EXHIBIT B

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC. Plaintiff, v. GOOGLE INC. Defendant.	Case No. CV 10-03561 WHA

SUPPLEMENTAL EXPERT REPORT OF JOHN C. MITCHELL REGARDING INFRINGEMENT OF THE ’205 PATENT

SUBMITTED ON BEHALF OF PLAINTIFF ORACLE AMERICA, INC.

CONFIDENTIAL PURSUANT TO PROTECTIVE ORDER
Highly Confidential – Attorneys Eyes Only

TABLE OF CONTENTS

I, John C. Mitchell, Ph.D., submit the following expert report ("Supplemental '205 Patent Infringement Report") on behalf of plaintiff Oracle America, Inc. ("Oracle"):

I. INTRODUCTION

1. In light of the Court's recent construction of the phrase "at runtime," I have been asked to supplement my opinion on whether Claims 1 and 2 of the '205 patent are infringed by Google.

2. I have detailed my retention, scope of work performed, materials relied upon, expected testimony, compensation, and qualifications in my reports submitted earlier in this case.

3. Instead of repeating the content of my earlier reports here, I incorporate them here by reference.

4. This report supplements my earlier reports with respect to Google's infringement of United States Patent No. 6,910,205 ("the '205 patent"), in light of the Court's recent construction of the phrase "at runtime," which appears in Claim 1 of the '205 patent. This report does not change my analysis or my opinion; I intend only to highlight additional evidence to show, with respect to the "inline" theory, that the claim limitation "generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction" is met, given the specificity of the Court's construction.

II. EXECUTIVE SUMMARY

5. Based on my investigation and analysis, it remains my opinion that Google, by making, distributing, and using Android, literally meets the limitations of these asserted claims, in the manner described in the Exhibits to Oracle's infringement contentions submitted to Google on April 1, 2011, in my earlier reports, in Oracle's further supplemental infringement contentions with respect to the '205 patent, and in this report.

III. CLAIM CONSTRUCTION

6. I have reviewed the Court's Claim Construction Order, dated May 9, 2011, and the Court's Supplemental Claim Construction Order, dated January 25, 2012, and have applied

the interpretation of each claim term as construed by the Court in my infringement analysis. In particular, I note the Court construed the following claim terms contained in various asserted claims of the ’205 patents:

Claim Construction Term or Phrase	**Patent**	**Excerpted Rulings from Court’s January 25, 2012 Supplemental Claim Construction Order**
at runtime	’205	Accordingly, the phrase “at runtime” shall be construed to mean “during execution of one or more virtual machine instructions.”

7. Pursuant to the parties’ Joint Claim Construction Statement, dated February 22, 2011, I further note that the parties agreed on the construction of the following terms, which I have adopted in arriving at my infringement opinions set forth in this report:

Claim Term or Phrase	**Patent**	**Agreed Construction**
function	’205	a software routine (also called a subroutine, procedure, member and method)
machine instruction	’205	an instruction that directs a computer to perform an operation specified by an operation code (OP code) and optionally one or more operands
native machine instruction / native instruction	’205	a machine instruction that is designed for a specific microprocessor or computer architecture (also called native code)
virtual machine instructions	’205	a machine instruction that is designed for a software emulated microprocessor or computer architecture (also called virtual code)

8. For all other claim terms in the ’205 patent, I have applied their plain meaning as would be understood by one of ordinary skill in the art when read in the context of the patent specification.

IV. SUPPLEMENTAL INFRINGEMENT ANALYSIS OF THE ’205 PATENT

9. On January 25, 2012, the Court construed “at runtime” in Claim 1 of the ’205 patent to mean “during execution of one or more virtual machine instructions.” The phrase “at runtime” appears in the body of Claim 1 as part of the step of “generating, at runtime, a new

virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction." The phrase does not separately appear in the body of Claim 2.

