Oracle America, Inc. v. Google Inc. Doc. 97

1 DONALD F. ZIMMER, JR. (SBN 112279)
fzimmer@kslaw.com
2 CHERYL A. SABNIS (SBN 224323)
csabnis@kslaw.com
3 KING & SPALDING LLP
101 Second Street - Suite 2300
4 San Francisco, CA 94105
Telephone: (415) 318-1200
5 Facsimile: (415) 318-1300

IAN C. BALLON (SBN 141819)
ballon@gtlaw.com
HEATHER MEEKER (SBN 172148)
meekerh@gtlaw.com
GREENBERG TRAURIG, LLP
1900 University Avenue
East Palo Alto, CA 94303
Telephone: (650) 328-8500
Facsimile: (650) 328-8508

6
7 SCOTT T. WEINGAERTNER (*Pro Hac Vice*)
sweingaertner@kslaw.com
8 ROBERT F. PERRY
rperry@kslaw.com
9 BRUCE W. BABER (*Pro Hac Vice*)
10 bbaber@kslaw.com
KING & SPALDING LLP
11 1185 Avenue of the Americas
New York, NY 10036-4003
12 Telephone: (212) 556-2100
Facsimile: (212) 556-2222
13

14 Attorneys for Defendant
GOOGLE INC.
15

UNITED STATES DISTRICT COURT
16
NORTHERN DISTRICT OF CALIFORNIA
17
SAN FRANCISCO DIVISION
18

19 ORACLE AMERICA, INC.

20 Plaintiff,

21 v.

22 GOOGLE INC.

23 Defendant.
24
25

Case No. 3:10-cv-03561-WHA

DECLARATION OF TRUMAN FENTON

Dept.: Courtroom 9, 19th Floor

Judge: Honorable William Alsup

Tutorial: April 6, 2011, 1:30 p.m.

Hearing: April 20, 2011, 1:30 p.m.

26
27
28

Dockets.Justia.com

I, Truman Fenton, hereby declare and state as follows:

1. I am an attorney with the law firm of King & Spalding LLP, which is counsel of record for Google Inc. I have personal knowledge of the facts set forth in this declaration unless otherwise noted, and, if called to do so, I could and would competently testify thereto.

2. Exhibit A is a true and correct copy of U.S. Patent No. 7,213,240 obtained from an online patent database.

3. Exhibit B-1 is a true and correct copy of pages excerpted from the uncertified file history of U.S. Patent No. 5,367,685 that was produced by Oracle America, Inc. ("Oracle") with document numbers OAGOOGLE0000057167–254.

4. Exhibit B-2 is a true and correct copy of pages excerpted from the uncertified file history of U.S. Patent No. RE36,204 that was produced by Oracle with document numbers OAGOOGLE0000059190–351.

5. Exhibit B-3 is a true and correct copy of pages excerpted from the uncertified file history of U.S. Patent No. RE38,104 that was produced by Oracle with document numbers OAGOOGLE0000059352–570.

6. Exhibit B-4 is a true and correct copy of pages excerpted from the uncertified file history of U.S. Patent No. 6,061,520 that was produced by Oracle with document numbers OAGOOGLE0000057445–662.

7. Exhibit C is a table I prepared citing portions of patents issued to Sun Microsystems based on applications filed around the same time as the patents in suit. The quotations are of the text of the patents available at the U.S. Patent and Trademark Office's website.

8. Exhibit D is a true and correct copy of a letter published in the U.S. Patent and Trademark Office Official Gazette and obtained from that website at the following web address:

http://www.uspto.gov/web/offices/com/sol/og/2010/week08/TOC.htm.

9. Exhibit E is a true and correct copy of an article written by R. Nigel Horspool and Jason Corless that was obtained from Pennsylvania State University's online research library, CiteSeer.

10. Exhibit F is document I prepared containing a partial summary of the prosecution history of U.S. Patent No. RE38,104 (and its family). The portions summarized identify the earliest references in each application of the family in which the term "computer-readable medium" was used.

I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct and that this declaration was executed this 17th day of March, 2011, in Austin, Texas.

Dated: March 17, 2011

Truman Fenton

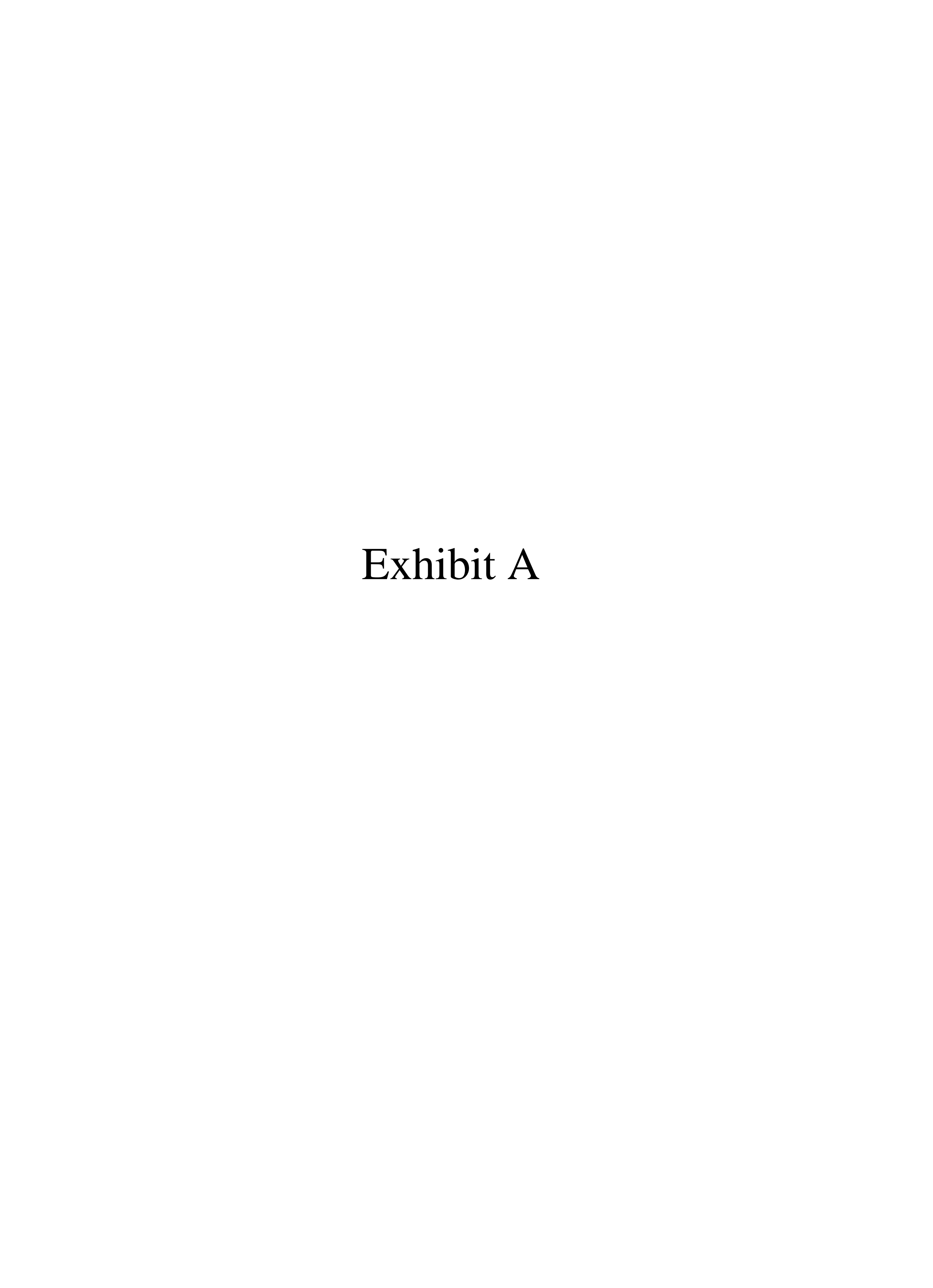
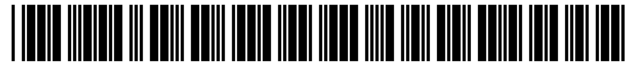


Exhibit A



(12) **United States Patent**
Wong et al.

(10) **Patent No.: US 7,213,240 B2**
(45) **Date of Patent: May 1, 2007**

(54) **PLATFORM-INDEPENDENT SELECTIVE AHEAD-OF-TIME COMPILATION**

(75) Inventors: **Hinkmond Wong**, Sunnyvale, CA (US); **Nedim Fresko**, San Francisco, CA (US); **Mark Lam**, Milpitas, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Menlo Park, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 348 days.

(21) Appl. No.: **09/970,661**

(22) Filed: **Oct. 5, 2001**

(65) **Prior Publication Data**

US 2003/0070161 A1 Apr. 10, 2003

(51) **Int. Cl.**
G06F 9/45 (2006.01)

(52) **U.S. Cl.** **717/148**

(58) **Field of Classification Search** 717/139, 717/146–148, 151, 153, 154; 706/52
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,638,298 A 1/1987 Spiro
5,276,881 A 1/1994 Chan et al.
5,280,613 A 1/1994 Chan et al.
5,339,419 A 8/1994 Chan et al.
5,812,854 A * 9/1998 Steinmetz et al. 717/159
5,920,720 A * 7/1999 Toutonghi et al. 717/148
5,966,542 A 10/1999 Tock
5,966,702 A 10/1999 Fresko et al.
6,081,665 A * 6/2000 Nilsen et al. 717/116
6,110,226 A 8/2000 Bothner
6,158,048 A 12/2000 Luch et al.
6,289,506 B1 * 9/2001 Kwong et al. 717/148

OTHER PUBLICATIONS

Andrew P. Black, "Supporting Distributed Applications: Experience with Eden", Department of Computer Science, University of Washington, Technical Report 85-03-02, Mar. 1985, pp. 1-21.
Andrew P. Black, "The Eden Programming Language", Department of Computer Science, FR-35, University of Washington, Technical Report 85-09-01, Sep. 1985 (Revised, Dec. 1985), pp. 1-19.

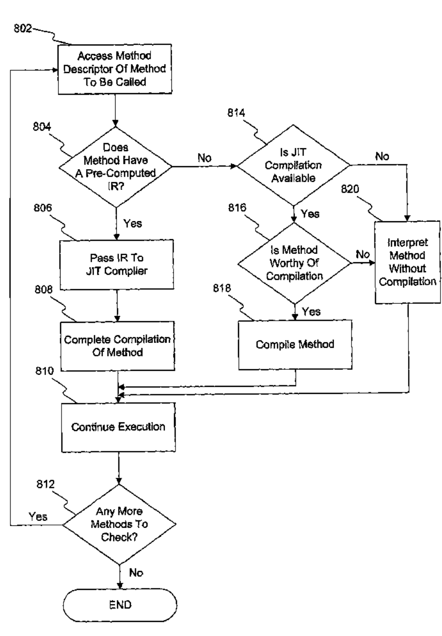
(Continued)

Primary Examiner—Tuan Dam
Assistant Examiner—Andre R Fowlkes
(74) *Attorney, Agent, or Firm*—Finnegan, Henderson, Farabow, Garrett & Dunner LLP

(57) **ABSTRACT**

Methods and systems for platform-independent selective ahead-of-time compilation are herein described. A method selector comprising a profiling tool and heuristic selects a subset of methods for ahead-of-time compilation. The profiling tool ranks a set of methods according to predetermined criteria, and the heuristic identifies the subset of methods from the set of methods. An ahead-of-time compiler comprises a first unit and a second unit. The first unit converts, for each selected method, bytecodes corresponding to the selected method to a platform-independent intermediate representation. The second unit optimizes the platform-independent intermediate representation of each selected method, wherein each optimized intermediate representation is stored with a corresponding selected method. A virtual machine on a device converts an optimized intermediate representation associated with a selected method loaded onto the device to platform-dependent machine code.

60 Claims, 10 Drawing Sheets



OTHER PUBLICATIONS

Andrew P. Black, "The Eden Project: Overview and Experiences", Department of Computer Science, University of Washington, EUUG, Autumn '86 Conference Proceedings, Manchester, UK, Sep. 22-25, 1986, pp. 177-189.

Andrew P. Black, Edward D. Lazowska, Jerre D. Noe and Jan Sanislo, "The Eden Project: A Final Report", Department of Computer Science, University of Washington, Technical Report 86-11-01, Nov. 1986, pp. 1-28.

Calton Pu, "Replication and Nested Transactions in the Eden Distributed System", Doctoral Dissertation, University of Washington, Aug. 6, 1986, pp. 1-179 (1 page Vita).

USPTO Office Action mailed Aug. 16, 2004 in related U.S. Appl. No. 10/455,341.

