means to remove as taught by Thatte. Further, one of ordinary skill in the art would be motivated to combine LAT with Thatte and recognize the benefits of doing so. For example, the removal of expired records described in LAT can be burdensome on the system, adding to the system's load and slowing down the system's processing. One of ordinary skill in the art would recognize that combining LAT with the teachings of Thatte would solve this problem by dynamically determining how many records to delete based on, among other things, the system load. Moreover, the '120 patent discloses that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		many records to delete can be a dynamic one." '120 at 7:10-15. Thus, the '120 patent provides motivations to combine LAT with Thatte. Alternatively, it would also be obvious to combine LAT with the ’663 patent. Disclosure of these claim elements in the ’663 patent is clearly shown in the chart of the ‘663 patent, which is hereby incorporated by reference in its entirety. For example, as summarized in the ’663 patent: during normal times when the load on the storage system is not excessive, a non-contaminating but slow deletion of records is used. This slow, non-contaminating deletion involves closing the collision-resolution chain of locations by moving a record from a later position in the chain into the position of the record to be deleted. This leaves no deleted record locations in the storage space to slow down future searches. U.S. Patent 4,996,663 to Nemes at 2:24-34 (“The ’663 patent”). In times of heavy use, when deletions must be done rapidly and no time is available for decontamination, the record is simply marked as “deleted” and left in place. Later non-contaminating probes in the vicinity of such deleted record locations automatically remove the contaminating deleted records by moving records in the chain as described above. *Id.* at 2:35-41. This hybrid hashing technique has the decided advantage of automatically eliminating contamination caused by the fast-secure deletion procedure when the slower, non-contaminating deletion is used when the load on the system is at lower levels.

EXHIBIT D-17

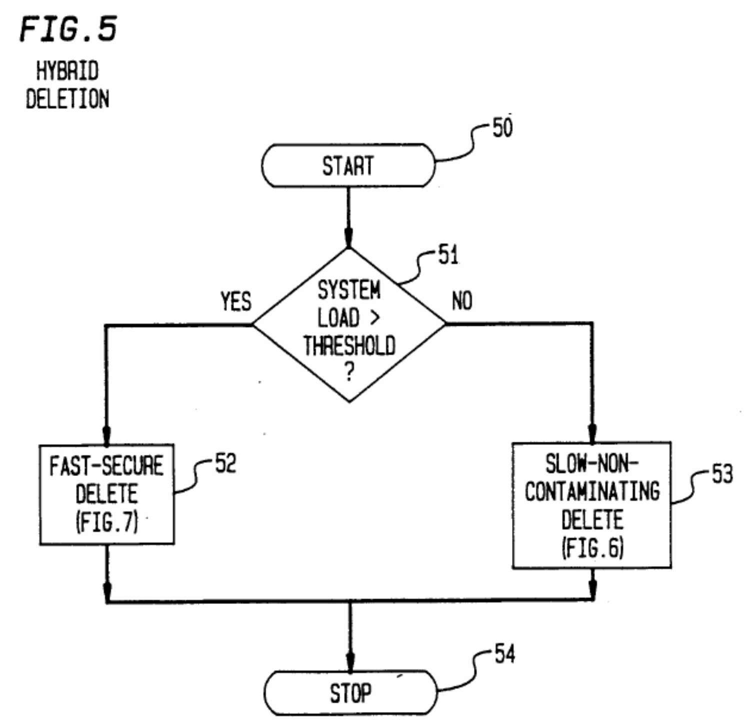
Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		Id. at 2:42-46. This hybrid deletion is shown in Figure 5.  FIG.5 HYBRID DELETION START 50 SYSTEM LOAD > THRESHOLD ? 51 YES NO FAST-SECURE DELETE (FIG.7) 52 SLOW-NON-CONTAMINATING DELETE (FIG.6) 53 STOP 54 *Id.* at Figure 5. During the hybrid deletion procedure decision block 51 checks the system load

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		**Local Area Transport Protocol ("LAT") alone and in combination**
		to determine if the system load is greater than a threshold. If the system load is greater than the threshold, then a fast-secure delete 52 is used. *Id.* at 6:40-64, Figure 5. On the other hand, if the system load is less than the threshold, then a slow-non-contaminating delete 53 is used. *Id.* The fast-secure delete 52 does not actually delete records, rather it marks records as deleted. *Id.* at 8:1-33, Figure 7. These records are then actually deleted by a subsequent slow-non-contaminating delete 53. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. Thus, the hybrid deletion procedure in the '663 patent dynamically determines a maximum number of records to remove. *See id.* at 6:40-64, Figure 5. If the fast-secure delete 52 is used, then maximum number of records is zero because records are not deleted they are only marked. *Id.* at 8:1-33, Figure 7. If the slow-non-contaminating delete 53 is used, then the maximum number of records to remove is all of the contaminated records in the bucket. *Id.* at 6:65-7:68, Figures 6, 6A, 6B. As both LAT and the '663 patent relate to deletion of records from hash tables using external chaining, one of ordinary skill in the art would understood how to use the '663 patent's dynamic decision on whether to perform a deletion based on a systems load in other hash table implementations such as that described in LAT. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the '663