10. I have analyzed the evidence regarding Android in light of the Court's construction of "at runtime," and my opinion remains that Google infringes Claims 1 and 2 of the '205 patent in the manner that I described in my earlier infringement reports. The Court's construction does not implicate my infringement analysis with respect to the limitations of Claims 1 and 2 other than the "generating, at runtime" step. Although the phrase "at runtime" also appears in the preamble of Claim 1, the Court's construction does not affect infringement for two reasons. First, the preamble is not limiting. Preambles generally do not limit the scope of a claim; this preamble is only the intended purpose of the claimed method and is not needed to provide any support to the body of the claim. Second, the Court's construction adds little when applied to the preamble, which already included virtual machine instruction execution: "a method for increasing the execution speed of virtual machine instructions at runtime."

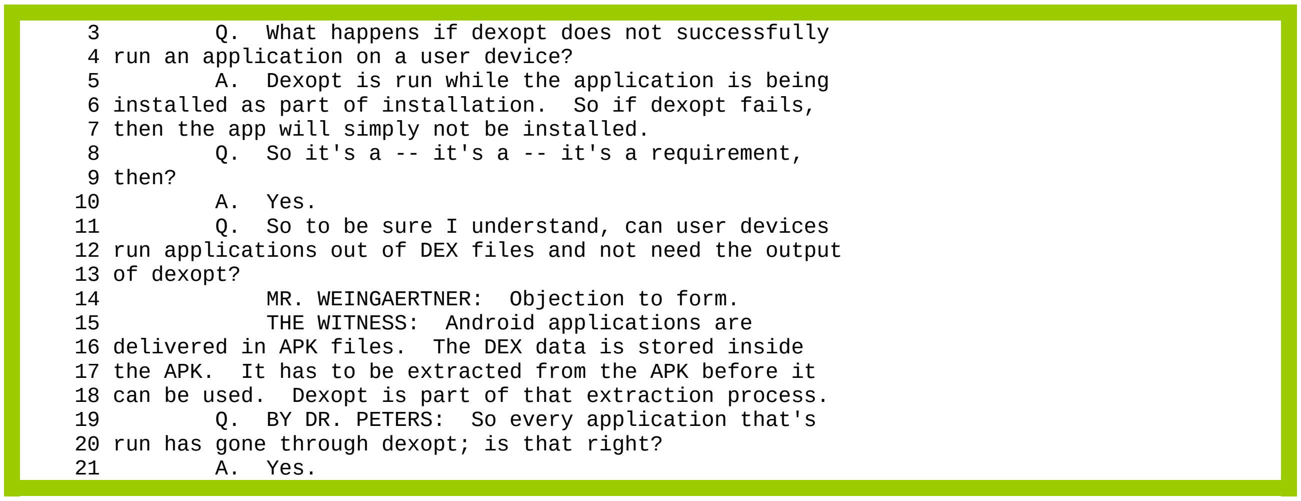
11. With respect to the "generating, at runtime" step of Claim 1, in my earlier infringement reports, I identified and discussed the Android functionality, for both the "JIT" and the "inline" infringement theories, that performs the step of "generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction." It remains my opinion that that functionality satisfies that claim limitation. In this report, I identify and discuss additional evidence pertinent to the Court's construction of "at runtime" with respect to the "inline" theory. This evidence further demonstrates that Android performs the "generating" step "at runtime."

12. The infringement evidence illustrated below is exemplary and not exhaustive. The cited examples are largely taken from Android 2.2 ("Froyo"). I understand that the publicly released versions of Android from version 2.3 ("Gingerbread") and earlier operate as I describe below; I understand that Google did not produce source code for the Honeycomb or Ice Cream Sandwich versions, so I have not had an opportunity to analyze them.

13. Android's dexopt program loads virtual machine instructions into a Dalvik virtual machine and replaces selected virtual machine instructions with new virtual machine instructions that reference or represent native code to be executed instead. (*See, e.g.*, 5/4/2011 McFadden Dep. 154:21-156:7.) In my earlier reports, I discussed how Android's dexopt, through routines such as optimizeMethod(), "generat[es], at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction," as claimed in the '205 patent.

14. The Court recently construed "at runtime" in this phrase to mean "during execution of one or more virtual machine instructions." Applying this new construction to the evidence, I find that Android's dexopt performs the "generating" step during execution of one or more virtual machine instructions, and therefore satisfies the "at runtime" limitation of Claim 1 as construed by the Court.