* cited by examiner

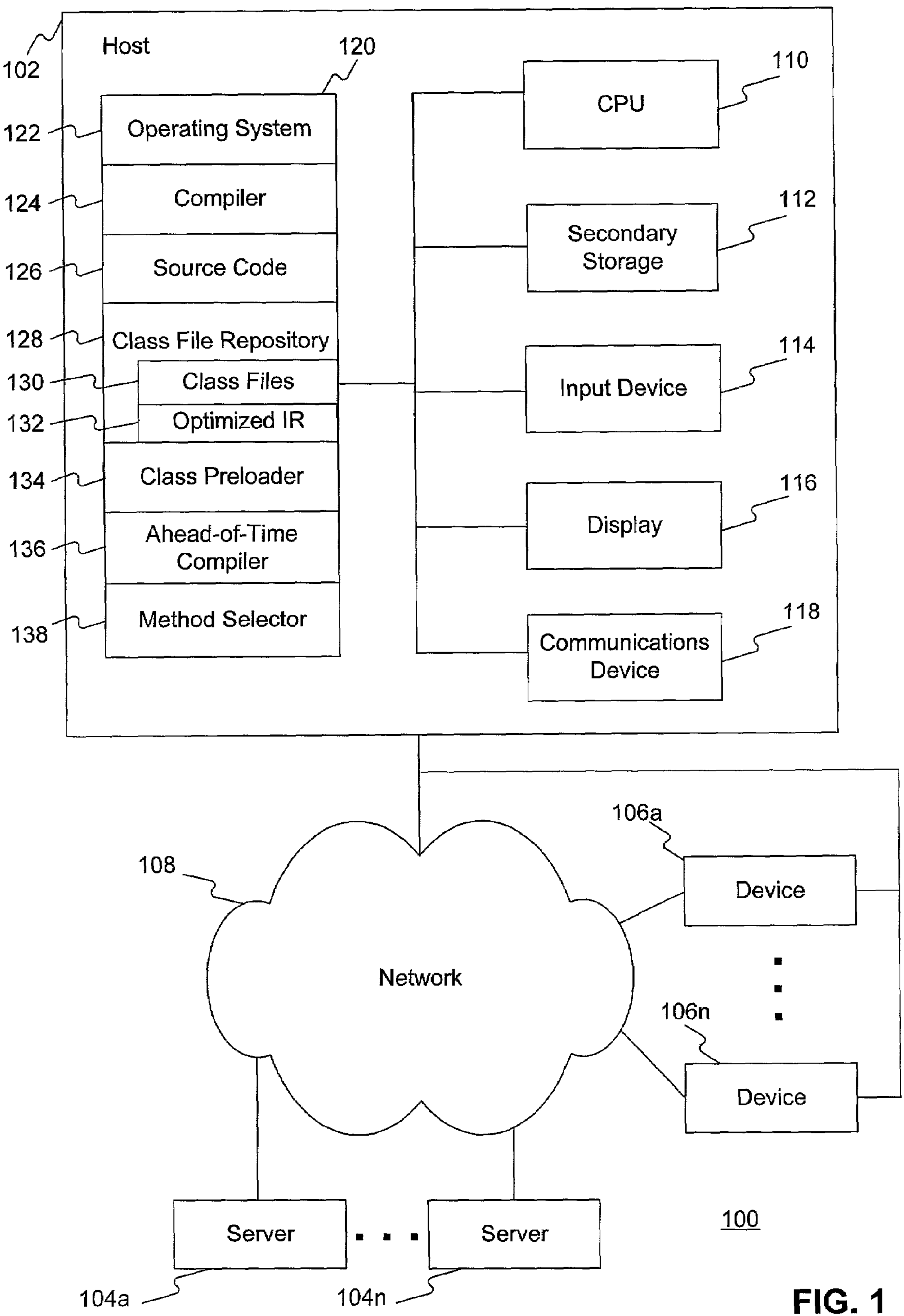


FIG. 1

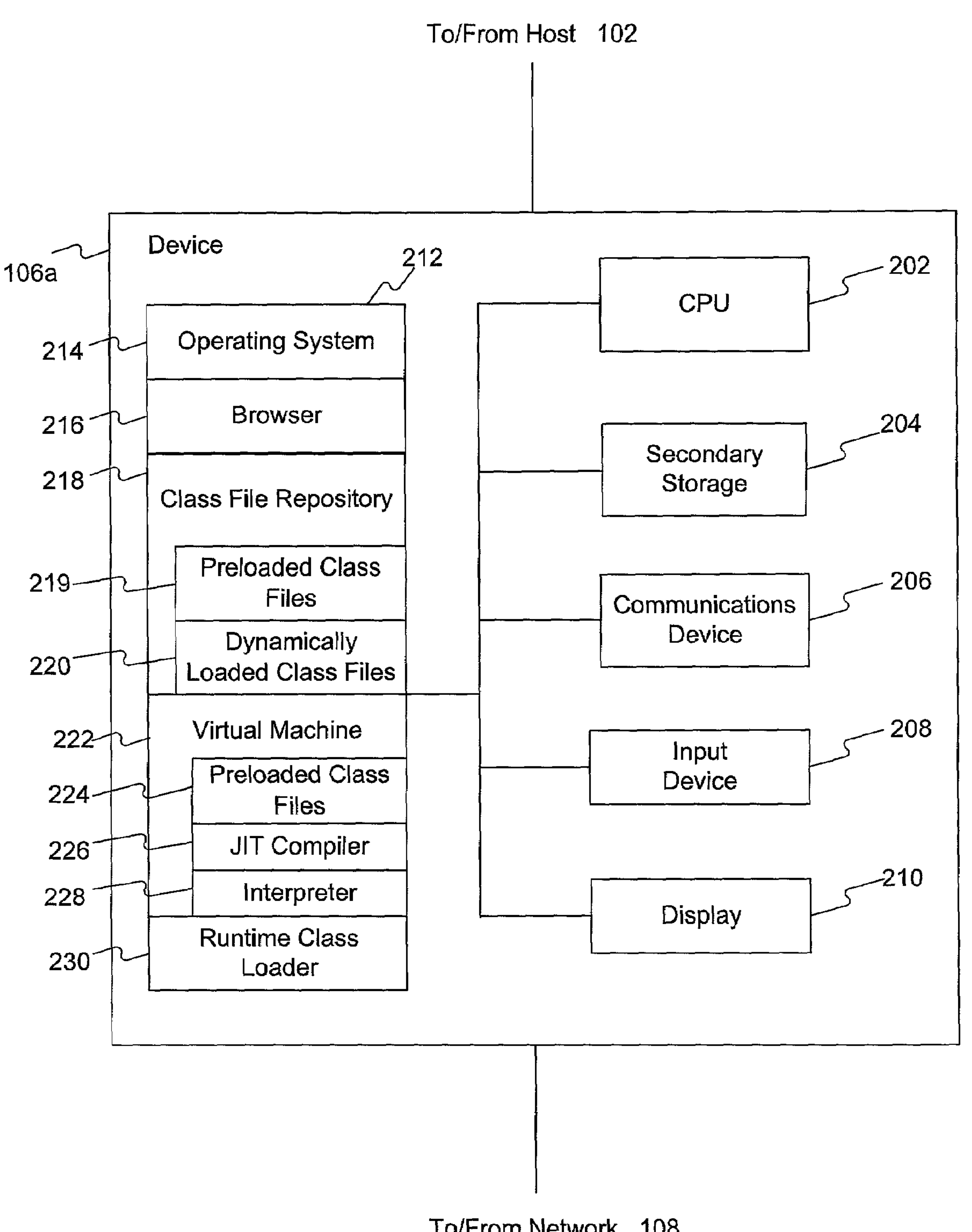


FIG. 2A

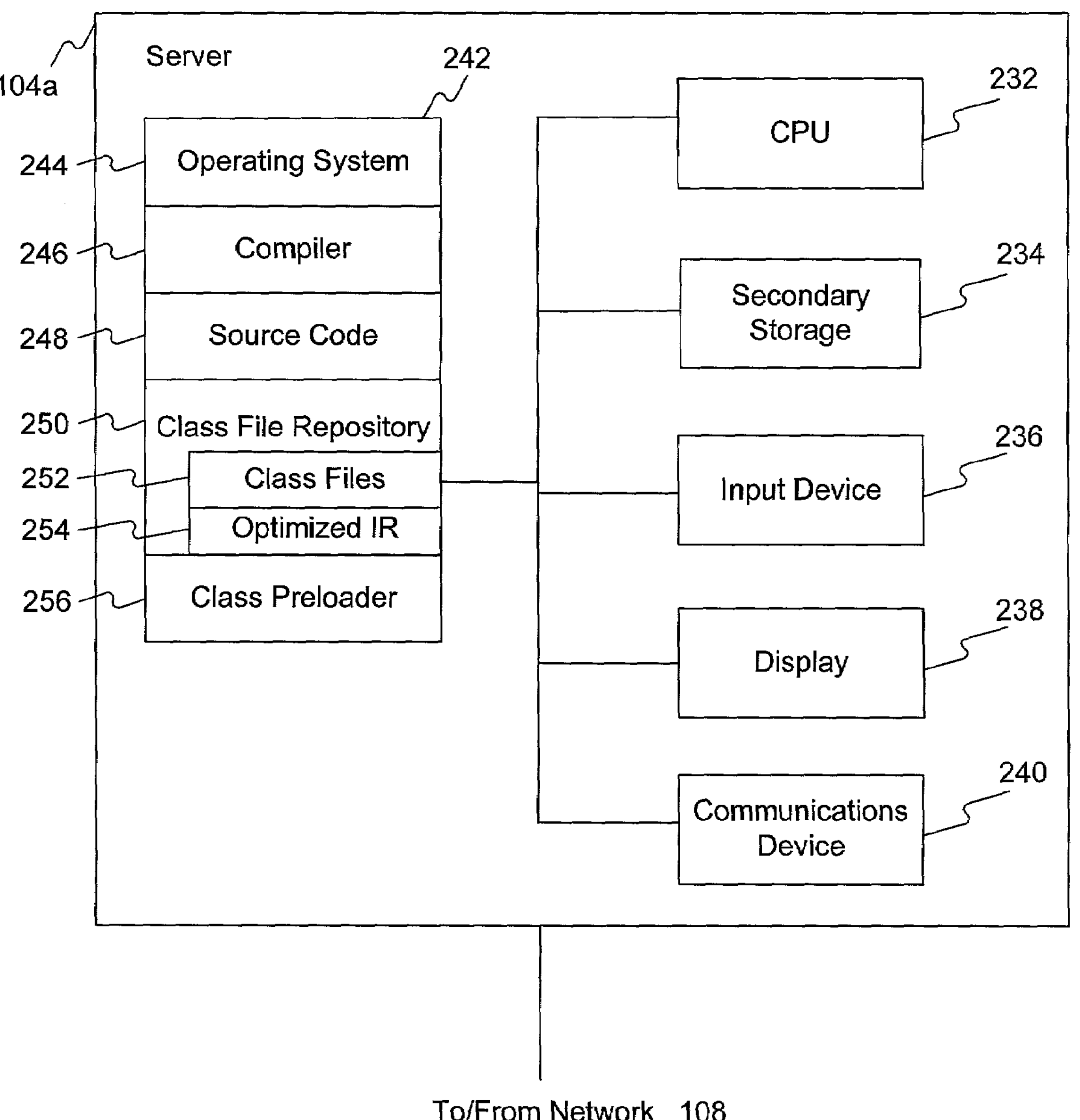


FIG. 2B

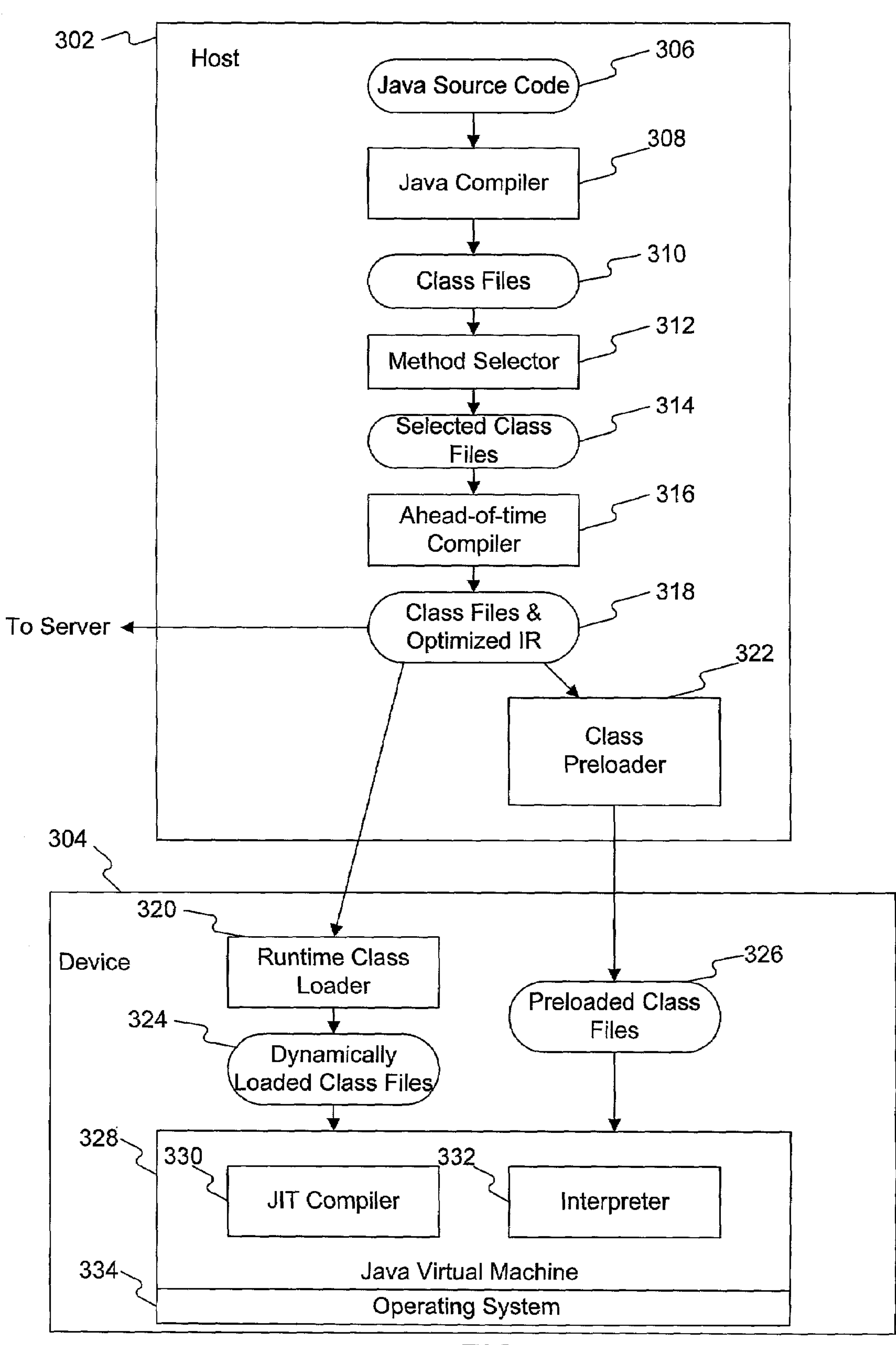


FIG. 3

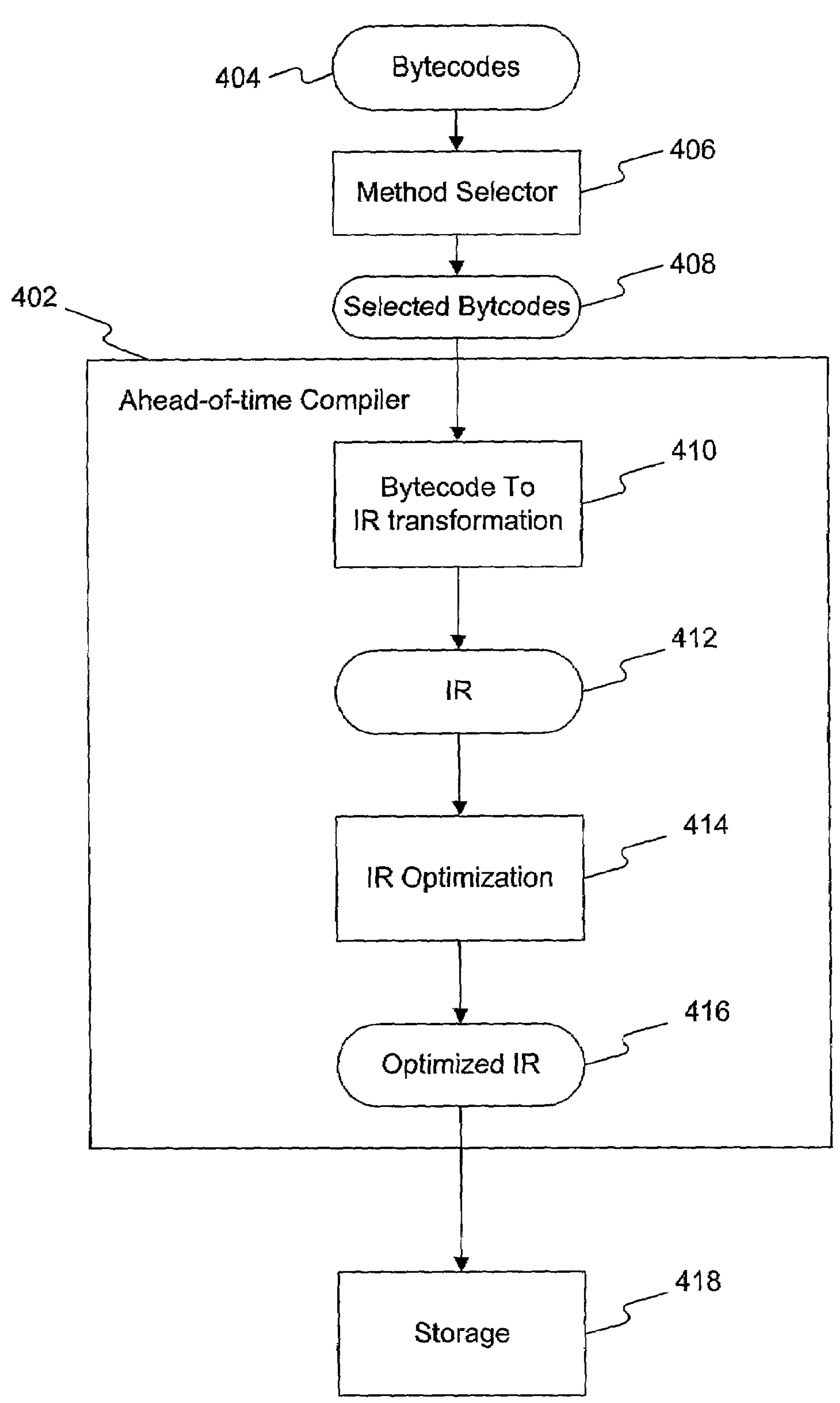


FIG. 4A

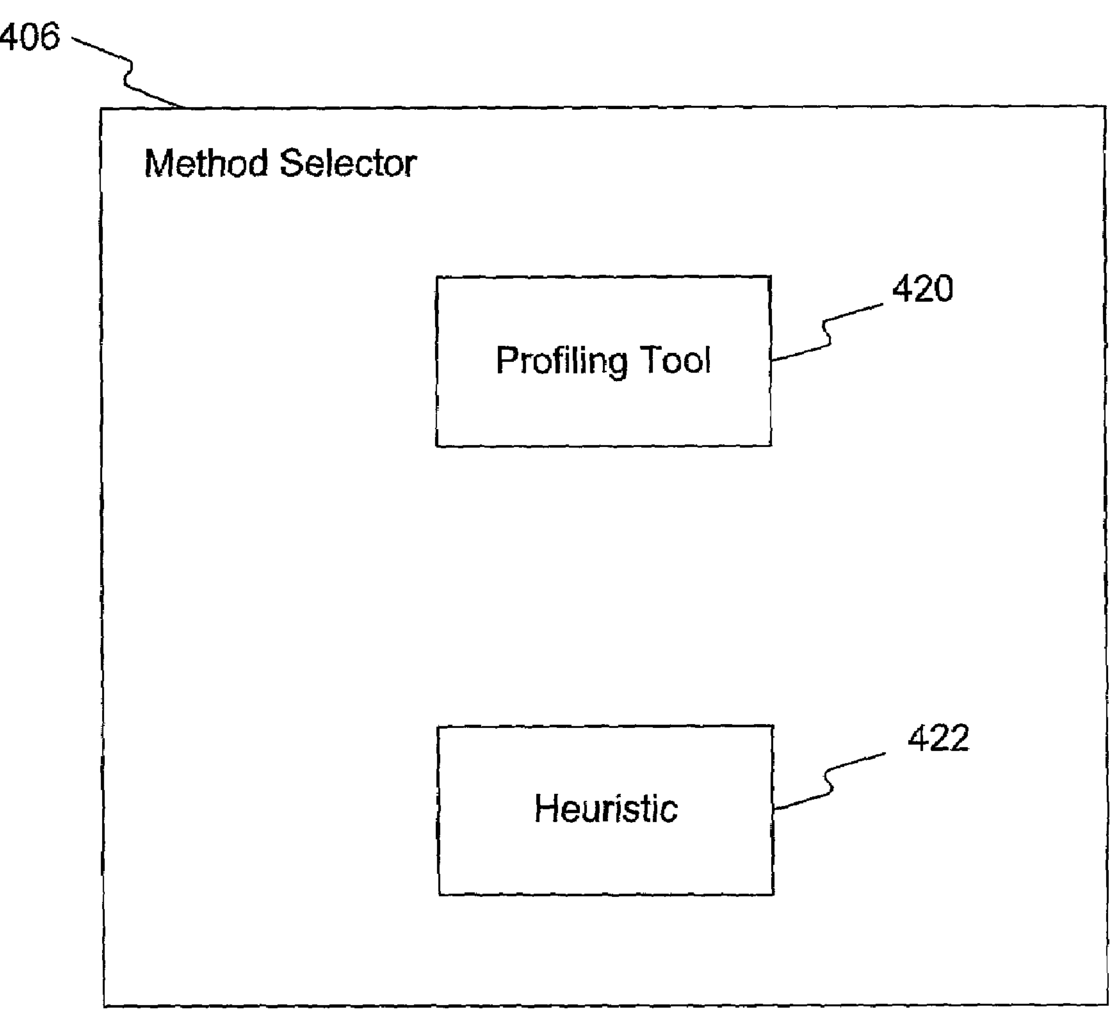


FIG. 4B

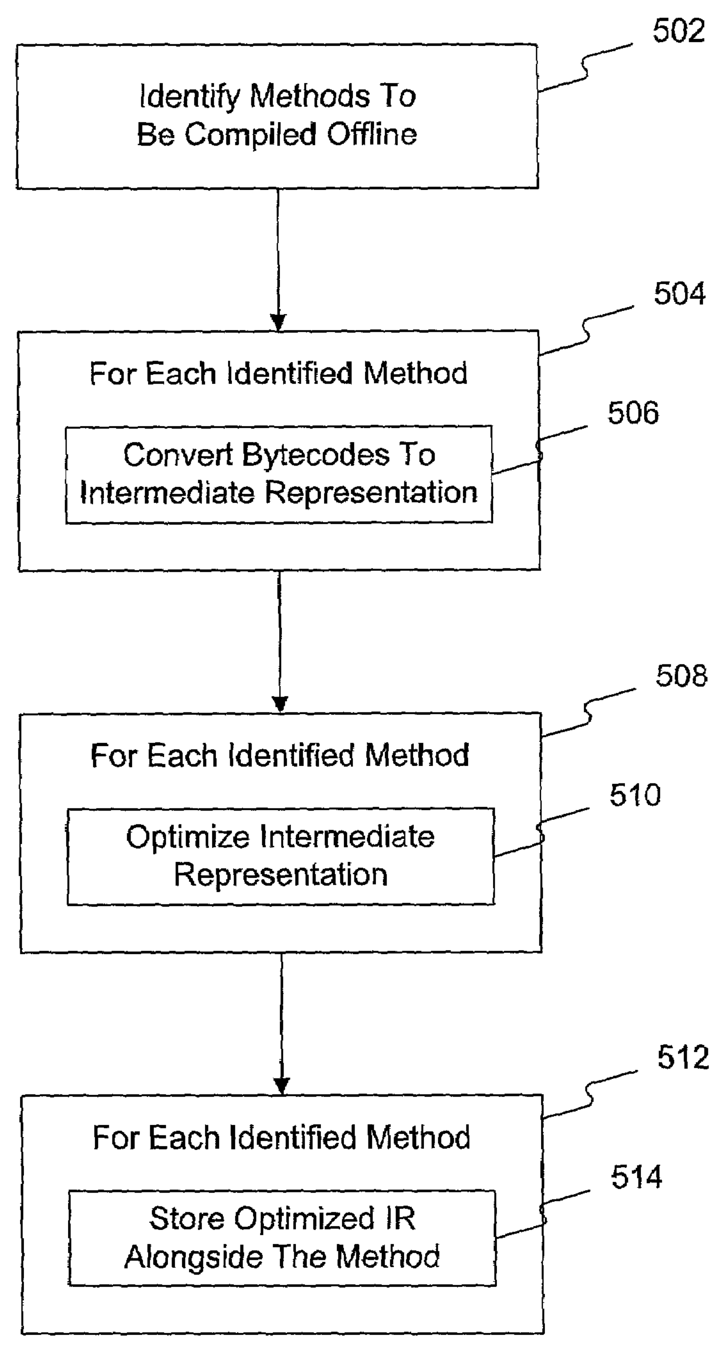


FIG. 5

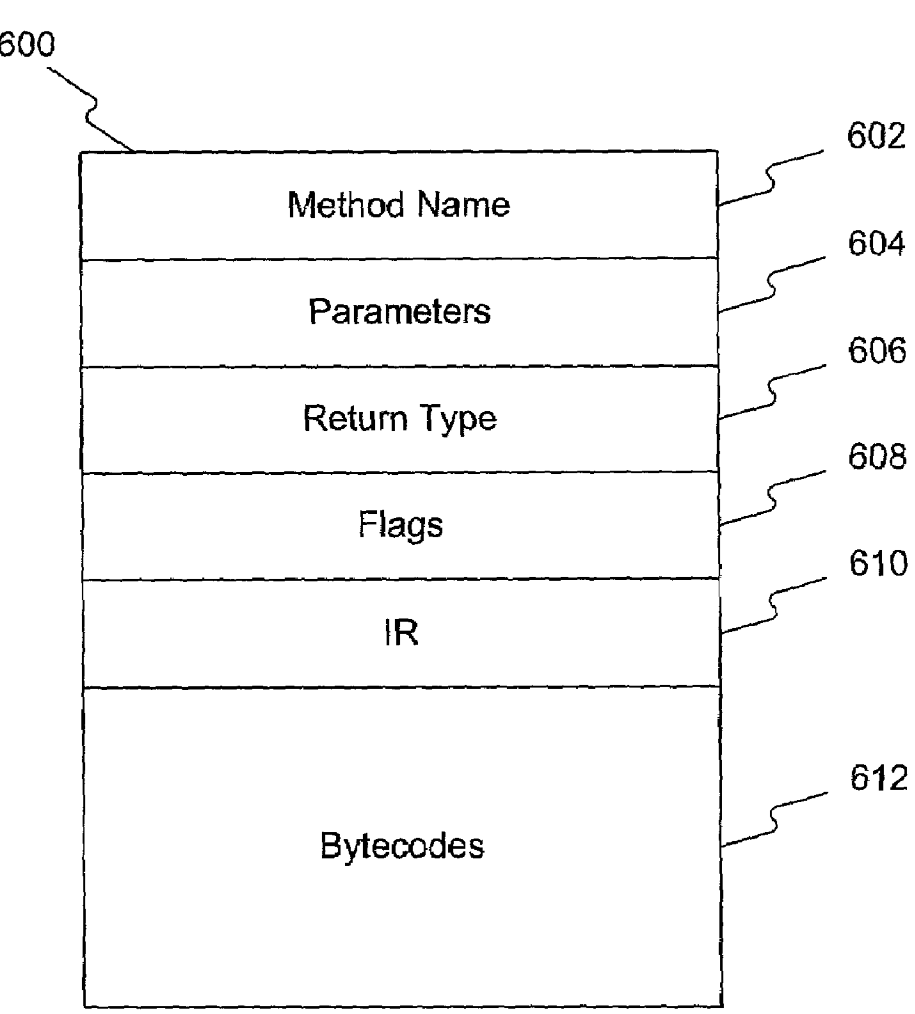


FIG. 6

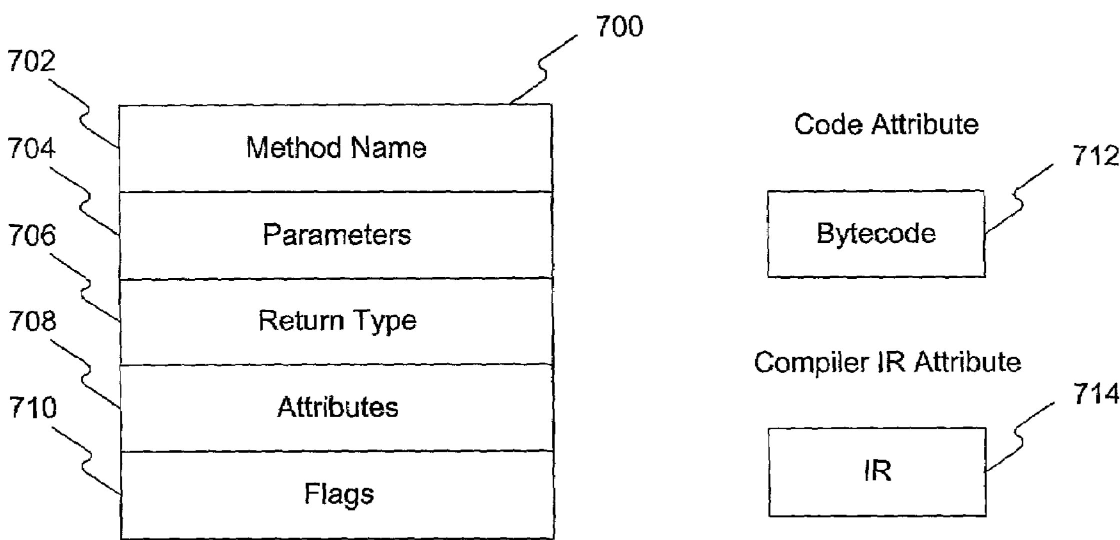


FIG. 7

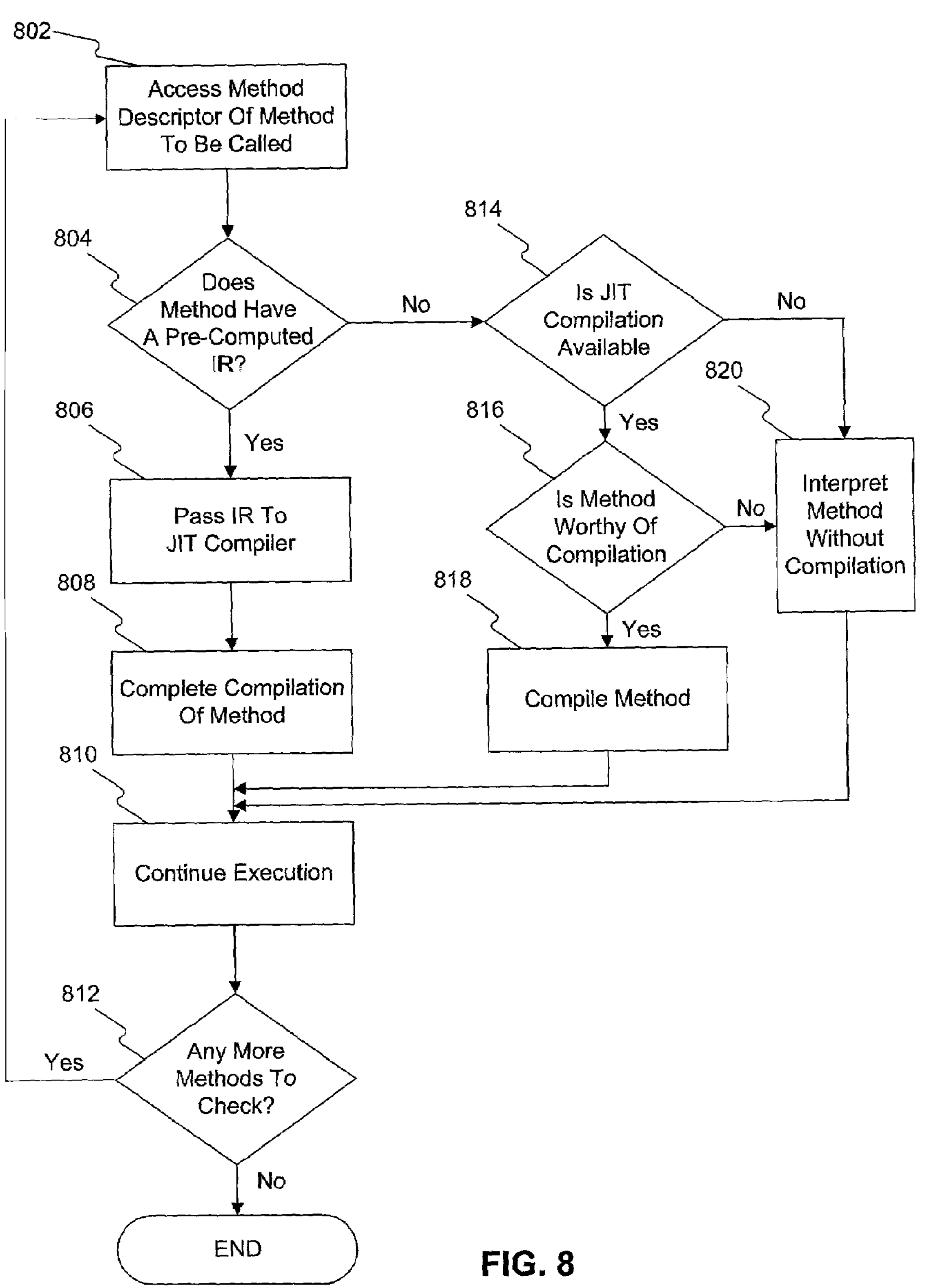


FIG. 8

PLATFORM-INDEPENDENT SELECTIVE AHEAD-OF-TIME COMPILATION

RELATED APPLICATIONS

The following identified U.S. patent applications are relied upon and are incorporated by reference in this application.

U.S. patent application Ser. No. 09/131,686, entitled "METHOD AND SYSTEM FOR LOADING CLASSES IN READ-ONLY MEMORY," filed Aug. 10, 1998, now U.S. Pat. No. 5,966,542.

FIELD OF THE INVENTION

The present invention relates generally to data processing systems and, more particularly, to platform-independent selective ahead-of-time compilation.

BACKGROUND AND MATERIAL INFORMATION

In today's society, the Internet has become an important medium for information exchange. Although the Internet is now very popular among the general public, it initially began as a system (or network) of interconnected computers used by government and academic researchers. An early problem of this network stemmed from the fact that the interconnected computers were not the same; they employed different hardware as well as different operating systems. Information exchange on such a heterogeneous network posed a communication problem. This problem was resolved through agreement on common standards, including protocols such as Transmission Control Protocol/Internet Protocol (TCP/IP) and HyperText Transfer Protocol (HTTP). These protocols enabled varied interconnected machines to share information in the form of static text or graphic documents.

These protocols, however, represented only two steps in the evolution of the Internet. Although users can exchange information documents among varied computers connected to the Internet, they cannot exchange executable application programs written in conventional languages such as C or C++, which are designed to interface with a particular processor (e.g., the Intel Pentium™ processor) and/or a particular operating system (e.g., Windows 95™ or DOS). This problem was solved with the advent of the Java™ programming language and its related runtime system.

Java is an object-oriented programming language that is described, for example, in a text entitled "The Java™ Tutorial" by Mary Campione and Kathy Walrath, Addison-Wesley, 1996. Importantly, Java is an interpreted language that is platform-independent-that is, its utility is not limited to one particular computer system. Using the Java programming language, a software developer writes programs in a form commonly called Java source code. When the developer completes authoring the program, he then compiles it with a Java compiler into an intermediate form called bytecode. Both the Java source code and the bytecode are platform-independent.

The compiled bytecode can then be executed on any computer system that employs a compatible runtime system that includes a virtual machine (VM), such as the Java virtual machine described in a text entitled "The Java Virtual Machine Specification," by Tim Lindholm and Frank Yellin, Addison Wesley, 1996. The Java VM acts as an interpreter between the bytecode and the particular computer system being used. By use of platform-independent bytecode and the Java VM, a program written in the Java language can be executed on any computer system. This is particularly useful in networks such as the Internet that interconnect heterogeneous computer systems.

Interpreting bytecodes, however, make Java programs many times slower than comparable C or C++ programs. One approach to improving this performance is just-in-time (JIT) compilers. A JIT compiler is a compiler running as part of a Java virtual machine that dynamically translates bytecode to machine code just before a method is first executed. This can provide substantial speed-up over a system that just interprets bytecodes. A JIT compilation typically consists of a few phases executed in the following order: 1) byte-codes are converted to a platform-independent intermediate representation (IR); 2) the IR is transformed to an optimized IR using compiler optimization techniques; 3) the IR is converted to platform-dependent machine code.

Java virtual machine implementations are becoming very popular on devices with limited CPU and memory resources. On such devices, the above JIT compilation process has a few drawbacks. For example, the memory requirements of the compilation process may be prohibitive, because each of the stages has runtime memory requirements which may be excessive on a limited-resource device. Also, the memory requirements of storing each method's translation may be prohibitive. Therefore, JIT's on such devices will have to make decisions on which methods are really worthy of compilation, and will have to handle only those. In addition, some translations will have to be discarded to make room for new ones. This results in slower execution because re-translating is costly.

Another drawback is that runtime handling of byte-code to IR transformation and IR optimization may result in large compiler code sizes. Dynamic method selection online is also costly in terms of compiler code size.

Yet another drawback is that due to lower processing power on a limited resource machine, the optimization phase cannot do much work without slowing down user program execution considerably.

Accordingly, there is a need for a system and method for byte code compilation that is less memory intensive, results in faster compilation and execution, and reduces re-compilation cost.

SUMMARY OF THE INVENTION

Methods and systems consistent with the principles of the invention enable platform-independent selective ahead-of-time compilation. A method selector selects a subset of methods for ahead-of-time compilation. An ahead-of-time compiler comprises a first unit and a second unit. The first unit converts, for each selected method, bytecodes corresponding to the selected method to a platform-independent intermediate representation. The second unit optimizes the platform-independent intermediate representation of each selected method, wherein each optimized intermediate representation is stored as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure.