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
		patent’s deletion decision procedure with LAT would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the ’663 patent’s dynamic decision on whether to perform a deletion based on a systems load as taught by the ’663 patent and with LAT and would have seen the benefits of doing so. One such benefit, for example, is that the system would avoid performing deletions when the system load exceeded a threshold. Alternatively, it would also be obvious to combine LAT with the Opportunistic Garbage Collection Articles. The Opportunistic Garbage Collection Articles disclose a generational based garbage collection which dynamically determines how much garbage to collect. *See generally*, Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*, OOPSLA ’89 Proceedings, October 1-6, 1989; Paul R. Wilson, *Opportunistic Garbage Collection*, ACM SIGPLAN Notices, Vol. 23, No. 12, December 1988. For example, the Opportunistic Garbage Collection Articles disclose in part: When a significant pause has been detected, a decision procedure is invoked to decide whether to garbage collect, and how many generations to scavenge. The fuller a generation is, the more likely it is to be scavenged; also, the longer the pause that has been detected, the larger the scope of the garbage collection is likely to be. *Design of the Opportunistic Garbage Collector* at 32.

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		Every time a user-input routine is invoked, a decision routine can decide whether to garbage collect. As long as the decision routine takes no more than a few milliseconds to execute, it should not interfere with responsiveness. Since it is only invoked at these times, it does not incur a continual run-time overhead. *Opportunistic Garbage Collection* at 100. This decision routine should take several things into account: 1) the volume of data allocated since the last scavenge, 2) how long it has been since the user has had an opportunity to interact, and 3) the height of the stack relative to its average height at reads since the last scavenge. If the product of the allocation and the compute time is high, and if the stack is low, the scavenge favorability measure is high. If it is especially high, a multi-generation scavenge is in order. *Id.* If these heuristics fail and a scavenge is forced instead by the filling of a generation's space, it is likely to happen during a significant compute-bound pause--the one that has just allocated the data that forced the collection. When the opportunistic mechanism fails to find the end of a pause, it may still succeed by default, embedding a scavenge pause within a larger pause. *Design of the Opportunistic Garbage Collector* at 32. As both LAT and the Opportunistic Garbage Collection Articles relate to deletion of aged records, one of ordinary skill in the art would have understood how to use the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion based on a system load in other hash table implementations such as LAT. Moreover, one of ordinary skill in the art would recognize that it would improve similar systems and methods in the

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		same way. As the '120 patent states "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." The '120 patent at 7:10-15. Additionally, one of ordinary skill in the art would recognize that the result of combining the Opportunistic Garbage Collection Articles' deletion decision procedure with LAT would be nothing more than the predictable use of prior art elements according to their established functions. By way of further example, one of ordinary skill in the art would have combined the Opportunistic Garbage Collection Articles' dynamic decision on whether to perform a deletion and how many generations to scavenge as taught by the Opportunistic Garbage Collection Articles and with LAT and would have seen the benefits of doing so. One such benefit, for example, is preventing slowdown of the system. Additionally, it would have been obvious to one of ordinary skill in the art to modify the system disclosed in LAT to dynamically determine the maximum number of expired records to remove in the accessed linked list of records. It is a fundamental concept in computer science and the relevant art that any variable or parameter affecting any aspect of a system can be dynamically determined based on information available to the system. One of ordinary skill in the art would have been motivated to combine the system disclosed in LAT with the fundamental concept of dynamically determining the maximum number of expired records to remove in an accessed linked list of records to solve a number of potential problems. For example, the removal of expired records described in LAT can be burdensome on the system, adding to the