15. As I discussed in my earlier reports, Google's documented descriptions of dexopt show that dexopt runs at runtime. I understand that dexopt is an essential part of Android, because Android devices will only run application files that have been processed by dexopt, as Google engineer Andrew McFadden testified:



Q. What happens if dexopt does not successfully run an application on a user device?

A. Dexopt is run while the application is being installed as part of installation. So if dexopt fails, then the app will simply not be installed.

Q. So it's a -- it's a -- it's a requirement, then?

A. Yes.

Q. So to be sure I understand, can user devices run applications out of DEX files and not need the output of dexopt?

MR. WEINGAERTNER: Objection to form.

THE WITNESS: Android applications are delivered in APK files. The DEX data is stored inside the APK. It has to be extracted from the APK before it can be used. Dexopt is part of that extraction process.

Q. BY DR. PETERS: So every application that's run has gone through dexopt; is that right?

A. Yes.

(5/4/2011 McFadden Dep. 110:3-21.)

16. The Google document entitled "Dalvik Optimization and Verification With dexopt," which I quoted in my earlier reports, explains how it works:

http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html:

Dalvik Optimization and Verification With *dexopt*

The Dalvik virtual machine was designed specifically for the Android mobile platform. The target systems have little RAM, store data on slow internal flash memory, and generally have the performance characteristics of decade-old desktop systems. They also run Linux, which provides virtual memory, processes and threads, and UID-based security mechanisms.

The features and limitations caused us to focus on certain goals:

- Class data, notably bytecode, must be shared between multiple processes to minimize total system memory usage.
- The overhead in launching a new app must be minimized to keep the device responsive.
- Storing class data in individual files results in a lot of redundancy, especially with respect to strings. To conserve disk space we need to factor this out.
- Parsing class data fields adds unnecessary overhead during class loading. Accessing data values (e.g. integers and strings) directly as C types is better.
- Bytecode verification is necessary, but slow, so we want to verify as much as possible outside app execution.
- Bytecode optimization (quickened instructions, method pruning) is important for speed and battery life.
- For security reasons, processes may not edit shared code.

The typical VM implementation uncompresses individual classes from a compressed archive and stores them on the heap. This implies a separate copy of each class in every process, and slows application startup because the code must be uncompressed (or at least read off disk in many small pieces). On the other hand, having the bytecode on the local heap makes it easy to rewrite instructions on first use, facilitating a number of different optimizations.

The goals led us to make some fundamental decisions:

- Multiple classes are aggregated into a single "DEX" file.
- DEX files are mapped read-only and shared between processes.
- Byte ordering and word alignment are adjusted to suit the local system.
- Bytecode verification is mandatory for all classes, but we want to "pre-verify" whatever we can.
- Optimizations that require rewriting bytecode must be done ahead of time.
- The consequences of these decisions are explained in the following sections.

….
dexopt

We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in.

The solution is to invoke a program called dexopt, which is really just a back door into the VM. It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap class path, and then sets about verifying and optimizing whatever it can from the target DEX. On completion, the process exits, freeing all resources.

It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once.
….
Optimization

Virtual machine interpreters typically perform certain optimizations the first time a piece of code is used. Constant pool references are replaced with pointers to internal data structures, operations that always succeed or always work a certain way are replaced with simpler forms. Some of these require information only available at runtime, others can be inferred statically when certain assumptions are made.

The Dalvik optimizer does the following:

- For virtual method calls, replace the method index with a vtable index.
- For instance field get/put, replace the field index with a byte offset. Also, merge the boolean / byte / char / short variants into a single 32-bit form (less code in the interpreter means more room in the CPU I-cache).
- Replace a handful of high-volume calls, like String.length(), with "inline" replacements. This skips the usual method call overhead, directly switching from the interpreter to a native implementation.
- Prune empty methods. The simplest example is Object.<init>, which does nothing, but must be called whenever any object is allocated. The instruction is replaced with a new version that acts as a no-op unless a debugger is attached.
- Append pre-computed data. For example, the VM wants to have a hash table for lookups on class name. Instead of computing this when the DEX file is loaded, we can compute it now, saving heap space and computation time in every VM where the DEX is loaded.

All of the instruction modifications involve replacing the opcode with one not defined by the Dalvik specification. This allows us to freely mix optimized and unoptimized instructions. The set of optimized instructions, and their exact representation, is tied closely to the VM version.