Other methods and systems consistent with the principles of the invention enable platform-independent selective ahead-of-time compilation. A method selector comprising a profiling tool and heuristic selects a subset of methods for ahead-of-time compilation. The profiling tool ranks a set of methods according to predetermined criteria, and the heuristic identifies the subset of methods from the set of

methods. An ahead-of-time compiler comprises a first unit and a second unit. The first unit converts, for each selected method, bytecodes corresponding to the selected method to a platform-independent intermediate representation. The second unit optimizes the platform-independent intermediate representation of each selected method, wherein each optimized intermediate representation is stored with a corresponding selected method.

Other methods and systems consistent with the principles of the invention also enable platform-independent selective ahead-of-time compilation. A method selector comprising a profiling tool and heuristic selects a subset of methods for ahead-of-time compilation. The profiling tool ranks a set of methods according to predetermined criteria, and the heuristic identifies the subset of methods from the set of methods. An ahead-of-time compiler comprises a first unit and a second unit. The first unit converts, for each selected method, bytecodes corresponding to the selected method to a platform-independent intermediate representation. The second unit optimizes the platform-independent intermediate representation of each selected method, wherein each optimized intermediate representation is stored with a corresponding selected method. A class preloader may load, prior to runtime, at least one of the selected methods onto a device for execution. A dynamic class loader may load, during runtime, at least one of the selected methods onto the device for execution. A virtual machine on the device may receive at least one method from one of the class preloader and dynamic class loader. An interpreter accesses a method descriptor data structure of a method about to be called, and determines whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it. A just-in-time compiler converts the optimized intermediate representation associated with the method about to be called to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

Other methods and systems consistent with the principles of the invention also enable platform-independent selective ahead-of-time compilation. A method selector selects a subset of methods for ahead-of-time compilation. An ahead-of-time compiler comprises a first unit and a second unit. The first unit converts, for each selected method, bytecodes corresponding to the selected method to a platform-independent intermediate representation. The second unit optimizes the platform-independent intermediate representation of each selected method, wherein each optimized intermediate representation is stored as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure. A class preloader may load, prior to runtime, at least one of the selected methods onto a device for execution. A dynamic class loader may load, during runtime, at least one of the selected methods onto the device for execution. A virtual machine on the device may receive at least one method from one of the class preloader and dynamic class loader. An interpreter accesses a method descriptor data structure of a method about to be called, and determines whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it. A just-in-time compiler converts the optimized intermediate representation associated with the method about to be called to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

Other methods and systems consistent with the principles of the invention also enable platform-independent selective ahead-of-time compilation. A virtual machine on a device receives at least one method, wherein the method is from a subset of methods selected for ahead-of-time compilation, and wherein bytecodes corresponding to each selected method are converted to a platform-independent intermediate representation, the platform-independent intermediate representation of each selected method is optimized, and each optimized platform-independent intermediate representation is stored with a corresponding selected method. An interpreter accesses a method descriptor data structure of a method about to be called, and determines whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it. A just-in-time compiler converts the optimized intermediate representation associated with the method about to be called to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are incorporated in and constitute a part of this specification and, together with the description, explain the features and principles of the invention. In the drawings:

FIG. **1** is a diagram of an exemplary network environment in which features and aspects consistent with the present invention may be implemented;

FIG. **2**A is a diagram of a device consistent with the present invention;

FIG. **2**B is a diagram of a server consistent with the present invention;

FIG. **3** is a diagram showing the dataflow involved in platform-independent selective ahead-of-time compilation consistent with the present invention;

FIG. **4**A is a diagram showing the dataflow involved in the operation of an ahead-of-time compiler consistent with the present invention;

FIG. **4**B is a diagram of a method selector consistent with the present invention;

FIG. **5** is an exemplary flowchart of a method for compiling methods ahead-of-time consistent with the present invention;

FIG. **6** is a diagram of a method descriptor with an optimized IR stored as a field consistent with the present invention;

FIG. **7** is a diagram of a method descriptor and related attributes for use in dynamic class loading consistent with the present invention; and

FIG. **8** is an exemplary flowchart for executing processes consistent with the present invention.