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		system's load and slowing down the system's processing. Moreover, the removal could also force an interruption in real-time processing as the processing waits for the removal to complete. One of ordinary skill in the art would have known that dynamically determining the maximum number to remove would limit the burden on the system and bound the length of any real-time interruption to prevent delays in processing. Indeed, Nemes concedes that such dynamic determination was obvious when he states in the '120 patent that "[a] person skilled in the art will appreciate that the technique of removing all expired records while searching the linked list can be expanded to include techniques whereby not necessarily all expired records are removed, and that the decision regarding if and how many records to delete can be a dynamic one." '120 at 7:10-15. To the extent that dynamically determining a maximum number of expired records is not disclosed by LAT in combination with Dirks, Thatte, the '663 Patent, or the Opportunistic Garbage Collection References, it is disclosed by Linux 2.0.1, which describes dynamically determining maximum number of expired ones of the records to remove when the linked list is accessed. It would have been obvious to combine Linux 2.0.1 with LAT. For example, both Linux 2.0.1 and LAT describe systems and methods for performing data storage and retrieval using known programming techniques to yield a predictable result. When invoked, the function `rt_cache_add` automatically increments an integer variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1359. When the function `rt_cache_add` removes an expired record, the function

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
		`rt_cache_add` decrements the variable `rt_cache_size`. *See* Linux 2.0.1, route.c at line 1373. Thus, the variable `rt_cache_size` indicates the number of records in the hash table (i.e., `ip_rt_hash_table`). Because the function `rt_cache_add` automatically increments and decrements the variable `rt_cache_size`, the variable `rt_cache_size` is determined dynamically. Furthermore, Linux 2.0.1 includes the function `rt_garbage_collect_1`. The function `rt_garbage_collect_1` loops through each of the linked lists in the `ip_rt_hash_table` global variable. *See* Linux 2.0.1, route.c at lines 1122-1138. In this way, the function `rt_garbage_collect_1` accesses the linked list. When the function `rt_garbage_collect_1` identifies a record that is expired, the function `rt_garbage_collect_1` decrements the variable `rt_cache_size` and frees the record. *See* Linux 2.0.1, route.c at lines 1128-1135. Because all records in the linked list can be expired and all records in the hash table can be in the linked list, the variable `rt_cache_size` can represent a dynamically determined maximum number of expired ones of the records to remove when function `rt_garbage_collect_1` accesses the linked list. Furthermore, the function `rt_cache_add` determines whether the number of records in the hash table exceeds a predetermined threshold `RT_CACHE_SIZE_MAX`. If the number of records in the hash table exceeds the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_cache_add` invokes a function `rt_garbage_collect`. *See* Linux

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		2.0.1, route.c at lines 1341-1342. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect` invokes a function `rt_garbage_collect_1`. *See* Linux 2.0.1, route.c at line 1293. The function `rt_garbage_collect_1` loops through each linked list in the hash table. *See* Linux 2.0.1, route.c at lines 1116-1132. For each linked list in the hash table, the function `rt_garbage_collect_1` looks at each record in the linked list. *See* Linux 2.0.1, route.c at lines 1120-1131. For each record in a linked list, the function `rt_garbage_collect_1` determines whether the record's last use time plus the record's expiration factor is later than the current time. *See* Linux 2.0.1, route.c at line 1122. If the record's last use time plus the record's expiration factor is less than the current time, the function `rt_garbage_collect_1` removes the record from the linked list. *See* Linux 2.0.1, route.c at lines 1124-1130. The record's expiration factor is based on a variable `expire` and the record's reference count. *See* Linux 2.0.1, route.c at line 1122. The variable expire is initially one half of the fixed timeout value `RT_CACHE_TIMEOUT`. *See* Linux 2.0.1, route.c at line 1110. After looping through all of the linked lists in this manner, the function `rt_garbage_collect_1` determines again whether the number of records in the hash table is less than the predetermined threshold `RT_CACHE_SIZE_MAX`. *See* Linux 2.0.1, route.c at line 1133. If the number of items in the hash table is still greater than the predetermined threshold `RT_CACHE_SIZE_MAX`, the function `rt_garbage_collect_1` halves the variable `expire` and loops through each of the linked lists in the hash table. *See* Linux 2.0.1, route.c at line 1135. In this way, the function

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol ("LAT") alone and in combination
		rt_garbage_collect_1 can remove additional records from the linked lists in the hash table. The function rt_garbage_collect_1 repeats this process until the total number of records in the hash table is less than the predetermined threshold RT_CACHE_SIZE_MAX. Under Bedrock's proposed claim constructions, the records removed by the function rt_garbage_collect_1 are "expired" records. That is, the records removed by the function rt_garbage_collect_1 are data items which after a limited time or after the occurrence of some event become obsolete, such that their presence in the storage system is no longer needed or desired. The function rt_cache_add only removes a record from a linked list when the record's last use time plus the fixed timeout value RT_CACHE_TIMEOUT is less than the current time and the record's reference count is zero. *See* Linux 2.0.1, route.c at line 1369. Thus, the maximum number of records that the function rt_cache_add can remove from a given linked list is limited to those records whose reference counts are zero. In contrast, the maximum number of records that the function rt_garbage_collect_1 can remove from a given linked list is not limited to those records whose reference counts are zero. *See* Linux 2.0.1, route.c at line 1122. Rather, the function rt_garbage_collect_1 can remove records whose reference counts are zero and records whose reference counts are greater than zero. Consequently, the maximum number of records that the function rt_garbage_collect_1 can remove from a linked list is different than