Most of the optimizations are obvious "wins". The use of raw indices and offsets not only allows us to execute more quickly, we can also skip the initial symbolic resolution. Pre-computation eats up disk space, and so must be done in moderation.

There are a couple of potential sources of trouble with these optimizations. First, vtable indices and byte offsets are subject to change if the VM is updated. Second, if a superclass is in a different DEX, and that other DEX is updated, we need to ensure that our optimized indices and offsets are updated as well. A similar but more subtle problem emerges when user-defined class loaders are employed: the class we actually call may not be the one we expected to call.

These problems are addressed with dependency lists and some limitations on what can be optimized.

17. From Google's description above, it is apparent that the "generating, at runtime" step of Claim 1 is met. Dexopt replaces a handful of high-volume calls, like String.length(), with native implementation, which allows a direct switch from the interpreter to the native code. This

optimization involves replacing the incoming opcode (a first virtual machine instruction) with one not defined by the Dalvik specification (a “new” virtual machine instruction) that directs execution of the native implementation, and the optimization requires information only available at runtime.

18. Here, I consider additional evidence that dexopt runs “at runtime” in light of the Court’s construction.

19. The Google documentation I mentioned above further explains the circumstances of how dexopt is run:

http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html:

Preparation

There are at least three different ways to create a "prepared" DEX file, sometimes known as "ODEX" (for Optimized DEX):

1. The VM does it "just in time". The output goes into a special `dalvik-cache` directory. This works on the desktop and engineering-only device builds where the permissions on the `dalvik-cache` directory are not restricted. On production devices, this is not allowed.

2. The system installer does it when an application is first added. It has the privileges required to write to `dalvik-cache`.

3. The build system does it ahead of time. The relevant `jar` / `apk` files are present, but the `classes.dex` is stripped out. The optimized DEX is stored next to the original zip archive, not in `dalvik-cache`, and is part of the system image.

* * *

Preparation of the DEX file for the "just in time" and "system installer" approaches proceeds in three steps:

First, the dalvik-cache file is created. This must be done in a process with appropriate privileges, so for the "system installer" case this is done within installd, which runs as root.

20. Here, I focus on the second approach—the “system installer” approach. According to Google’s documentation, when an application is downloaded to an Android device, the system installer creates an “optimized DEX” file in the dalvik-cache directory by running dexopt, which verifies and optimizes all of the classes in the DEX file to the extent possible.

Creation of the "Optimized DEX" file in the dalvik-cache "must be done in a process with appropriate privileges, so for the 'system installer' case this is done within installd, which runs as root." (*Id.*)

21. Accordingly, to supplement my infringement analysis, I examined the Android source code involved in application installation. The examples below are taken from the Froyo version of Android.

22. In a running Android system, the PackageManagerService is responsible for installing applications and communicating with installd. The Java source code defining the PackageManagerService class is found in PackageManagerService.java.

23. In the PackageManagerService class, the processPendingInstall() method invokes installPackageLI(), which invokes either installNewPackageLI() or replacePackageLI(). replacePackageLI() invokes either replaceNonSystemPackageLI() or replaceSystemPackageLI(). All three of the installNewPackageLI(), replaceNonSystemPackageLI(), and replaceSystemPackageLI() methods invoke scanPackageLI(), which invokes the performDexOptLI() method.

24. To run dexopt (if it has not already been run before or otherwise is forced to run), the performDexOptLI() method of the PackageManagerService class invokes the dexopt() method of the Installer class. The performDexOptLI() method from PackageManagerService.java is reproduced here (note that mInstaller is an instance of the Installer class):

```
private int performDexOptLI(PackageParser.Package pkg, boolean forceDex) {
        boolean performed = false;
        if ((pkg.applicationInfo.flags&ApplicationInfo.FLAG_HAS_CODE) != 0 && mInstaller !=
null) {
            String path = pkg.mScanPath;
            int ret = 0;
            try {
                if (forceDex || dalvik.system.DexFile.isDexOptNeeded(path)) {
                    ret = mInstaller.dexopt(path, pkg.applicationInfo.uid,
                            !isForwardLocked(pkg));
                    pkg.mDidDexOpt = true;
                    performed = true;
                }
            } catch (FileNotFoundException e) {
                Slog.w(TAG, "Apk not found for dexopt: " + path);
                ret = -1;
            } catch (IOException e) {
                Slog.w(TAG, "IOException reading apk: " + path, e);
                ret = -1;
            } catch (dalvik.system.StaleDexCacheError e) {
                Slog.w(TAG, "StaleDexCacheError when reading apk: " + path, e);
                ret = -1;
            } catch (Exception e) {
                Slog.w(TAG, "Exception when doing dexopt : ", e);
                ret = -1;
            }
            if (ret < 0) {
                //error from installer
                return DEX_OPT_FAILED;
            }
        }

        return performed ? DEX_OPT_PERFORMED : DEX_OPT_SKIPPED;
    }
```