DETAILED DESCRIPTION

The following detailed description of the invention refers to the accompanying drawings. While the description includes exemplary implementations, other implementations are possible, and changes may be made to the implementations described without departing from the spirit and scope of the invention. The following detailed description does not limit the invention. Instead, the scope of the invention is defined by the appended claims and their equivalents.

Overview

Methods and systems consistent with the principles of the invention enable platform-independent selective ahead-of-time compilation. A method selector comprising a profiling tool and heuristic select a subset of methods for ahead-of-time compilation. The profiling tool ranks a set of methods according to predetermined criteria, and the heuristic identifies the subset of methods from the set of methods. An ahead-of-time compiler comprises a first unit and a second unit. The first unit converts, for each selected method, bytecodes corresponding to the selected method to a platform-independent intermediate representation. The second unit optimizes the platform-independent intermediate representation of each selected method, wherein each optimized intermediate representation is stored with a corresponding selected method. A class preloader is operable to load, prior to runtime, at least one of the selected methods onto a device for execution. Furthermore, a dynamic class loader is operable to load, during runtime, at least one of the selected methods onto the device for execution.

A virtual machine located on a device is operable to receive at least one method from one of the class preloader and dynamic class loader. An interpreter in the virtual machine may access a method descriptor data structure of a method about to be called, and determine whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it. A just-in-time compiler in the virtual machine may convert the optimized intermediate representation associated with the method about to be called to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

Network Environment

FIG. **1** is a diagram of an exemplary network environment in which features and aspects consistent with the present invention may be implemented. Network environment **100** may include host **102**, servers **104***a*–**104***n*, devices **106***a*–**106***n*, and network **108**. The components of FIG. **1** may be implemented through hardware, software, and/or firmware. The number of components in network environment **100** is not limited to what is shown.

Host **102** and servers **104***a*–**104***n* may supply devices **106***a*–**106***n* with programs written in a platform-independent language, such as Java. For example, a software developer may create one or more Java programs and compile them into class files that contain bytecodes executable by a virtual machine, such as a Java virtual machine. When a device, such as device **106***a*, wishes to execute a Java program, it may issue a request to a server, such as server **104***a*, that contains the program. In response, server **104***a* transmits the corresponding class files to device **106***a* via an appropriate communication channel, such as network **108** (which may comprise a wired or wireless communication network, including the Internet). Device **106***a* may load the class files into a virtual machine located in device **106***a* and proceed to execute the Java program. Alternatively, a device may receive a program, such as a Java program, from host **102** via a direct connection, or from another device.

Host **102** may include CPU **110**, secondary storage **112**, input device **114**, display **116**, communications device **118**, and memory **120**. Memory **120** may include operating system **122**, compiler **124**, source code **126**, class file repository **128**, which includes class files **130** and optimized intermediate representations (IR) **132**, class preloader **134**, ahead-of-time compiler **136**, and method selector **138**. An IR may be platform independent, system architecture neutral data which is processed from source code to a format that can be quickly processed into efficient, optimized machine dependent code at some future time.

Compiler **124** translates source code into class files that contain bytecodes executable by a virtual machine. Source code **126** may be files containing code written in the Java programming language. Class file repository **128** includes class files **130** and optimized IR **132**. Class files **130** are bytecodes executable by a virtual machine and contain data representing a particular class, including data structures, method implementations, and references to other classes. Optimized IR **132** are platform-independent intermediate representations that have been optimized using common compiler techniques and associated with particular methods subjected to ahead-of-time compilation. Class files and optimized IR stored in class file repository **128** may be stored either temporarily or on a more permanent basis.

Class preloader **134** is used to preload, onto a device with a virtual machine, certain classes prior to runtime. Any Java application, or any other set of methods that are normally loaded at runtime could be preloaded using class preloader **134**. The operation of a class preloader is more particularly described in U.S. Pat. No. 5,966,542 to Tock, which has already been incorporated by reference.

Ahead-of-time compiler **136** handles the platform-independent parts of method compilation, leaving the final, platform-dependent part of method compilation to a JIT compiler running on a virtual machine. For example, ahead-of-time compiler **136** may utilize a profiling tool to identify methods that should be compiled prior to runtime. Thereafter, ahead-of-time compiler **136** performs bytecode to IR transformation and IR optimization on the identified methods. The resulting optimized IR from the compilation of each method is stored alongside the method.

Method selector **138** determines which methods should be compiled prior to runtime using ahead-of-time compiler **136**. Method selector **138** may run a profiling tool on class files to create an ordered list of methods based on predetermined criteria. This ordered list is used with a heuristic to determine which methods should be compiled prior to runtime.

FIG. **2**A is a diagram of device **106***a* in greater detail, although the other devices **106***b*–**106***n* may contain similar components. Device **106***a* may include CPU **202**, secondary storage **204**, communications device **206**, input device **208**, display **210**, and memory **212**. Memory **212** may include operating system **214**, browser **216**, class file repository **218**, virtual machine **222**, and runtime class loader **230**.

When a user of device **106***a* wishes to execute a program stored on a server, such as server **104***a*, the user may use browser **216** to issue a request to server **104***a*. In response, server **104***a* transmits the corresponding class files to device **106***a* via network **108**. For example, class files from server **104***a* may be stored either temporarily or on a more permanent basis in class file repository **218**. Device **106***a* may load the class files into a virtual machine, such as virtual machine **222**, located in device **106***a* and proceed to execute the program. Alternatively, device **106***a* may load class files from class file repository **218** that were not received from a server. For example, class file repository **218** may receive class files from host **102** for later loading into the virtual machine.

Class file repository **218** may include preloaded class files **219** and dynamically loaded class files **220**. Preloaded class

files **219** are those class files that are loaded onto device **106***a* prior to runtime using, for example, class preloader **134** on host **102**. Dynamically loaded class files **220** are those class files that are dynamically loaded at runtime using, for example, runtime class loader **230**. Preloaded class files **219** and dynamically loaded class files **220** may include optimized IR associated with particular methods that were subjected to ahead-of-time compilation.

Virtual machine **222** may include preloaded class files **224**, JIT compiler **226** and interpreter **228** and is operable to execute class files. In one implementation, virtual machine **222** is a Java virtual machine. One of ordinary skill in the art will recognize that other types of virtual machines may be used instead. Preloaded class files **224** are those class files that are loaded onto device **106***a* prior to runtime using, for example, class preloader **134** on host **102**, and may include optimized IR associated with particular methods that were subjected to ahead-of-time compilation. Virtual machine **222** may utilize both JIT compiler **226** and interpreter **228** to help execute class files.

JIT compiler **226** performs either fast compilation or JIT compilation of methods. Fast compilation may occur when JIT compiler **226** processes a method that is associated with a pre-computed optimized IR. For example, when JIT compiler **226** receives a method with an optimized IR, it converts the IR to platform-dependent machine code, which may then be executed by virtual machine **222**. JIT compilation may occur when JIT compiler **226** processes a method that is not associated with a pre-computed optimized IR. For example, when JIT compiler **226** receives a method without an optimized IR, the method may be processed during runtime using traditional JIT compilation (e.g., bytecode to IR transformation, IR optimization, and code generation).

Interpreter **228** interprets Java class files without compilation to platform-dependent code. Interpreter **228** examines methods being executed by virtual machine **224** to determine whether an optimized IR is associated with specific methods. If a method does have an optimized IR, then interpreter **228** causes JIT compiler **226** to perform a fast compilation on the method. Otherwise, interpreter **228** may perform further tests on a method to decide whether the method should be interpreted or compiled by JIT compiler **226** using JIT compilation.

Runtime class loader **230** dynamically loads classes into a user's address space at runtime. For example, runtime class loader **230** may pull class files (which may include optimized IR) during runtime from a local class file repository, such as class file repository **218**, or from a remote class file repository on a server or another device. These class files may then be appropriately processed for execution.

FIG. **2**B is a diagram of server **104***a* in greater detail, although the other devices **106***b*–**106***n* may contain similar components. Server **104***a* may include CPU **232**, secondary storage **234**, input device **236**, display **238**, communications device **240**, and memory **242**. Memory **242** may include operating system **244**, compiler **246**, source code **248**, class file repository **250**, which includes class files **252** and optimized intermediate representations (IR) **254**, and class preloader **256**. The various units of server **104***a* function in a manner similar to the similarly named units of host **102**.

Server **104***a* may receive class files **252** and optimized IR **254** from host **102**, which may perform ahead-of-time compilation. Server **104***a* may then distribute class files **252** and optimized IR **254** to a device, such as device **106***a*, as needed. For example, server **104***a* may utilize class preloader **256** to load the appropriate data onto device **106***a* prior to runtime. Alternatively, a runtime class loader on device **106***a* may pull data from class file repository **250** during runtime.

FIG. **3** is a diagram showing the dataflow involved in platform-independent selective ahead-of-time compilation consistent with the present invention. In the diagram depicted in FIG. **3**, device **304** executes a Java program that was initially located on host **302**. Java source code **306** is provided to Java compiler **308**, which translates Java source code into class files **310** that contain bytecodes executable by a virtual machine. Class files **310** may be provided to method selector **312**. Method selector **312** may select various methods to be compiled prior to runtime and pass the selected class files **314** associated with the methods to ahead-of-time compiler **316**, which may perform an ahead-of-time compilation on some of the selected class files **314**.

Ahead-of-time compiler **316** performs ahead-of-time compilation (e.g., prior to runtime) on identified methods. For example, ahead-of-time compiler **316** first converts bytecodes to an IR. Next, ahead-of-time compiler **316** optimizes the IR using common compiler techniques. Ahead-of-time compiler **316** also causes the optimized IR to be stored alongside the respective relevant methods. As a result, ahead-of-time compiler **316** may output various class files, along with associated optimized IR (class files & optimized IR **318**), if any. In one implementation, ahead-of-time compiler **316** may also output some class files that do not have any optimized IR associated with them, such that some class files from ahead-of-time compiler **316** have optimized IR and some do not. Greater detail on the operation of an ahead-of-time compiler is provided below with reference to FIGS. **4**–**5**.

Class files & optimized IR **318** may be provided to either class preloader **322** or to storage local to device **304**, such as a class file repository (where runtime class loader **320** may then access the data), without preloading. Alternatively, class files & optimized IR **318** may be provided to a server for later distribution to a device.

Runtime class loader **320** may receive class files at the same time that it receives related optimized IR. For example, runtime class loader **320** may receive class files and optimized IR where the optimized IR are stored as new attributes in method descriptors. Class preloader **322**, however, may receive class files at a different time than it receives related optimized IR. In this manner, class preloader **322** operates like an assembly line, storing optimized IR as a field of method descriptors as it receives them. Alternatively, ahead-of-time compiler **316** or a separate not shown) may store optimized IR as a field of method descriptors in a class file before being forwarded to class preloader **322**. Although runtime class loader **320** is depicted in FIG. **3** as being external to Java virtual machine **328**, some or all of runtime class loader **320** may alternatively be internal to Java virtual machine **328**.

Runtime class loader **320** and class preloader **322** load class files onto device **304** dynamically or prior to runtime, respectively. Both preloaded class files **326** and dynamically loaded class files **324** may include optimized IR stored as a field of method descriptors corresponding to methods compiled prior to runtime. Preloaded class files **326** are stored on device **304** prior to runtime. Accordingly, if a method has an optimized IR associated with it, the optimized IR needs to be stored as a field in the method descriptor either by class preloader **322** or a unit exterior to class preloader **322** before runtime commences. Although preloaded class files **326** are depicted in FIG. **3** as initially being external to Java virtual machine **328**, some or all of the preloaded class files **326**

may alternatively reside in Java virtual machine **328**. Runtime class loader **320** receives class files & optimized IR during runtime from a server, another device, or local storage, and then produces dynamically loaded class files **324**. Accordingly, a method with an optimized IR associated with it may have the optimized IR stored in its method descriptor either at runtime or prior to runtime. Greater detail on storing optimized IR with a method is provided below with reference to FIGS. **4**–**7**.

Dynamically loaded class files **324** and/or preloaded class files **326** are provided to Java virtual machine **328**, where JIT compiler **330**, interpreter **332**, services from the underlying operating system **334**, and the computer hardware (not shown) aid in the execution of the class files. Interpreter **332** recognizes whether a particular method has an optimized IR associated with it and may cause JIT compiler **330** to compile the method using fast compilation (e.g., skip bytecode to IR transformation and IR optimization), if there is such an optimized IR. Greater detail on the operation of a JIT compiler and interpreter consistent with the present invention is provided below with reference to FIG. **8**.

FIG. **4**A is a diagram showing the dataflow involved in the operation of an ahead-of-time compiler consistent with the present invention. Bytecodes **404** from a class file are provided to method selector **406**, which may be outside ahead-of-time compiler **402**, where methods that are to be compiled prior to runtime are selected. Alternatively, method selector **406** may be internal to ahead-of-time compiler **402**. Method selector **406** may subsequently send selected bytecodes **408** associated with the selected methods to ahead-of-time compiler **402**. Specifically, selected bytecodes **408** are sent to bytecode to IR transformation unit **408**. The selected bytecodes **408** are provided to bytecode to IR transformation unit **410** for conversion to platform-independent intermediate representations (IR). The IR **412** are provided to IR optimization unit **414**, where they are changed into optimized IR **416** using common compiler techniques. Subsequently, optimized IR **416** are stored alongside the relevant methods (storage **418**). The optimized IR and corresponding methods are made available for use by a class preloader or runtime class loader.

FIG. **4**B is a diagram of a method selector **406** in greater detail. Profiling tool **420** runs on class files or Java source code to create an ordered list of methods based on predetermined criteria. For example, profiling tool **420** may rank methods according to number of times called, execution time, memory size, predetermined list, randomly, and various other factors. Heuristic **422** may examine the list created by profiling tool **420** and, using developer-chosen criteria, determine which specific methods from the list should be compiled prior to runtime. Profiling tool **420** and heuristic **422** need not be part of the same unit (e.g., method selector **406**). Additionally, profiling tool **420** may be located on a device with a virtual machine. In such a configuration, profiling tool **420** may collect statistics on programs as they are running on the virtual machine, and subsequently send the heuristic (which may be on a host, server, another device, or the same device as the profiling tool) an ordered list of methods for further processing.

FIG. **5** is an exemplary flowchart of a method for compiling methods ahead-of-time consistent with the present invention. The flowchart of FIG. **5** corresponds to the dataflow of FIG. **4**. Although the steps of the flow chart are described in a particular order, one skilled in the art will appreciate that these steps may be performed in a different order, or that some of these steps may be concurrent.

First, a method selector identifies the methods that should be compiled prior to runtime (step **502**). For example, a profiling tool of the method selector may run an application or a set of applications (e.g., Java source code or bytecodes). As the profiling tool runs the applications, it may collect statistics on the various methods in the applications. The profiling tool may then create an ordered list of methods, ranked according to predetermined criteria. For example, the profiling tool may determine which methods are called the most often and rank the methods accordingly, with the most-called method ranked first. Another factor which the profiling tool may use is memory size. For example, the profiling tool may rank methods according to memory size, because space available for storing an IR may be limited. Other factors that may be used to help determine how methods are initially ranked include execution time, a list predetermined by a developer, or random ranking. One skilled in the art will recognize that the aforementioned factors may each be used as a sole basis for ranking or in some combination with each other, and that additional factors not specifically listed here may be used.

Once the profiling tool creates an ordered list of methods, it uses the ordered list with a heuristic to determine which of the most used methods should be compiled prior to runtime. The heuristic essentially shortens the ordered list created by the profiling tool. The shortened list comprises the methods that should be compiled prior to runtime. A developer may choose the criteria that the heuristic uses to determine exactly which methods should selected. For example, a developer may decide that only the first ten methods on the ordered list should be compiled prior to runtime, or that only methods that were called more than a certain number of times should be selected. Alternatively, the developer may specify that a certain number of random methods from the ordered list should be selected, or that only those methods from a predetermined list that are in the top 40 methods of the ordered list should be selected. One skilled in the art will recognize that the developer may choose criteria not specifically mentioned here to determine which methods from an ordered list should be compiled prior to runtime. The method selector (e.g., profiling tool and heuristic) described above may be part of an ahead-of-time compiler, or it may be a separate unit.

For each method identified as a method that should be compiled prior to runtime (step **504**), the ahead-of-time compiler converts the bytecodes of the method to an intermediate representation (IR) (step **506**). The ahead-of-time compiler also optimizes the IR of each identified method (steps **508**, **510**). An IR may be optimized using common compiler techniques. Because the techniques are utilized prior to runtime, compiler techniques that may be too expensive for runtime computation may be utilized. Examples of such techniques include global common sub-expression, loop invariant hoisting, common sub-expression elimination, and liveness analysis. One skilled in the art will recognize that other compiler techniques may be used.

After the intermediate representations (IR) have been optimized, the ahead-of-time compiler may cause the optimized IR of each identified method to be stored alongside its corresponding method (steps **512**, **514**). Optimized IR may be stored in two different ways. The type of storage is dependent on whether classes associated with the optimized IR are preloaded or dynamically loaded. When a class associated with an optimized IR is initially designated to be preloaded, the optimized IR is stored as a field of the method descriptor data structure of the method that was compiled to create the optimized IR. Alternatively, the method descriptor

may contain a pointer to the optimized IR instead of containing the optimized IR itself. The method descriptor may also contain a flag indicating that the method has an optimized IR associated with it. Once the optimized IR has been stored with the method descriptor, the class preloader may proceed to preload the method descriptor or store the method descriptor for later dynamic loading by the runtime class loader.

FIG. **6** is an exemplary diagram of a method descriptor that has an optimized IR stored as a field. One skilled in the art will recognize that method descriptor **600** is not limited to the specific fields depicted in FIG. **6**. Method descriptor **600** includes method name **602**, parameters **604**, return type **606**, flags **608**, IR **610**, and bytecode **612**. Method name **602** represents the name to be used when referencing the method. Parameters **604** is a list of arguments that the method uses. Each parameter is a Java class type. Return type **606** is a Java class type that is returned by the method upon execution. Flags **608** are a number of indicators used to denote various properties of the method. Flags **608** may include a flag indicating that there is an IR associated with the method. IR **610** is a platform-independent intermediate representation (IR) resulting from the ahead-of-time compilation of the method. IR **610** may be the IR itself or a pointer to the IR. Bytecodes **612** are the bytecodes and auxiliary information needed to implement the method in cases where the IR does not end up being compiled.