EXHIBIT D-17

Asserted Claims From U.S. Pat. No. 5,893,120		Local Area Transport Protocol (“LAT”) alone and in combination
		the maximum number of records that the function `rt_cache_add` can remove from a linked list.

EXHIBIT E

ADDITIONAL PRIOR ART

I. ADDITIONAL PRIOR ART PUBLICATIONS

Author, Title, Publisher, (Publication Information, Date of Publication).
Moshe Augenstein and Aaron Tennebaum, Data Structures and pl/I Programming 536-542, 550-555, 585-604 (Prentice-Hall 1979) (QA76.9.D35 A93)
Robert J. Baron and Linda G. Shapiro, Data Structures and their Implementation 239-253, 303-314 (Von Nostrand Reinhold 1980) (QA76.9.D35 B37)
A.T. Berztiss, Data Structures Theory and Practice 316-319, 329-339 (Academic Press 1971) (QA76.6.B745)
A.T. Berztiss, Data Structures Theory and Practice 416-420, 431-460 (Academic Press 2d Edition 1975) (QA 76.6.B475 1975)
David Clark, Van Jacobson, John Romkey, and Howard Salwen, *An Analysis of TCP Processing Overhead*, IEEE COMMUNICATIONS MAGAZINE, June 1989, at p. 23-29
Luc Devroye, Lecture Notes on Bucket Algorithms 10-16 (Birkhauser Boston 1986) (QA76.9.D35 D48)
I. Ganapathy and R.F. Hobson, *GPMS, A general Purpose Memory Management System --- User's Memory --- That is*, Proceedings of the Eighth International Conference on APL, 155-165 (1976)
C.C. Gotlieb and L.R. Gotlieb, Data Types and Structures 341-346 (Prentice-Hall 1978) (QA76.9.D35 G67)
Patrick A.V. Hal, Computational Structures an Introduction to Non-numerical Computing 119-134 (Macdonald & Co. 1975) (QA76.9.D35 H34)
Ellis Horowitz and Sartaj Sahni, Fundamentals of Data Structures 456-471 (Computer Science Press 1983) (QA76.9.D35 H67 1983)
A Klinger, K.S. Fu, T.L. Kunii, Data Structures, Computer Graphics, and Pattern recognition 109-114 (Academic Press 1977) (QA 76.9.D35 D37)
Robert L. Kruse, Programming with Data Structures, Pascal Version 499-523 (Prentice-Hall 1989) (QA76.6.K774 1989)
James Richard Low, Automatic Coding: Choice of Data Structures 24,25,32(Birkhauser Verlag

EXHIBIT E

Author, Title, Publisher, (Publication Information, Date of Publication).
Basel 1976) (QA76.9.D35 L68)
Udi Manber, Introduction to Algorithms a Creative Approach 78-83 (Addison-Wesley 1989) (QA 76.9+.D35 M36 1989)
William G. McArthur and J. Winston Crawley, Structuring Data with PASCAL 604-608 (Prentice Hall 1992) (QA76.9.D35 M39 1992)
Edward M. Reingold and Wilfred J. Hansen, Data Structures in Pascal 268-277, 376-414 (Little, Brown 1986) (QA76.9.D35 R443 1986)
Edward M. Reingold and Wilfred J. Hansen, Data Structures 246-253, 332-364 (Little, Brown 1983) (QA76.9.D35 R44 1983)
M.J.R. Shave, Data Structures 94-116 (McGraw Hill 1975) (QA 76.9.D35 S47)
Jean-Paul Tremblay and Paul G. Sorenson, An Introduction to data Structures with Applications 518-524, 563-568, 611-623 (McGraw-Hill 2d Edition 1984) (QA76.9.D35 T73 1984)
Steven Wartik, Boolean Operations p.282-292 and Steven Wartik, Edward Fox, Lenwood Heath, and Qi-fan Chen, Hashing Algorithms p.293-318; both published in Information Retireval Data Structures & Algorithms, edited by William B. Frakes and Ricardo Baeza-Yates (Prentice Hall 1992) (QA76.9.D35 I543 1992)