25. performDexOptLI() catches any execeptions thrown during the execution of Installer.dexopt() and returns with an error code of -1. If Installer.dexopt() returns with an error, performDexOptLI() returns with the error code DEX_OPT_FAILED. If Installer.dexopt() returns successfully, performDexOptLI() returns with the success code DEX_OPT_PERFORMED.

26. The Installer class communicates with the installer daemon installd. (The code for installd may be found in the directory \frameworks\base\cmds\installd.) The Java code relevant to dexopt from Installer.java is reproduced here:

```
    private synchronized String transaction(String cmd) {
       if (!connect()) {
             Slog.e(TAG, "connection failed");
             return "-1";
         }

         if (!writeCommand(cmd)) {
                 /* If installd died and restarted in the background
                  * (unlikely but possible) we'll fail on the next
                  * write (this one).  Try to reconnect and write
                  * the command one more time before giving up.
                  */
             Slog.e(TAG, "write command failed? reconnect!");
             if (!connect() || !writeCommand(cmd)) {
                 return "-1";
             }
         }
//        Slog.i(TAG,"send: '"+cmd+"'");
       if (readReply()) {
             String s = new String(buf, 0, buflen);
//            Slog.i(TAG,"recv: '"+s+"'");
           return s;
       } else {
//            Slog.i(TAG,"fail");
           return "-1";
       }
    }
```

```
     private int execute(String cmd) {
        String res = transaction(cmd);
        try {
            return Integer.parseInt(res);
        } catch (NumberFormatException ex) {
            return -1;
        }
     }

   * * *

       public int dexopt(String apkPath, int uid, boolean isPublic) {
           StringBuilder builder = new StringBuilder("dexopt");
           builder.append(' ');
           builder.append(apkPath);
           builder.append(' ');
           builder.append(uid);
           builder.append(isPublic ? " 1" : " 0");
           return execute(builder.toString());
       }
```

27. Installer's dexopt() method builds a command to run dexopt on the application file and sends that command to installd by invoking the execute() method, which invokes the transaction() method. The transaction() method sends the command to run dexopt to installd, then waits for a reply from installd about success or failure of dexopt. The return value of Installer's dexopt() method (which is checked by performDexOptLI()) indicates that success or failure.

28. When installd receives a command to run dexopt, it confirms that it has the correct number of arguments, then calls the function "dexopt," which is defined in commands.c. This function performs various checks, opens the file to be optimized, opens the destination cache file, then calls fork() to create a child process. The parent process waits for the child to exit. The child process calls the function "run_dexopt," which runs the dexopt executable (/system/bin/dexopt) by calling the execl() subroutine with appropriate arguments (which include the "--zip" argument).

29. The main() entry point for dexopt is defined in OptMain.c:

```
/*
 * Main entry point.  Decide where to go.
 */
int main(int argc, char* const argv[])
{
    set_process_name("dexopt");

    setvbuf(stdout, NULL, _IONBF, 0);
```

```
        if (argc > 1) {
            if (strcmp(argv[1], "--zip") == 0)
                return fromZip(argc, argv);
            else if (strcmp(argv[1], "--dex") == 0)
                return fromDex(argc, argv);
        }

        fprintf(stderr, "Usage: don't use this\n");
        return 1;
    }
```