When a class associated with an optimized IR is initially designated to be dynamically loaded, the IR is stored as a new attribute of the method descriptor for the method. Accordingly, when a program being executed by a virtual machine on a device needs a method with an optimized IR from a server (or from another device or local storage area on the same device), a runtime class loader may load the appropriate class file onto the device. Prior to loading, if the runtime class loader is programmed to recognize the new attribute that corresponds to the IR, then it accesses the IR attribute and stores the IR as a field in the method descriptor (the IR itself or a pointer to the IR may be stored as a field). Also, the method descriptor is flagged as containing an IR. In this manner, the runtime class loader may transform the method descriptor into a method descriptor that is similar to that normally used for preloading. Alternatively, an ahead-of-time compiler may recognize the IR attribute, store the IR as a field in the method descriptor, and flag the method descriptor as containing an IR. Moreover, a development tool may perform these steps during development time (e.g., outside of runtime).

FIG. **7** is an exemplary diagram of a method descriptor and related attributes for use in dynamic class loading. One skilled in the art will recognize that method descriptor **700** is not limited to the specific fields depicted in FIG. **7**, and that additional attributes may be associated with the descriptor. Method descriptor **700** includes method name **702**, parameters **704**, return type **706**, attributes **708**, and flags **710**. Attributes **708** includes a list of attribute structures for use in conjunction with the method. In order to properly process attributes, a virtual machine, runtime class loader, or ahead-of-time compiler must be able to recognize and correctly read the attribute structures. Code attribute **712** includes the bytecodes and auxiliary information needed to implement the method. Compiler IR attribute **714** includes a platform-independent intermediate representation resulting from the ahead-of-time compilation of the method. If the runtime class loader or ahead-of-time compiler recognizes Compiler IR attribute **714** and Code attribute **712**, it includes the bytecode and IR of these attributes as fields of method descriptor **700** (e.g., it stores the bytecode and IR in the method descriptor).

FIG. **8** is an exemplary flowchart for executing methods consistent with the present invention. Although the steps of the flow chart are described in a particular order, one skilled in the art will appreciate that these steps may be performed in a different order, or that some of these steps may be concurrent.

When a virtual machine, such as virtual machine **222**, runs a program, various methods are loaded into the virtual machine either as part of preloading or dynamic class loading. If the runtime class loader recognizes that methods containing the Compiler IR attribute are loaded as part of dynamic class loading, the IR becomes a field of the method descriptor data structure prior to being loaded into the virtual machine. Also, the method descriptor is flagged as containing an IR. Alternatively, the IR of the Compiler IR attribute may be stored in the method descriptor data structure prior to runtime. Pre-loaded methods with an IR are already flagged and, have the IR as a field. As the virtual machine proceeds with executing a program, each time a method is about to be called, the interpreter of the virtual machine accesses the method descriptor of the method to be called (step **802**).

Next, the interpreter makes a determination as to whether the method to be called has a pre-computed IR (step **804**). Specifically, the interpreter checks the flags of the method descriptor to see if there is a flag indicating that there is an IR associated with the method. If there is an IR associated with the method, then the interpreter passes the IR to a JIT compiler (step **806**). Alternatively, instead of automatically sending the IR to the JIT compiler, the virtual machine may subject the method to further tests to determine whether the IR should be compiled. For example, the virtual machine may use factors such as memory usage during runtime, processor usage, user decision (e.g., user decides that he does not want IR compiled), execution time, and/or fuzzy logic, to decide whether an IR should be compiled. If the virtual machine determines that the method should still be compiled, the interpreter may pass the IR to the JIT compiler. If the virtual machine decides that the IR should not be compiled, then the interpreter proceeds to interpret the method without compilation.

Upon receiving the IR, the JIT compiler completes the compilation of the method by performing a fast compilation on it (step **808**). For example, the JIT compiler may translate the optimized IR to machine dependent code (e.g., code generation). The bytecode to IR transformation and IR optimization steps that are normally part of a JIT compilation are not performed. After fast compilation has been performed, the virtual machine continues execution (step **810**). Specifically, the virtual machine jumps to the now compiled code of the method. After the virtual machine executes the method, execution of the rest of the Java program may continue. If execution does not lead to the calling of any more methods, then execution continues until it is complete (step **812**—No). If the virtual machine determines that another method is about to be called, then the appropriate method descriptor may be accessed and processed as described above (step **812**—Yes).

If the interpreter determines that a method to be called does not have a pre-computed IR (or if the interpreter is not programmed to recognize whether a method has a pre-computed IR), then the interpreter makes a determination as to whether JIT compilation is available (step **814**). For example, the interpreter may check a flag or other indicator

associated with the JIT compiler to determine whether the JIT compiler is configured to perform JIT compilation. JIT compilation refers to compilation that at least includes bytecode to IR transformation, IR optimization, and code generation. If the interpreter determines that JIT compilation is available, the JIT compiler makes a determination as to whether the method to be called is worthy of compilation (step **816**). For example, a method may not be worthy of compilation if the bytecode is so short, that it is not worth the time it would take to compile the bytecode. If the JIT compiler determines that the method is worthy of compilation, then it proceeds to compile the method using JIT compilation (step **818**). Thereafter, the virtual machine continues execution with a jump to the now compiled code of the method.

If the interpreter determines that JIT compilation is not available, or if the JIT compiler determines that a method is not worthy of compilation, then the interpreter proceeds to interpret the method without compilation (step **820**). The virtual machine may then continue execution of the rest of the Java program.

While the present invention has been described in connection with various embodiments, many modifications will be readily apparent to those skilled in the art. Although aspects of the present invention are described as being stored in memory, one skilled in the art will appreciate that these aspects can also be stored on or read from other types of computer-readable media, such as secondary storage devices, like hard disks, floppy disks, or CD-ROM; a carrier wave, optical signal or digital signal from a network, such as the Internet; or other forms of RAM or ROM either currently known or later developed. Additionally, although a number of the software components are described as being located on the same machine, one skilled in the art will appreciate that these components may be distributed over a number of machines. The invention, therefore, is not limited to the disclosure herein, but is intended to cover any adaptations or variations thereof.

What is claimed is:

1. A process for platform-independent selective ahead-of-time compilation in a system including a host and a device, comprising:

selecting, by the host, a subset of methods from bytecodes for ahead-of-time compilation, the selecting including ranking a set of methods according to predetermined criteria and identifying the subset of methods from the set of methods using a heuristic, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation;

converting, by the host, for each selected method, the bytecodes corresponding to the selected method to a platform-independent intermediate representation;

storing each of the intermediate representations with a corresponding selected method; and

loading at least one of the selected methods onto the device for execution by a virtual machine on the device, wherein the virtual machine accesses a method descriptor data structure associated with the at least one selected method, determines whether the at least one selected method has an intermediate representation associated with it, and, based on a determination that the at least one selected method is associated with an intermediate representation, converts the intermediate representation associated with the at least one selected method to platform-dependent machine code.

2. The process of claim **1**, further comprising:

optimizing, by the host, the platform-independent intermediate representation of each selected method before the storing.

3. The process of claim **1**, wherein the ranking includes creating an ordered list of methods in the set of methods.

4. The process of claim **1**, wherein the predetermined criteria is number of times called.

5. The process of claim **1**, wherein the predetermined criteria is memory size.

6. The process of claim **1**, wherein the predetermined criteria is execution time.

7. The process of claim **1**, wherein the predetermined criteria is a list determined by a developer.

8. The process of claim **1**, wherein the heuristic is based on developer-chosen criteria.

9. The process of claim **2**, said storing comprising storing each optimized intermediate representation as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure.

10. The process of claim **9**, wherein an optimized intermediate representation is stored as a field of a corresponding method descriptor data structure when the corresponding selected method is preloaded.

11. The process of claim **9**, wherein an optimized intermediate representation is stored as an attribute of a corresponding method descriptor data structure when the corresponding selected method is dynamically loaded.

12. The process of claim **1**, wherein the virtual machine determines whether the at least one selected method has a platform-independent intermediate representation associated with it by checking a flag in the associated method descriptor data structure.

13. The process of claim **1**, wherein the virtual machine selectively converts the platform-independent intermediate representation associated with the at least one selected method to platform-dependent code.

14. The process of claim **13**, wherein the selective conversion is based on at least one of memory usage during runtime, processor usage, user decision, or fuzzy logic.

15. The process of claim **2**, wherein the optimizing is performed according to at least one of global common subexpression, loop invariant hoisting, common sub-expression elimination, and liveness analysis.

16. A process for platform-independent selective ahead-of-time compilation in a system including a host and a device, comprising:

selecting, by the host, a subset of methods from bytecodes for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation;

converting, by the host, for each selected method, the bytecodes corresponding to the selected method to a platform-independent intermediate representation;

optimizing, by the host, the platform-independent intermediate representation of each selected method; and

storing each of the optimized intermediate representations as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure;

loading at least one of the selected methods onto the device for execution by a virtual machine on the device, wherein the virtual machine accesses a method descriptor data structure associated with the at least one selected method, determines whether the at least one selected method has an optimized intermediate repre-

sentation associated with it, and, based on a determination that the at least one selected method is associated with an optimized intermediate representation, converts the optimized intermediate representation associated with the at least one selected method to platform-dependent machine code.

17. The process of claim **16**, said selecting comprising:

ranking a set of methods according to predetermined criteria; and

identifying the subset of methods from the set of methods using a heuristic.

18. The process of claim **17**, wherein the ranking includes creating an ordered list of methods in the set of methods.

19. The process of claim **17**, wherein the predetermined criteria is number of times called.

20. The process of claim **17**, wherein the predetermined criteria is memory size.

21. The process of claim **17**, wherein the predetermined criteria is execution time.

22. The process of claim **17**, wherein the predetermined criteria is a list determined by a developer.

23. The process of claim **17**, wherein the heuristic is based on developer-chosen criteria.

24. The process of claim **16**, wherein an optimized intermediate representation is stored as a field of a corresponding method descriptor data structure when the corresponding selected method is preloaded.

25. The process of claim **16**, wherein an optimized intermediate representation is stored as an attribute of a corresponding method descriptor data structure when the corresponding selected method is dynamically loaded.

26. The process of claim **16**, wherein the virtual machine determines whether the at least one selected method has a platform-independent intermediate representation associated with it by checking a flag in the associated method descriptor data structure.

27. The process of claim **16**, wherein the virtual machine selectively converts the platform-independent intermediate representation associated with the at least one selected method to platform-dependent code.

28. The process of claim **27**, wherein the selective conversion is based on at least one of memory usage during runtime, processor usage, user decision, or fuzzy logic.

29. A process, performed by a device, for platform-independent selective ahead-of-time compilation in a system including a host and the device, comprising:

receiving at least one method from the host, wherein the method is from a subset of methods from bytecodes selected by the host for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation, and wherein the bytecodes corresponding to each selected method are converted by the host to a platform-independent intermediate representation, and each of the platform-independent intermediate representations is stored with a corresponding selected method;

accessing a method descriptor data structure of a method about to be called;

determining whether the method descriptor data structure has a platform-independent intermediate representation associated with it; and

converting the intermediate representation associated with the at least one selected method to platform-dependent machine code based on a determination that the method descriptor data structure has a platform-independent intermediate representation associated with it.

30. The process of claim **29**, wherein the platform-independent intermediate representation of each selected method is optimized by the host before storing with a corresponding selected method.

31. An apparatus for platform-independent selective ahead-of-time compilation in a system including a host and a device, comprising:

a method selector on the host and operable to select a subset of methods from bytecodes for ahead-of-time compilation, the method selector including a profiling tool operable to rank a set of methods according to predetermined criteria and a heuristic operable to identify the subset of methods from the set of methods, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation; and

an ahead-of-time compiler on the host, the ahead-of-time compiler including a first unit and a second unit, the first unit operable to convert, for each selected method, the bytecodes corresponding to the selected method to a platform-independent intermediate representation, and the second unit operable to optimize the platform-independent intermediate representation of each selected method, wherein each of the optimized intermediate representations is stored with a corresponding selected method;

a class preloader operable to load, prior to runtime, at least one of the selected methods onto the device for execution by a virtual machine, wherein

the virtual machine accesses a method descriptor data structure associated with the at least one selected method, determines whether the at least one selected method has an optimized intermediate representation associated with it, and, based on a determination that the at least one selected method is associated with an optimized intermediate representation, converts the optimized intermediate representation associated with the at least one selected method to platform-dependent machine code.

32. The apparatus of claim **31**, wherein the profiling tool creates an ordered list of methods in the set of methods.

33. The apparatus of claim **31**, wherein the predetermined criteria is number of times called.

34. The apparatus of claim **31**, wherein the predetermined criteria is memory size.

35. The apparatus of claim **31**, wherein the predetermined criteria is execution time.

36. The apparatus of claim **31**, wherein the predetermined criteria is a list determined by a developer.

37. The apparatus of claim **31**, wherein the heuristic identifies the subset of methods based on developer-chosen criteria.

38. The apparatus of claim **31**, each optimized intermediate representation is stored as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure.

39. The apparatus of claim **38**, wherein an optimized intermediate representation is stored as a field of a corresponding method descriptor data structure when the corresponding selected method is loaded by the class preloader prior to runtime.

40. The apparatus of claim **38**, wherein an optimized intermediate representation is stored as an attribute of a corresponding method descriptor data structure when the corresponding selected method is loaded by a dynamic class loader during runtime.

41. An apparatus for platform-independent selective ahead-of-time compilation in a system including a host, comprising:

a virtual machine operable to receive at least one method from the host, wherein the method is from a subset of methods from bytecodes selected by the host for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation, and wherein the bytecodes corresponding to each selected method are converted by the host to a platform-independent intermediate representation, the platform-independent intermediate representation of each selected method is optimized by the host, and each of the optimized platform-independent intermediate representations is stored with a corresponding selected method;

an interpreter operable to access a method descriptor data structure of a method about to be called, and determine whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it; and

a just-in-time compiler operable to convert the optimized intermediate representation associated with the method about to be called to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

42. The apparatus of claim 41, wherein the interpreter determines whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it by checking a flag in the method descriptor data structure.

43. The apparatus of claim 41, wherein the just-in-time compiler selectively converts the optimized platform-independent intermediate representation associated with the method about to be called to platform-dependent code.

44. The apparatus of claim 43, wherein the selective conversion is based on at least one of memory usage during runtime, processor usage, user decision, or fuzzy logic.

45. A system for platform-independent selective ahead-of-time compilation, comprising:

a method selector on a host and operable to select a subset of methods from bytecodes for ahead-of-time compilation, the method selector including a profiling tool operable to rank a set of methods according to predetermined criteria and a heuristic operable to identify the subset of methods from the set of methods, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation;

an ahead-of-time compiler on the host, the ahead-of-time compiler including a first unit and a second unit, the first unit operable to convert, for each selected method, the bytecodes corresponding to the selected method to a platform-independent intermediate representation, and the second unit operable to optimize the platform-independent intermediate representation of each selected method, wherein each of the optimized intermediate representations is stored with a corresponding selected method;

a class preloader operable to load, prior to runtime, at least one of the selected methods onto a device for execution;

a dynamic class loader operable to load, during runtime, at least one of the selected methods onto the device for execution;

a virtual machine on a device, the virtual machine operable to receive at least one method from one of the class preloader and dynamic class loader;

an interpreter on the device, the interpreter operable to access a method descriptor data structure of a method about to be called, and determine whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it; and

a just-in-time compiler on the device, the just-in-time compiler operable to convert the optimized intermediate representation associated with the method about to be called to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

46. An apparatus for platform-independent selective ahead-of-time compilation, comprising:

means for selecting a subset of methods from bytecodes for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation;

means for converting, for each selected method, the bytecodes corresponding to the selected method to a platform-independent intermediate representation;

means for optimizing the platform-independent intermediate representation of each selected method; and

means for storing each of the optimized intermediate representations as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure;

means for loading at least one of the selected methods onto a device for execution by a virtual machine on the device, wherein the virtual machine accesses a method descriptor data structure associated with the at least one selected method, determines whether the at least one selected method has an optimized intermediate representation associated with it, and, based on a determination that the at least one selected method is associated with an optimized intermediate representation, converts the optimized intermediate representation associated with the at least one selected method to platform-dependent machine code.

47. The apparatus of claim 46, said means for selecting comprising:

means for ranking a set of methods according to predetermined criteria; and

means for identifying the subset of methods from the set of methods using a heuristic.

48. The apparatus of claim 47, wherein the means for ranking creates an ordered list of methods in the set of methods.

49. The apparatus of claim 46, wherein an optimized intermediate representation is stored as a field of a corresponding method descriptor data structure when the corresponding selected method is preloaded.

50. The apparatus of claim 46, wherein an optimized intermediate representation is stored as an attribute of a corresponding method descriptor data structure when the corresponding selected method is dynamically loaded.

51. The apparatus of claim 46, wherein the virtual machine determines whether the at least one selected method has a platform-independent intermediate representation associated with it by checking a flag in the associated method descriptor data structure.

52. The process of claim 46, wherein the virtual machine selectively converts the platform-independent intermediate

representation associated with the at least one selected method to platform-dependent code.

53. The process of claim **52**, wherein the selective conversion is based on at least one of memory usage during runtime, processor usage, user decision, or fuzzy logic.

54. A computer-readable storage medium containing instructions for performing a process for platform-independent selective ahead-of-time compilation in a system including a host and a device, the process comprising: selecting, by the host, a subset of methods from bytecodes for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation; converting, by the host, for each selected method, the bytecodes corresponding to the selected method to a platform-independent intermediate representation; optimizing, by the host, the platform-independent intermediate representation of each selected method; and storing each of the optimized intermediate representations as one of a field of a corresponding method descriptor data structure and an attribute of the corresponding method descriptor data structure; loading at least one of the selected methods onto the device for execution by a virtual machine on the device, wherein the virtual machine accesses a method descriptor data structure associated with the at least one selected method, determines whether the at least one selected method has an optimized intermediate representation associated with it, and, based on a determination that the at least one selected method is associated with an optimized intermediate representation, converts the optimized intermediate representation associated with the at least one selected method to platform-dependent machine code.

55. The computer-readable storage medium of claim **54**, said selecting comprising: ranking a set of methods according to predetermined criteria; and identifying the subset of methods from the set of methods using a heuristic.

56. The computer-readable storage medium of claim **55**, wherein the ranking includes creating an ordered list of methods in the set of methods.

57. The computer-readable storage medium of claim **54**, wherein an optimized intermediate representation is stored as a field of a corresponding method descriptor data structure when the corresponding selected method is preloaded.

58. The computer-readable storage medium of claim **54**, wherein an optimized intermediate representation is stored as an attribute of a corresponding method descriptor data structure when the corresponding selected method is dynamically loaded.

59. A computer-readable storage medium containing instructions for performing a process for platform-independent selective ahead-of-time compilation in a system including a host and a device, the process comprising: receiving at least one method from the host, wherein the method is from a subset of methods from bytecodes selected by the host for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation, and wherein the bytecodes corresponding to each selected method are converted by the host to a platform-independent intermediate representation, the platform-independent intermediate representation of each selected method is optimized by the host, and each of the optimized platform-independent intermediate representations is stored with a corresponding selected method; accessing, by a virtual machine on the device, a method descriptor data structure of a method about to be called; determining, by the virtual machine, whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it; and converting, by the virtual machine, the optimized intermediate representation associated with the at least one selected method to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

60. An apparatus for platform-independent selective ahead-of-time compilation in a system including a host and a device, comprising:

means for receiving at least one method from the host, wherein the method is from a subset of methods from bytecodes selected for ahead-of-time compilation, wherein the bytecodes are compiled from source codes prior to the ahead-of-time compilation, and wherein the bytecodes corresponding to each selected method are converted to a platform-independent intermediate representation, the platform-independent intermediate representation of each selected method is optimized, and each of the optimized platform-independent intermediate representations is stored with a corresponding selected method;

means for accessing a method descriptor data structure of a method about to be called;

means for determining whether the method descriptor data structure has an optimized platform-independent intermediate representation associated with it; and

means for converting the optimized intermediate representation associated with the at least one selected method to platform-dependent machine code based on a determination that the method descriptor data structure has an optimized platform-independent intermediate representation associated with it.

* * * * *

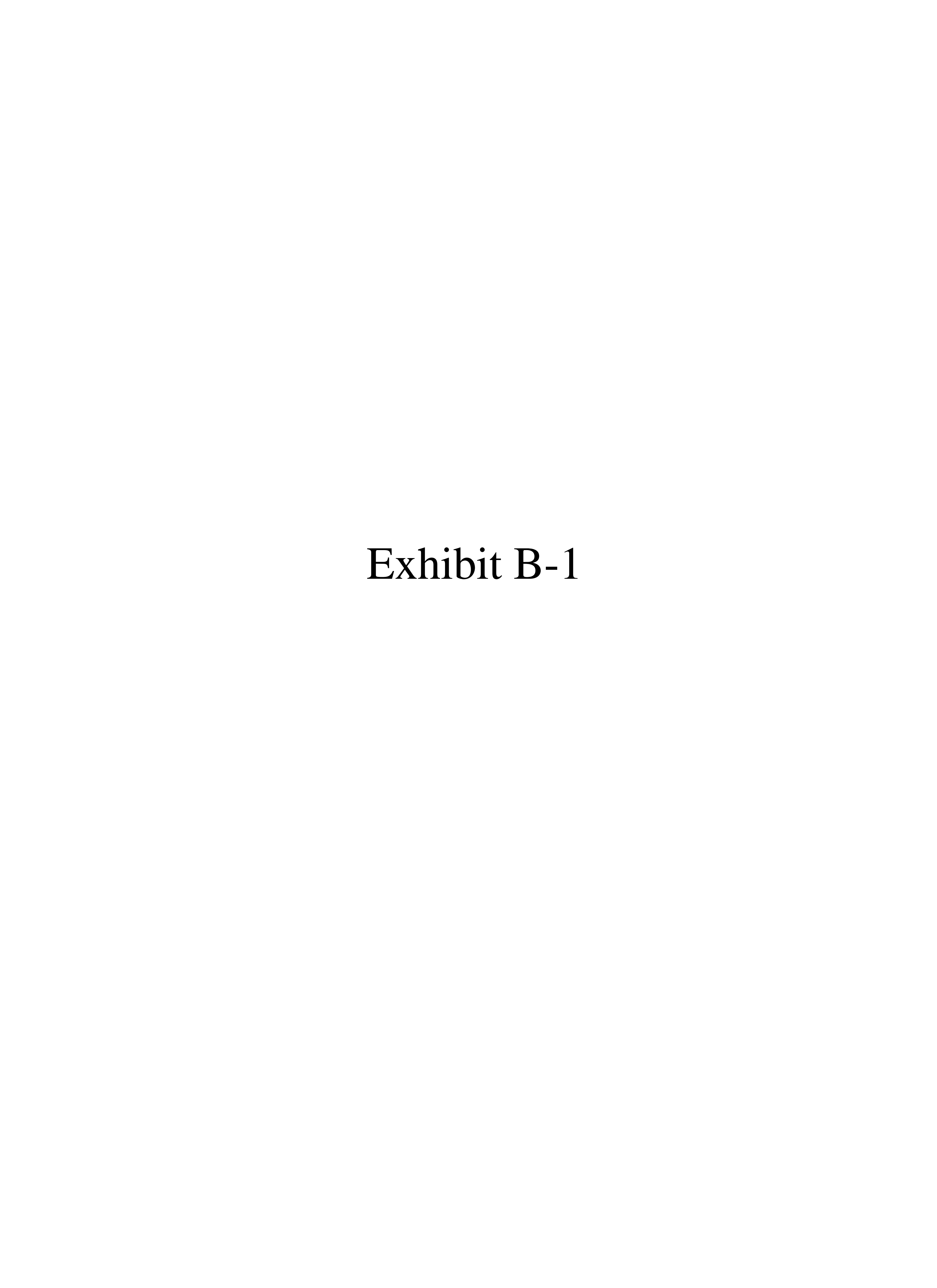


Exhibit B-1

Our Ref.: 82225.P370

APPLICATION FOR UNITED STATES LETTERS PATENT

FOR

METHOD AND APPARATUS FOR RESOLVING DATA REFERENCES IN GENERATED CODE

Inventor: JAMES GOSLING

Prepared by:

Blakely Sokoloff Taylor & Zafman
595 Market Street, Suite 1330
San Francisco CA 94105-2800
(415) 243-4354

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail (RB713125679) in an envelope addressed to : Commissioner of Patents and Trademarks, Washington, D.C. 20231 on: December 22, 1992

Judith A. Hromyko — 12/22/92
Judith A. Hromyko (Date)

OAGOOGLE0000057182

WHAT IS CLAIMED IS

1. In a computer system comprising a program in source code form, a method for generating executable code for said program and resolving data references in said generated code, said method comprising the steps of:

a) generating executable code in intermediate form for said program in source code form with data references being made in said generated code on a symbolic basis, said generated code comprising a plurality of instructions of said computer system;

b) interpreting said instructions, one at a time, in accordance to a program execution control;

c) resolving said symbolic references to corresponding numeric references, replacing said symbolic references with their corresponding numeric references, and continuing interpretation without advancing program execution, as said symbolic references are encountered while said instructions are being interpreted; and

d) obtaining data in accordance to said numeric references, and continuing interpretation after advancing program execution, as said numeric references are encountered while said instructions are being interpreted;

said steps b) through d) being performed iteratively and interleavingly.

2. The method as set forth in claim 1, wherein, said program in source code form is implemented in source code form of an object oriented programming language.

3. The method as set forth in claim 2, wherein, said programming language is C.

OAGOOGLE0000057197

4. The method as set forth in claim 2, wherein, said programming language is C++.

5. The method as set forth in claim 1, wherein,

said program execution control is a program counter;

said continuing interpretation in step c) is achieved by performing said step b) after said step c) without incrementing said program counter; and

said continuing interpretation in said step d) is achieved by performing said step b) after said d) after incrementing said program counter.

6. In a computer system comprising a program in source code form, an apparatus for generating executable code for said program and resolving data references in said generated code, said apparatus comprising:

a) compilation means for receiving said program in source code form and generating executable code in intermediate form for said program in source code form with data references being made in said generated code on a symbolic basis, said generated code comprising a plurality of instructions of said computer system;

b) interpretation means for receiving said generated code and interpreting said instructions, one at a time;

c) dynamic reference handling means coupled to said interpretation means for resolving said symbolic references to corresponding numeric references, replacing said symbolic references with their corresponding numeric references, and continuing interpretation by said interpretation means without advancing program execution, as said symbolic references are encountered while said instructions are being interpreted by said interpretation means; and

OAGOOGLE0000057198

d) static reference handling means coupled to said interpretation means for obtaining data in accordance to said numeric references, and continuing interpretation by said interpretation means after advancing program execution, as said numeric references are encountered while said instruction are being interpreted by said interpretation means;

said interpretation means, said dynamic reference handling means, and said static reference handling means performing their corresponding functions iteratively and interleavingly.

7. The apparatus as set forth in claim 6, wherein, said program in source code form is implemented in source code form of an object oriented programming language.

8. The apparatus as set forth in claim 7, wherein, said programming language is C.

9. The apparatus as set forth in claim 7, wherein, said programming language is C++.

10. The apparatus as set forth in claim 6, wherein,

said program execution control is a program counter; .

said continuing interpretation in step c) is achieved by performing said step b) after said step c) without incrementing said program counter; and

said continuing interpretation in said step d) is achieved by performing said step b) after said d) after incrementing said program counter.

17

OAGOOGLE0000057199

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

SERIAL NUMBER	FILING DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NO.
07/994,655	12/22/92	GOSLING	J 82225.P370

EXAMINER	
HECKLER,T	
ART UNIT	PAPER NUMBER
2316	4

B3M1/0527

IRELL & MANELLA
1800 AVENUE OF THE STARS
SUITE 900
LOS ANGELES, CA 90067

DATE MAILED: 05/27/94

EXAMINER INTERVIEW SUMMARY RECORD

All participants (applicant, applicant's representative, PTO personnel):

(1) Mr. Jeffrey Blatt (3) ______

(2) Ex. Heckler (4) ______

Date of interview 5-26-94

Type: ☒ Telephonic ☐ Personal (copy is given to ☐ applicant ☐ applicant's representative).

Exhibit shown or demonstration conducted: ☐ Yes ☐ No. If yes, brief description: ______

Agreement ☒ was reached with respect to some or all of the claims in question. ☐ was not reached.

Claims discussed: 1, 10

Identification of prior art discussed: ______

Description of the general nature of what was agreed to if an agreement was reached, or any other comments:

changes made as per Examiner's Amendment.

(A fuller description, if necessary, and a copy of the amendments, if available, which the examiner agreed would render the claims allowable must be attached. Also, where no copy of the amendments which would render the claims allowable is available, a summary thereof must be attached.)

Unless the paragraphs below have been checked to indicate to the contrary, A FORMAL WRITTEN RESPONSE TO THE LAST OFFICE ACTION IS NOT WAIVED AND MUST INCLUDE THE SUBSTANCE OF THE INTERVIEW (e.g., items 1–7 on the reverse side of this form). If a response to the last Office action has already been filed, then applicant is given one month from this interview date to provide a statement of the substance of the interview.

☒ It is not necessary for applicant to provide a separate record of the substance of the interview.

☐ Since the examiner's interview summary above (including any attachments) reflects a complete response to each of the objections, rejections and requirements that may be present in the last Office action, and since the claims are now allowable, this completed form is considered to fulfill the response requirements of the last Office action.

Thomas Heckler
Examiner's Signature

PTOL-413 (REV 1-84)

ORIGINAL FOR INSERTION IN RIGHT HAND FLAP OF FILE WRAPPER

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

SERIAL NUMBER	FILING DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NO
07/994,655	12/22/92	GOSLING	J 82225.P370

EXAMINER
HECKLER,T

B3M1/0527

IRELL & MANELLA
1800 AVENUE OF THE STARS
SUITE 900
LOS ANGELES, CA 90067

ART UNIT	PAPER NUMBER
2316	5

DATE MAILED: 05/27/94

NOTICE OF ALLOWABILITY

PART I.

1 ☐ This communication is responsive to ______.
2 ☒ All the claims being allowable, PROSECUTION ON THE MERITS IS (OR REMAINS) CLOSED in this application. If not included herewith (or previously mailed), a Notice Of Allowance And Issue Fee Due or other appropriate communication will be sent in due course.
3 ☒ The allowed claims are 1-10.
4 ☐ The drawings filed on ______ are acceptable.
5 ☐ Acknowledgment is made of the claim for priority under 35 U.S.C. 119. The certified copy has [_] been received. [_] not been received. [ ] been filed in parent application Serial No ______, filed on ______.
6. ☒ Note the attached Examiner's Amendment.
7 ☒ Note the attached Examiner Interview Summary Record, PTOL-413.
8 ☒ Note the attached Examiner's Statement of Reasons for Allowance.
9. ☒ Note the attached NOTICE OF REFERENCES CITED, PTO-892.
10 ☐ Note the attached INFORMATION DISCLOSURE CITATION, PTO-1449.

PART II.

A SHORTENED STATUTORY PERIOD FOR RESPONSE to comply with the requirements noted below is set to EXPIRE THREE MONTHS FROM THE "DATE MAILED" indicated on this form. Failure to timely comply will result in the ABANDONMENT of this application. Extensions of time may be obtained under the provisions of 37 CFR 1.136(a).

1. ☐ Note the attached EXAMINER'S AMENDMENT or NOTICE OF INFORMAL APPLICATION, PTO-152, which discloses that the oath or declaration is deficient. A SUBSTITUTE OATH OR DECLARATION IS REQUIRED.
2 ☒ APPLICANT MUST MAKE THE DRAWING CHANGES INDICATED BELOW IN THE MANNER SET FORTH ON THE REVERSE SIDE OF THIS PAPER.
   a ☒ Drawing informalities are indicated on the NOTICE RE PATENT DRAWINGS, PTO-948, attached hereto ~~or to Paper No.~~ ______. CORRECTION IS REQUIRED.
   b. ☐ The proposed drawing correction filed on ______ has been approved by the examiner. CORRECTION IS REQUIRED.
   c ☐ Approved drawing corrections are described by the examiner in the attached EXAMINER'S AMENDMENT. CORRECTION IS REQUIRED.
   d ☒ Formal drawings are now REQUIRED.

Any response to this letter should include in the upper right hand corner, the following information from the NOTICE OF ALLOWANCE AND ISSUE FEE DUE: ISSUE BATCH NUMBER, DATE OF THE NOTICE OF ALLOWANCE, AND SERIAL NUMBER.

Attachments.

X Examiner's Amendment
X Examiner Interview Summary Record, PTOL-413
X Reasons for Allowance
X Notice of References Cited, PTO-892
_ Information Disclosure Citation, PTO-1449
_ Notice of Informal Application, PTO-152
X Notice re Patent Drawings, PTO-948
_ Listing of Bonded Draftsmen
_ Other

Thomas Heckler

THOMAS M. HECKLER
PRIMARY EXAMINER
ART UNIT 237

1. An Examiner's Amendment to the record appears below. Should the changes and/or additions be unacceptable to applicant, an amendment may be filed as provided by 37 C.F.R. § 1.312. To ensure consideration of such an amendment, it **MUST** be submitted no later than the payment of the Issue Fee.

Authorization for this Examiner's Amendment was given in a telephone interview with Mr. Jeffrey Blatt on May 26, 1994.

2. The application has been amended as follows:

IN THE CLAIMS:

Claim 1, line 17, change "instruction" to --instructions--;

Claim 10, line 2, change the semicolon to a period;

delete lines 3-6.

3. The following is an Examiner's Statement of Reasons for Allowance: the prior art, either alone or in combination, does not teach a method or apparatus for generating executable code from source code comprising generating executable code in intermediate form with data references made on a symbolic basis, interpreting instructions, replacing symbolic references with numeric references and continuing interpretation without advancing program execution, obtaining data as numeric references are encountered and continuing interpretation after advancing program execution.

Any comments considered necessary by applicant must be submitted no later than the payment of the Issue Fee and, to avoid processing delays, should preferably **accompany** the Issue

Fee. Such submissions should be clearly labeled "Comments on Statement of Reasons for Allowance."

4. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Tom Heckler whose telephone number is (703) 305-9666.

Thomas Heckler

TH
May 26, 1994

THOMAS M. HECKLER
PRIMARY EXAMINER
ART UNIT 237

OAGOOGLE0000057223

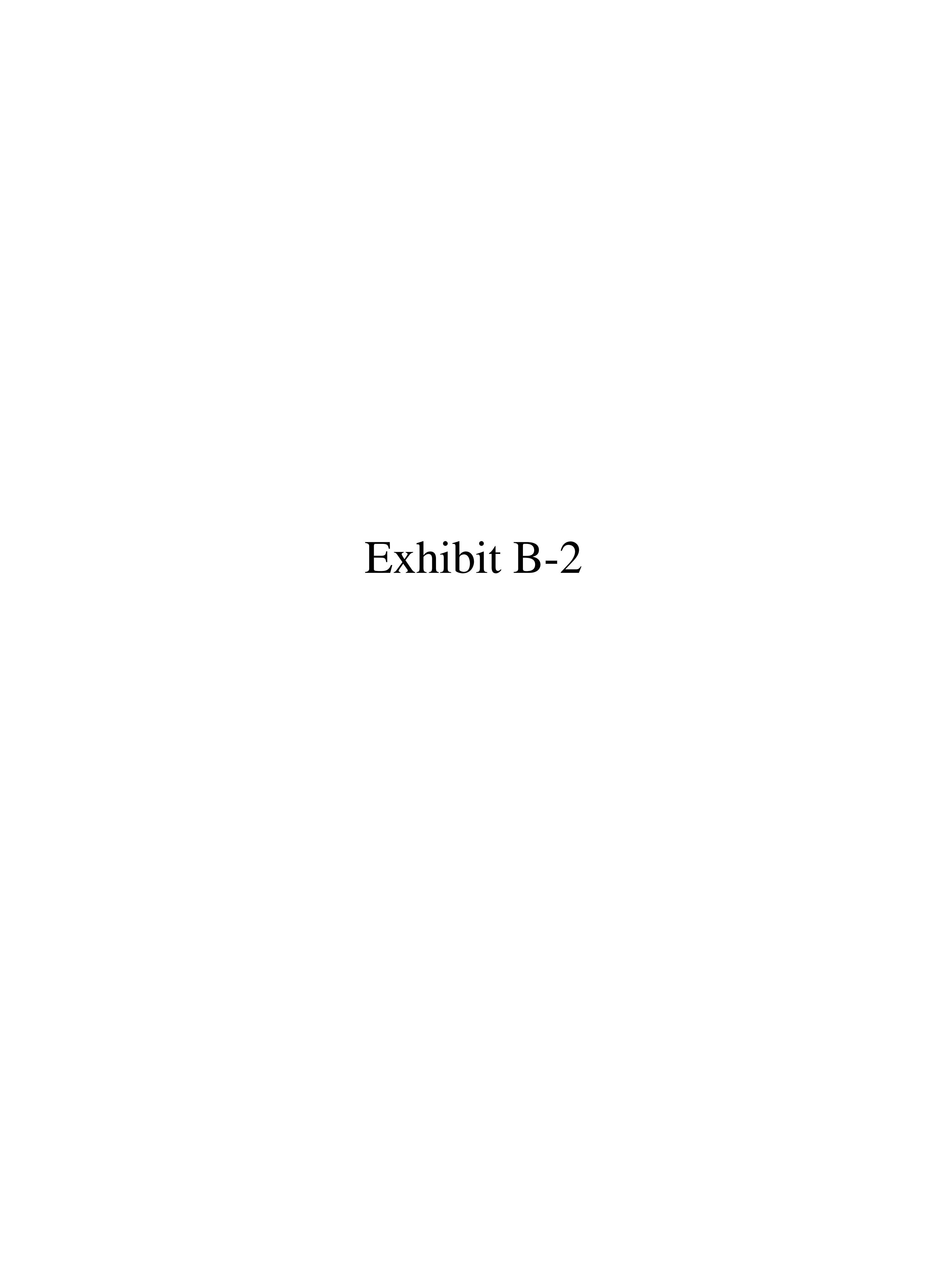


Exhibit B-2

08 755764

149907-0064

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re reissue application of U.S. Patent No. 5,367,685 Issued: November 22, 1994 Inventors: James Gosling For: METHOD AND APPARATUS FOR RESOLVING DATA REFERENCES IN GENERATED CODE	)	CERTIFICATE OF MAILING BY "EXPRESS MAIL" "EXPRESS MAIL" Mailing Label Number EM458150000US 49945 November 21, 1996 Date of Deposit I hereby certify that this paper or fee is being deposited with the United States Postal Service "EXPRESS MAIL POST OFFICE TO ADDRESSEE" service under 37 C.F.R. § 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231. Dee Henderson Typed or Printed Name of Person Mailing Paper or Fee Signature of Person Mailing Paper or Fee November 21, 1996 Date

REISSUE APPLICATION TRANSMITTAL

Hon. Commissioner of
Patents and Trademarks
Washington, D.C. 20231
Attn: Box Reissue Application

Sir:

Transmitted herewith is the application for reissue of the above-referenced U.S. patent. Enclosed are the following:

DHEN1105.WP

OAGOOGLE0000059251

a third device configured to build an intermediate representation from said parsed output, and

a fourth device configured to generate intermediate form code containing symbolic field references from said intermediate representation.