30. When dexopt is invoked by installd, it will call fromZip(), because "--zip" is the second argument. fromZip() is defined in OptMain.c. After performing various checks, fromZip() calls extractAndProcessZip(). After extractAndProcessZip() creates a DEX optimization header and then extracts the DEX data into the cache file, it calls dvmContinueOptimization() to "do the optimization":

```
        /* do the optimization */
        if (!dvmContinueOptimization(cacheFd, dexOffset, uncompLen, debugFileName,
                modWhen, crc32, isBootstrap))
        {
            LOGE("Optimization failed\n");
            goto bail;
        }
```

(*See* OptMain.c)

31. dvmContinueOptimization() calls rewriteDex(), which in turn calls loadAllClasses(), which loads the classes in the DEX file. Once the classes are loaded, rewriteDex() calls optimizeLoadedClasses() to optimize them. In Froyo, these routines are defined in DexOptimize.c.

32. In my earlier report, I described how dexopt's optimizeLoadedClasses() routine (which results in calls to createInlineSubsTable(), optimizeClass(), and optimizeMethod()) generates new virtual machine instructions that represent or refer to native instructions to be executed instead of the original virtual machine instructions, and how the original instructions are overwritten with the new instructions.

33. If there were no errors in the optimizeLoadedClasses() routine or otherwise, rewriteDex() returns successfully to dvmContinueOptimization(). dvmContinueOptimization() then takes care of some housekeeping matters, such as ensuring the optimized DEX file is

written to storage, with a correct header and dependency information, and then returns to extractAndProcessZip(). If dvmContinueOptimization() returned successfully, extractAndProcessZip() will return successfully to fromZip(), which will cause dexopt to return successfully.

34. As discussed above, the dexopt executable (/system/bin/dexopt) ran in a child process of installd. The parent process has been waiting for the child process to finish (by calling the wait_dexopt() function), and checks its return status once it has. If the dexopt executable finished successfully, the installd dexopt function returns successfully. installd will then send the result code to Installer.

35. Installer's transaction() method has been waiting for installd to send a reply containing the result code for the running of dexopt, as shown in Installer.java:

```
     private synchronized String transaction(String cmd) {
        if (!connect()) {
              Slog.e(TAG, "connection failed");
              return "-1";
          }

          if (!writeCommand(cmd)) {
                  /* If installd died and restarted in the background
                   * (unlikely but possible) we'll fail on the next
                   * write (this one).  Try to reconnect and write
                   * the command one more time before giving up.
                   */
              Slog.e(TAG, "write command failed? reconnect!");
              if (!connect() || !writeCommand(cmd)) {
                  return "-1";
              }
          }
//         Slog.i(TAG,"send: '"+cmd+"'");
        if (readReply()) {
              String s = new String(buf, 0, buflen);
//              Slog.i(TAG,"recv: '"+s+"'");
            return s;
        } else {
//              Slog.i(TAG,"fail");
            return "-1";
        }
     }

     private int execute(String cmd) {
        String res = transaction(cmd);
        try {
            return Integer.parseInt(res);
        } catch (NumberFormatException ex) {
            return -1;
        }
     }
. . .
      public int dexopt(String apkPath, int uid, boolean isPublic) {
          StringBuilder builder = new StringBuilder("dexopt");
          builder.append(' ');
```

```
            builder.append(apkPath);
            builder.append(' ');
            builder.append(uid);
            builder.append(isPublic ? " 1" : " 0");
            return execute(builder.toString());
        }
```

36. Installer's dexopt() method thus returns to the performDexOptLI() method of the PackageManagerService class a value that indicates the success or failure of the dexopt executable running on the application file being installed.

37. As discussed above, it was the performDexOptLI() method of the PackageManagerService class that invoked the dexopt() method of the Installer class. The performDexOptLI() method is reproduced again here (note that mInstaller is an instance of the Installer class):

```
    private int performDexOptLI(PackageParser.Package pkg, boolean forceDex) {
            boolean performed = false;
            if ((pkg.applicationInfo.flags&ApplicationInfo.FLAG_HAS_CODE) != 0 && mInstaller !=
    null) {
                String path = pkg.mScanPath;
                int ret = 0;
                try {
                    if (forceDex || dalvik.system.DexFile.isDexOptNeeded(path)) {
                        ret = mInstaller.dexopt(path, pkg.applicationInfo.uid,
                                !isForwardLocked(pkg));
                        pkg.mDidDexOpt = true;
                        performed = true;
                    }
                } catch (FileNotFoundException e) {
                    Slog.w(TAG, "Apk not found for dexopt: " + path);
                    ret = -1;
                } catch (IOException e) {
                    Slog.w(TAG, "IOException reading apk: " + path, e);
                    ret = -1;
                } catch (dalvik.system.StaleDexCacheError e) {
                    Slog.w(TAG, "StaleDexCacheError when reading apk: " + path, e);
                    ret = -1;
                } catch (Exception e) {
                    Slog.w(TAG, "Exception when doing dexopt : ", e);
                    ret = -1;
                }
                if (ret < 0) {
                    //error from installer
                    return DEX_OPT_FAILED;
                }
            }

            return performed ? DEX_OPT_PERFORMED : DEX_OPT_SKIPPED;
        }
```