10

29. A computer program product comprising:

a computer readable medium having computer readable code embodied therein for interpreting software in an intermediate form code, said intermediate form code comprising instructions, certain of said instructions containing one or more symbolic references, said computer readable medium comprising:

5

a computer readable program code device configured to interpret said instructions in accordance with a program execution control;

a second computer readable program code device configured, operative when an instruction being interpreted contains an unresolved symbolic reference, to resolve said unresolved symbolic reference, said second computer readable program code device comprising:

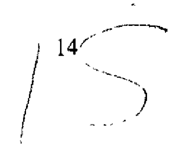
10

a third computer readable program code device configured to determine a numerical value corresponding to said unresolved symbolic reference, and

a fourth computer readable program code device configured to store said numerical value in a memory of said at least one of said plurality of computers; and

15

DHEN110A.WP



OAGOOGLE0000059223

a fifth computer readable program code device configured, operative when an instruction being interpreted contains a resolved symbolic reference, to interpret said instruction by reading said stored numerical value.

20

30. A computer program product comprising:

a computer readable medium having computer readable code embodied therein for compiling software into intermediate form code, said computer readable medium comprising:

5 a first computer readable program code device configured to lexically analyze source code of said software,

a second computer readable program code device configured to parse the output of said first computer readable program code device,

a third computer readable program code device configured to build an intermediate representation of said parsed output, and

10

a fourth computer readable program code device configured to generate intermediate form code containing symbolic field references from said intermediate representation.

31. A system for distributing code stored on a computer readable medium, said code comprising computer readable code for interpreting software in an intermediate form code, said intermediate form code comprising instructions, certain of said

OAGOOGLE0000059224

instructions containing one or more symbolic references, said computer readable medium
5 comprising:

a computer readable program code device configured to interpret said instructions in accordance with a program execution control;

a second computer readable program code device configured, operative when an instruction being interpreted contains an unresolved symbolic reference, to
10 resolve said unresolved symbolic reference, said second computer readable program code device comprising:

a third computer readable program code device configured to determine a numerical value corresponding to said unresolved symbolic reference, and

15 a fourth computer readable program code device configured to store said numerical value in a memory of said at least one of said plurality of computers; and

a fifth computer readable program code device configured, operative when an instruction being interpreted contains a resolved symbolic reference, to
20 interpret said instruction by reading said stored numerical value.

32. A system for distributing code stored on a computer readable medium, said code comprising computer readable code for compiling software into intermediate form code, said computer readable medium comprising:

a first computer readable program code device configured to lexically analyze
5 source code of said software,

OAGOOGLE0000059225

a second computer readable program code device configured to parse the output of said first computer readable program code device,

a third computer readable program code device configured to build an intermediate representation of said parsed output, and

10 a fourth computer readable program code device configured to generate intermediate form code containing symbolic field references from said intermediate representation.

33. A method for distributing code stored on a computer readable medium, said code comprising computer readable code for interpreting software in an intermediate form code, said intermediate form code comprising instructions, certain of said instructions containing one or more symbolic references, said computer readable medium

5 comprising:

a computer readable program code device configured to interpret said instructions in accordance with a program execution control;

a second computer readable program code device configured, operative when an instruction being interpreted contains an unresolved symbolic reference, to

10 resolve said unresolved symbolic reference, said second computer readable program code device comprising:

a third computer readable program code device configured to determine a numerical value corresponding to said unresolved symbolic reference, and

DHEN110A.WP

15

OAGOOGLE0000059226

15 a fourth computer readable program code device configured to store said numerical value in a memory of said at least one of said plurality of computers; and

a fifth computer readable program code device configured, operative when an instruction being interpreted contains a resolved symbolic reference, to

20 interpret said instruction by reading said stored numerical value,

said method comprising the steps of reading a portion said computer readable code from said computer readable medium into a memory and writing said portion of said code to a second computer readable medium.

34. A method for distributing code stored on a computer readable medium, said code comprising computer readable code for compiling software into intermediate form code, said computer readable medium comprising:

a first computer readable program code device configured to lexically analyze

5 source code of said software,

a second computer readable program code device configured to parse the output of said first computer readable program code device,

a third computer readable program code device configured to build an intermediate representation of said parsed output, and

10 a fourth computer readable program code device configured to generate intermediate form code containing symbolic field references from said intermediate representation.

OAGOOGLE0000059227

said method comprising the steps of reading a portion of said computer readable code from said computer readable medium into a memory and writing said portion of said

15 code to a second computer readable medium.

add A1

DHEN110A.WP

70

OAGOOGLE0000059228

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NUMBER	FILING DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NO
08/755,764	11/21/96	GOSLING	J 149907-0064

B3M1/0924

JEFFREY J BLATT
IRELL & MANELLA
1800 AVENUE OF THE STARS
SUITE 900
LOS ANGELES CA 90067

EXAMINER	
HECKLER, T	
ART UNIT	PAPER NUMBER
2316	3

DATE MAILED: 09/24/97

This is a communication from the examiner in charge of your application.
COMMISSIONER OF PATENTS AND TRADEMARKS

OFFICE ACTION SUMMARY

☐ Responsive to communication(s) filed on ________.

☐ This action is **FINAL**.

☐ Since this application is in condition for allowance except for formal matters, **prosecution as to the merits is closed** in accordance with the practice under *Ex parte Quayle*, 1935 D.C. 11; 453 O.G. 213.

A shortened statutory period for response to this action is set to expire 3 (three) month(s), or thirty days, whichever is longer, from the mailing date of this communication. Failure to respond within the period for response will cause the application to become abandoned. (35 U.S.C. § 133). Extensions of time may be obtained under the provisions of 37 CFR 1.136(a).

Disposition of Claims

☒ Claim(s) 1-34 is/are pending in the application.

Of the above, claim(s) ________ is/are withdrawn from consideration.

☐ Claim(s) ________ is/are allowed.

☒ Claim(s) 1-34 is/are rejected.

☐ Claim(s) ________ is/are objected to.

☐ Claims ________ are subject to restriction or election requirement.

Application Papers

☒ See the attached Notice of Draftsperson's Patent Drawing Review, PTO-948.

☐ The drawing(s) filed on ________ is/are objected to by the Examiner.

☐ The proposed drawing correction, filed on ________ is ☐ approved ☐ disapproved.

☐ The specification is objected to by the Examiner.

☐ The oath or declaration is objected to by the Examiner.

Priority under 35 U.S.C. § 119

☐ Acknowledgement is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d).

☐ All ☐ Some* ☐ None of the CERTIFIED copies of the priority documents have been

☐ received.

☐ received in Application No. (Series Code/Serial Number) ________.

☐ received in this national stage application from the International Bureau (PCT Rule 17.2(a)).

*Certified copies not received: ________.

☐ Acknowledgement is made of a claim for domestic priority under 35 U.S.C. § 119(e).

Attachment(s)

☒ Notice of Reference Cited, PTO-892

☐ Information Disclosure Statement(s), PTO-1449, Paper No(s). ________

☐ Interview Summary, PTO-413

☒ Notice of Draftsperson's Patent Drawing Review, PTO-948

☐ Notice of Informal Patent Application, PTO-152

-- SEE OFFICE ACTION ON THE FOLLOWING PAGES --

1. The reissue oath or declaration filed with this application is defective because it fails to particularly specify the errors and/or how the errors relied upon arose or occurred as required under 37 CFR 1.175(a)(5). Included are inadvertent errors in conduct, i.e., actions taken by the applicant, the attorney or others, before the original patent issued, which are alleged to be the cause of the actual errors in the patent. This includes how and when the errors in conduct arose or occurred, as well as how and when these errors were discovered. Applicant's attention is directed to *Hewlett-Packard v. Bausch & Lomb*, 11 USPQ2d 1750, 1758 (Fed. Cir. 1989). The reissue declaration fails to specifically indicate the manner and details of how the errors arose and were discovered.

2. Claims 1-34 are rejected as being based upon a defective reissue declaration under 35 U.S.C. 251 as set forth above. See 37 CFR 1.175.

3. Claims 11-34 are rejected under 35 U.S.C. § 112, first paragraph, as the disclosure is enabling only for claims limited to a hybrid compiler-interpreter. See M.P.E.P. §§ 706.03(n) and 706.03(z).

The disclosure teaches that the invention is a hybrid compiler-interpreter at page 2 lines 29-32, page 4 lines 7-10, and Figure 3 element 38. The disclosure does not indicate that

the compiler and interpreter are separate and usable independent of each other. The disclosed invention is a combination of the compiler and interpreter used together. The subcombinations of a compiler and interpreter are not disclosed as being used separately.

4. The prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

5. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Tom Heckler whose telephone number is (703) 305-9666.

TH
September 23, 1997

THOMAS M. HECKLER
PRIMARY EXAMINER
ART UNIT 237

OAGOOGLE0000059279

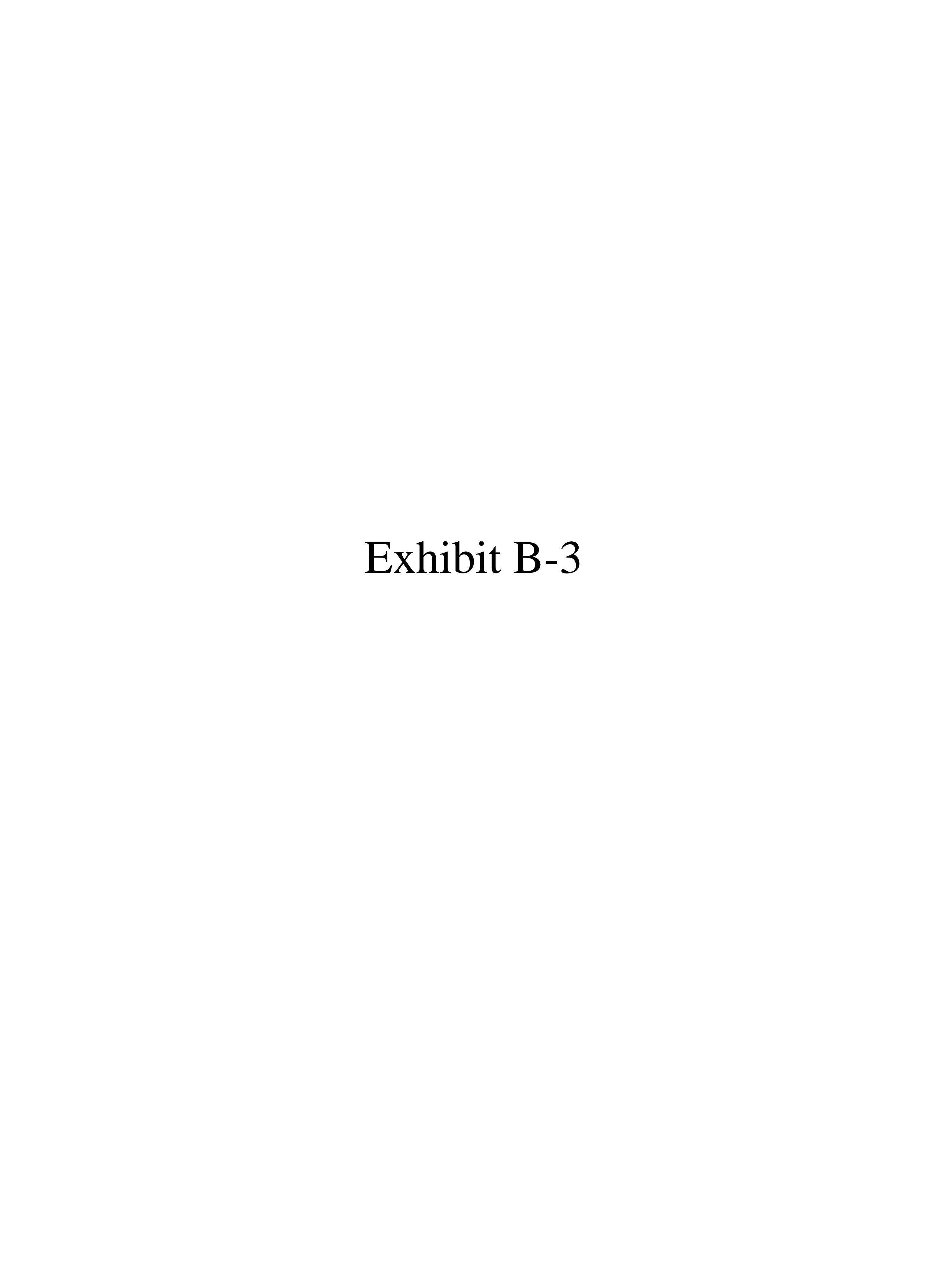


Exhibit B-3

149907-0064 PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re reissue application of U.S. Patent No. 5,367,685 Issued: November 22, 1994 Inventor: James Gosling For: METHOD AND APPARATUS FOR RESOLVING DATA REFERENCES IN GENERATED CODE	CERTIFICATE OF MAILING BY "EXPRESS MAIL" "EXPRESS MAIL" Mailing Label Number EM458150000US November 21, 1996 Date of Deposit I hereby certify that this paper or fee is being deposited with the United States Postal Service "EXPRESS MAIL POST OFFICE TO ADDRESSEE" service under 37 C.F.R. § 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231. Dee Henderson Typed or Printed Name of Person Mailing Paper or Fee [signature] Signature of Person Mailing Paper or Fee November 21, 1996 Date

REISSUE APPLICATION DECLARATION AND POWER OF ATTORNEY BY INVENTOR

Hon. Commissioner of
Patents and Trademarks
Washington, D.C. 20231

Sir:

I, James Gosling, declare:

1. I have reviewed and understand the contents of the specification and claims of the above-identified reissue application, and believe myself to be the only inventor of the invention described and claimed in the aforesaid reissue application and in U.S. Letters Patent No. 5,367,685 on which said reissue application is based.

DHEN10ED.WP

2. I do not know and do not believe that said invention was ever known or used in the United States of America before the invention thereof by myself.

3. U.S. Patent No. 5,367,685 is partly inoperative because it claims less than I had a right to claim in the patent (37 C.F.R. § 1.175(a)(3)).

4. The insufficiencies identified in paragraph 3 above arose as a result of errors on the part of applicant. The first error was failing to realize that the commercial embodiment of the invention was designed to be distributed in two parts, a compiler and an interpreter, which were to be packaged as two separate computer programs and which could be bought separately. My company and I realized on or around January of 1996 that competitors would probably also be shipping their competing products in two parts for the same market- and technology-related reasons that we were doing so. An infringer who shipped only one of those parts might argue that no claim reads on the compiler or the interpreter separately. I believe that I have the right to claim more specifically a compiler and an interpreter separately as supported by the original disclosure. This error may be remedied by the addition of claims 11-34 in the above-identified reissue application, which more specifically separately claim the compiler and the interpreter as disclosed in the application. Moreover, claiming the compiler and the interpreter separately, although they involve common inventive concepts, more closely corresponds to likely commercial embodiments for my invention.

5. A further error was a lack of specificity in the claim scope as regards what is done with the numerical values corresponding to symbolic references. In late 1995 or

OAGOOGLE0000059396

early 1996 a colleague suggested that an infringer could argue that the claims would not literally read on all possible ways of practicing the invention as regards numerical values corresponding to symbolic references. I believe I have a right to claim more specifically the handling of numerical values corresponding to symbolic references in my invention. This error may be remedied by the addition of claims 11 and 25 in the above-identified reissue application which more fully define the patentable aspects of my invention as supported by the disclosure.

6. A further error was that the claims may be challenged by an infringer as not reading literally on computer program code devices embodying the invention. Applicant is informed and believes that subsequent to the Commissioner of Patents' change of position in *In re Beauregard,* claims more specifically directed to computer program code devices are now permissible. I believe I have the right to further specifically claim computer program code devices which embody my invention as supported by the original disclosure. This error may be remedied by claims 29-32 of the above identified reissue application.

7. The errors identified above arose without any deceptive intention on the part of the undersigned or the assignee of the application on which U.S. Letters Patent No. 5,367,685 issued, Sun Microsystems Inc.