(*See* PackageManagerService.java)

38. performDexOptLI() catches any execeptions thrown during the execution of Installer.dexopt() and returns with an error code of -1 if there were any. If Installer.dexopt() returns with a value indicating an error, performDexOptLI() returns with the error code DEX_OPT_FAILED. If Installer.dexopt() returns successfully after running dexopt, performDexOptLI() returns with the success code DEX_OPT_PERFORMED.

39. It is clear from the Java source code for the performDexOptLI() method of the PackageManagerService class and for the dexopt() method of the Installer class (both shown above), that the entire dexopt process runs during the execution of the performDexOptLI() and dexopt() methods. Because the PackageManagerService and Installer classes are written in the Java programming language, they are compiled to virtual machine instructions for execution on Android devices, rather than native instructions. The entire dexopt process, including the generation of new virtual machine instructions that represent or reference native instructions that can be executed instead of virtual machine instructions, occurs during the execution of the virtual machine instructions comprising the PackageManagerService.performDexOptLI() and the Installer.dexopt() methods. Thus the "at runtime" limitation, as construed by the Court, is met.

40. Android engineers confirmed that dexopt runs at runtime. For example, dexopt calls createInlineSubsTable() before calling optimizeClass() and optimizeMethod(). createInlineSubsTable() depends on gDvmInlineOpsTable[], which is defined in the source code file InlineNative.c. An Android's programmer's opening comment to that file confirms my conclusion that dexopt's "native inlining" functionality runs at runtime:

```
/*
 * Inlined native functions.  These definitions replace interpreted or
 * native implementations at runtime; "intrinsic" might be a better word.
 */
#include "Dalvik.h"
```

41. Another example is Google engineer Ben Cheng, who, when asked by customer HTC "[w]hy do we need to do this [dexopt] in runtime? Couldn't it be done in compile time?", wrote back: "What you are seeing is the normal behavior" and referred to the dexopt documentation quoted above to provide "a high-level idea of why some of these optimizations

can only be performed at runtime." (*See* October 24, 2008 email from Ben Change to Kant Kang, GOOGLE-03-00434010.)

42. My discussion above referred to the Froyo version of Android. As I discussed in my earlier report, the same essential process flow is followed in other versions of Android. In Gingerbread, the functionality of DexOptimize.c was reorganized and split between dalvik\vm\analysis\DexPrepare.c and dalvik\vm\analysis\Optimize.c. For example, Optimize.c contains dvmOptimizeClass() and optimizeMethod(); DexPrepare.c contains rewriteDex(), loadAllClasses(), verifyAndOptimizeClasses(), and verifyAndOptimizeClass(). In Cupcake, Donut, and Éclair, the call to optimizeMethod() generates an EXECUTE_INLINE instruction rather than an EXECUTE_INLINE_RANGE instruction, but with the same purpose and effect; the instructions are the same from the perspective of the '205 patent. In all versions of Android that I have examined, optimizeMethod() is called to generate new virtual machine instructions and overwrite original ones during execution of virtual machine instructions, just as it is in Froyo.

V. CONCLUSION

43. It remains my opinion that Android satisfies the "generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction" limitation of Claim 1 of United States Patent No. 6,910,205, under both the "JIT" and "inline" theories of infringement, and that Google infringes Claims 1 and 2 of the '205 patent for the reasons described above and in my earlier reports.

Dated: February 24, 2012

[illegible]

John C. Mitchell