8. Claim 11 is directed to a method for interpreting software in an intermediate form code which includes instructions that contain symbolic references. In the method, instructions are interpreted in accordance with a program execution control. When an

OAGOOGLE0000059397

unresolved symbolic reference is encountered, a numerical value corresponding to the reference is determined and stored in memory. When a resolved symbolic reference is encountered, the instruction is interpreted by reading the stored numeric value. Claim 22 is directed to a corresponding method for compiling software into an intermediate form code which includes instructions that contain symbolic field references. In that method, the source code is lexically analyzed, the output of the lexical analysis step is parsed, an intermediate representation of the parsed output is built, and intermediate form code containing symbolic field references is generated from the intermediate representation. Both the interpreter claim 11 (and independent claims 25, 27, 29, 31, and 33) and the compiler claim 22 (as well as independent claims 26, 28, 30, 32, and 34) embody my original inventive concept, in that the compiler generates the intermediate form code, including symbolic references, which the interpreter interprets.

9. I acknowledge a duty to disclose information I am aware of which is material to the examination of this reissue application. The closest prior art known to applicant is that cited against the application which became original U.S. Patent No. 5,367,685.

10. I hereby revoke all previous powers of attorney and appoint as my attorneys of record in connection with this reissue application Matthew C. Rainey, Reg. No. 32,291; Jeffrey J. Blatt, Reg. No. 30,244; Robert Steinberg, Reg. No. 33,144; Bruce D. Kuyper, Reg. No. 33,937; Gary Frischling, Reg. No. 35,515; Robert Strawbrich, Reg. No. 36,692; and Wen Liu, Reg. No. 32,822. All communications regarding this reissue

OAGOOGLE0000059398

application are to be addressed to Jeffrey J. Blatt at Irell & Manella LLP, 1800 Avenue of the Stars, Suite 900, Los Angeles, CA 90067.

I declare further that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Full name of sole or first inventor James Gosling
Inventor's signature [signature]
Date 11/19/96 Country of Citizenship Canada
Residence 75 Fox Hollow Lane, Redwood City CA
Post Office Address P.O. Box 620509
Woodside CA 94062

DHEN10ED.WP

OAGOOGLE0000059399

3/B
T.L. 5-14-99

PATENT
Attorney Docket No. 06502.0083-02

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:

James GOSLING

Serial No.: Rule 53(b) continuation application of Serial No. 08/755,764

Filed: November 21, 1996

For: METHOD AND APPARATUS FOR RESOLVING DATA REFERENCES IN GENERATED CODE

Group Art Unit: Unassigned

Examiner: Unassigned

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

PRELIMINARY AMENDMENT

Prior to the examination of the above application, please amend this application as follows:

IN THE CLAIMS:

Please cancel claim 1 and add the following new claims 35-43:

11 --35. An apparatus comprising:

B1

a memory containing intermediate form object code constituted by a set of instructions, certain of said instructions containing one or more symbolic references; and

LAW OFFICES
NNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
SHINGTON, DC 20005
?02-408-4000

a processor configured to execute said instructions containing one or more symbolic references by determining a numerical reference corresponding to said symbolic reference, storing said numerical references, and obtaining data in accordance to said numerical references.

12 ~~36~~. A computer-readable medium containing instructions for controlling a data processing system to perform a method for interpreting intermediate form object code comprised of instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:

interpreting said instructions in accordance with a program execution control; and

B1 cont'd resolving a symbolic reference in an instruction being interpreted, said step of resolving said symbolic reference including the substeps of:

determining a numerical reference corresponding to said symbolic reference, and

storing said numerical reference in a memory.

13 ~~37~~. A computer-implemented method for executing instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:

resolving a symbolic reference in an instruction, said step of resolving said symbolic reference including the substeps of:

LAW OFFICES
INEGAN, HENDERSON,
ARABOW, GARRETT,
& DUNNER, L.L.P.
'0 I STREET, N. W.
NGTON, DC 20005
2-408-4000

OAGOOGLE0000059422

determining a numerical reference corresponding to said symbolic reference, and

storing said numerical reference in a memory.

14 ~~38~~. The method of claim 3 ~~37~~, wherein said substep of storing said numerical reference comprises the substep of replacing said symbolic reference with said numerical reference.

15 ~~39~~. The method of claim 3 ~~37~~, wherein said step of resolving said symbolic reference further comprises the substep of executing said instruction containing said symbolic reference using the stored numerical reference.

B1 cont'd 16 ~~40~~. The method of claim 3 ~~37~~, wherein said step of resolving said symbolic reference further comprises the substep of advancing program execution control after said substep of executing said instruction containing said symbolic reference.

17 ~~41~~. In a computer system comprising a program, a method for executing said program comprising the steps of:

receiving intermediate form object code for said program with symbolic data references in certain instructions of said intermediate form object code; and

converting the instructions of the intermediate form object code having symbolic data references, said converting step comprising the substeps of:

LAW OFFICES
NNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

13

OAGOOGLE0000059423

resolving said symbolic references to corresponding numerical references,

storing said numerical references, and

obtaining data in accordance to said numerical references.

18
~~42~~. A computer-implemented method for executing program operations, each operation being comprised of a set of instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of:

receiving a set of instructions reflecting each operation; and

performing an operation corresponding to the received set of instructions, wherein each of said symbolic references is resolved by determining a numerical reference corresponding to said symbolic reference, storing said numerical reference, and obtaining data in accordance to said stored numerical reference.

B1 cont'd 19
~~43~~. A memory for use in executing a program by a processor, the memory comprising:

intermediate form code containing symbolic field references associated with an intermediate representation of source code for the program,

the intermediate representation having been generated by lexically analyzing the source code and parsing output of said lexical analysis, and

wherein the symbolic field references are resolved by determining a numerical reference corresponding to said symbolic reference, and storing said numerical reference in a memory.--

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N.W.
WASHINGTON, D.C. 20005
202-408-4000

14

OAGOOGLE0000059424

<u>REMARKS</u>

This application is a continuation of reissue application Serial No. 08/755,764. Claims 1-34 of the parent application have been allowed. Thus, Applicant has cancelled those claims in this application.

Applicant submits this preliminary amendment for the Examiner to consider new claims 35-43.

The Commissioner is hereby authorized to charge any fees which may be required including fees due under 37 C.F.R. § 1.16 and any other fees due under 37 C.F.R. § 1.17, or credit any overpayment during the pendency of this application, to Deposit Account No. 06-0916.

Respectfully submitted,

FINNEGAN, HENDERSON, FARABOW, GARRETT & DUNNER, L.L.P.

By: [signature]
Jeffrey A. Berkowitz
Reg. No. 36,743

Dated: March 3, 1999

LAW OFFICES
FINNEGAN, HENDERSON, FARABOW, GARRETT, & DUNNER, L.L.P.
1300 I STREET, N.W.
WASHINGTON, D.C. 20005
202-408-4000

OAGOOGLE0000059425

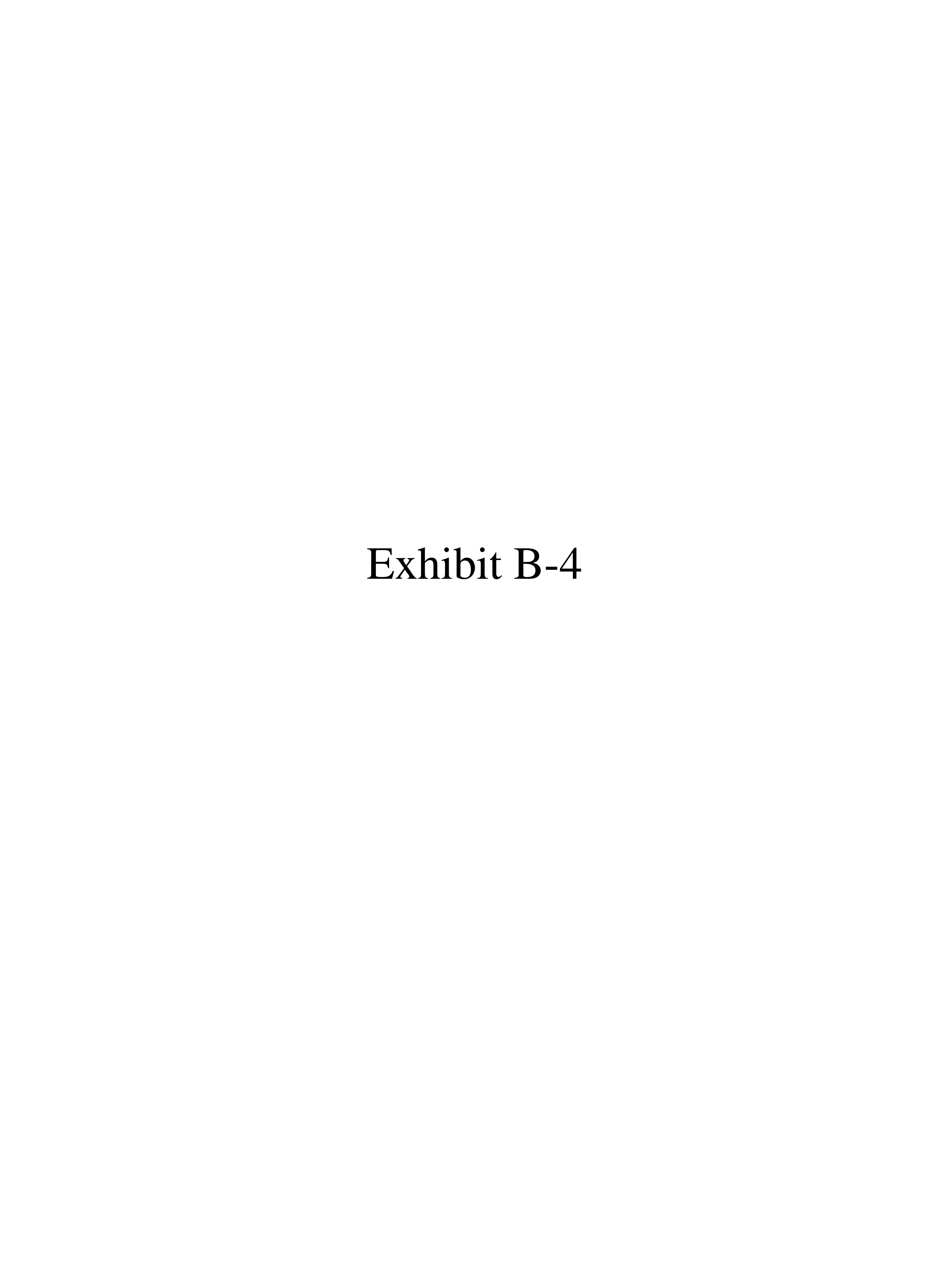


Exhibit B-4

UNITED STATES PATENT APPLICATION

of

FRANK YELLIN

and

RICHARD D. TUCK

for

METHOD AND SYSTEM FOR PERFORMING

STATIC INITIALIZATION

LAW OFFICES
INNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

Claims

What is claimed is:

1. A method in a data processing system for statically initializing an array, comprising the steps of:

compiling source code containing the array with static values to generate a class file with a clinit method containing byte codes to statically initialize the array to the static values;

receiving the class file into a preloader;

play executing the byte codes of the clinit method against a memory to identify the static initialization of the array by the preloader;

storing into an output file an instruction requesting the static initialization of the array; and

interpreting the instruction by a virtual machine to perform the static initialization of the array.

2. The method of claim 1 wherein the storing step includes step of:

storing a constant pool entry into the constant pool.

3. The method of claim 1 wherein the play executing step includes the steps of:

allocating a stack;

reading a byte code from the clinit method that manipulates the stack; and

performing the stack manipulation on the allocated stack.

LAW OFFICES
INNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

14

OAGOOGLE0000057480

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR		ATTORNEY DOCKET NO.
09/055,947	04/07/98	YELLIN	F	06502.0046

LM02/0721

FINNEGAN HENDERSON FARABOW GARRETT
& DUNNER
1300 I STREET NW
WASHINGTON DC 20005

EXAMINER	
BOOKER, K	
ART UNIT	PAPER NUMBER
2762	3

DATE MAILED: 07/21/99

Please find below and/or attached an Office communication concerning this application or proceeding.

Commissioner of Patents and Trademarks

PTO-90C (Rev. 2/95)

1- File Copy

OAGOOGLE0000057503

Office Action Summary	Application No. 09/055,947	Applicant(s) Frank Yellin, Richard D. Tuck
	Examiner Kelvin E. Booker	Group Art Unit 2762

☐ Responsive to communication(s) filed on ____________.

☐ This action is **FINAL.**

☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11; 453 O.G. 213.

A shortened statutory period for response to this action is set to expire *3* month(s), or thirty days, whichever is longer, from the mailing date of this communication. Failure to respond within the period for response will cause the application to become abandoned. (35 U.S.C. § 133). Extensions of time may be obtained under the provisions of 37 CFR 1.136(a).

Disposition of Claims

☒ Claim(s) *1-23* is/are pending in the application.

Of the above, claim(s) ____________ is/are withdrawn from consideration.

☒ Claim(s) *6-23* is/are allowed.

☒ Claim(s) *1 and 3* is/are rejected.

☒ Claim(s) *2, 4, and 5* is/are objected to.

☐ Claims ____________ are subject to restriction or election requirement.

Application Papers

☐ See the attached Notice of Draftsperson's Patent Drawing Review, PTO-948.

☒ The drawing(s) filed on *Apr 7, 1998* is/are objected to by the Examiner.

☐ The proposed drawing correction, filed on ____________ is ☐approved ☐disapproved.

☐ The specification is objected to by the Examiner.

☐ The oath or declaration is objected to by the Examiner.

Priority under 35 U.S.C. § 119

☐ Acknowledgement is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d).

☐ All ☐ Some* ☐ None of the CERTIFIED copies of the priority documents have been

☐ received.

☐ received in Application No. (Series Code/Serial Number) ____________.

☐ received in this national stage application from the International Bureau (PCT Rule 17.2(a)).

*Certified copies not received: ____________.

☐ Acknowledgement is made of a claim for domestic priority under 35 U.S.C. § 119(e).

Attachment(s)

☒ Notice of References Cited, PTO-892

☒ Information Disclosure Statement(s), PTO-1449, Paper No(s). *2*

☐ Interview Summary, PTO-413

☐ Notice of Draftsperson's Patent Drawing Review, PTO-948

☐ Notice of Informal Patent Application, PTO-152

--- SEE OFFICE ACTION ON THE FOLLOWING PAGES ---

OAGOOGLE0000057504

DETAILED ACTION

Drawings

1. This application has been filed with informal drawings which are acceptable for examination purposes only. Formal drawings will be required when the application is allowed.

Claim Rejections - 35 USC § 102

2. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless --
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

3. **Claims 1 and 3** are rejected under 35 U.S.C. 102(b) as being anticipated by Cierniak, "Briki: an optimizing Java compiler".

As per claim 1, Cierniak teaches of a method for statically initializing an array comprising the steps of:

a. compiling source code containing an array with static values to generate a class file with a class initialization (cinit) method (see page 179, abstract and section 1, generating JavaIR (Java Intermediate Representation));

OAGOOGLE0000057505

b. receiving a class file into a preloader (see page 181, section 3, recovering high-level structure);

c. compare the execution of byte codes of the cinit method against memory to identify the static initialization of an array by the preloader (see page 181, section 3, mapping between JavaIR and Java Source);

d. storing into an output file an instruction requesting the static initialization of an array (see page 183, section 4.1, array layout optimization); and

e. interpreting the instruction by a virtual machine to perform the static initialization of the array (see page 182, section 3.2, symbolic emulation).

As per claim 3, Cierniak teaches of a method whereby play executing comprise the steps:

a. allocating a stack (see page 182, section 3.2, stack allocation);

b. reading byte code from the clint method that manipulates the stack (see page 182, section 3.2, stack recovery); and

c. performing stack manipulation on the allocated stack (see page 182, sections 3.2-3.3, stack recovery and manipulation).

Allowable Subject Matter

4. **Claims 2, 4 and 5** are objected to as being dependent upon a rejected base claim, but would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claims. **Claims 6-23** are allowed.

5. The following is a statement of reasons for the indication of allowable subject matter: the cited prior art above in the examination fails to teach of:

a. constant pool storage; and

b. explicit operations (e.g., initialization, allocation, manipulation and simulation) with respect to "play" execution.

Conclusion

An inquiry concerning this communication or earlier communications from the examiner should be directed to Kelvin Booker whose telephone number is (703) 308-4088. The examiner can normally be reached on Monday-Thursday from 7:00 AM-5:30 PM EST.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Tariq Hafiz, can be reached on (703) 305-9643. The fax phone number for the organization where this application or proceeding is assigned is (703) 308-1396.

An inquiry of a general nature or relating to the status of this application proceeding should be directed to the receptionist whose telephone number is (703) 305-3900.

Kelvin E. Booker

Patent Examiner

Group Art Unit 2762

2762

PATENT
Attorney Docket No. **06502.0046-00**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE #4 10-21-99 P.2

OIPE JC77 OCT 18 1999 PATENT & TRADEMARK OFFICE

In re Application of:	)	
Frank Yellin et al.	)	
Serial No.: 09/055,947	)	Group Art Unit: 2762
Filed: April 7, 1998	)	Examiner: Kelvin E. Booker
For: METHOD AND SYSTEM FOR PERFORMING STATIC INITIALIZATION	)	

RECEIVED OCT 19 1999

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

AMENDMENT

Applicants submit this amendment in response to the Office Action dated July 21, 1999.

IN THE CLAIMS:

Please amend claim 1 as follows:

1. (Amended) A method in a data processing system for statically initializing an array, comprising the steps of:

compiling source code containing the array with static values to generate a class file with a clinit method containing byte codes to statically initialize the array to the static values;

receiving the class file into a preloader;

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, D.C. 20005
202-408-4000

OAGOOGLE0000057510

simulating execution of [play executing] the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader;

storing into an output file an instruction requesting the static initialization of the array; and

interpreting the instruction by a virtual machine to perform the static initialization of the array.

Q'

REMARKS

Claims 1-23 are pending in the application. In the Office Action, the Examiner rejected claims 1 and 3 under 35 U.S.C. §102(b) as being anticipated by Cierniak, "*Briki: an optimizing Java compiler*"; objected to claims 2, 4, and 5 as depending upon a rejected base claim; and allowed claims 6-23. Responsive to the rejection of claims 1 and 3, applicants have amended claim 1 to more particularly point out and distinctly claim the subject matter of applicants' invention.

Applicants wish to thank the Examiner for his consideration during the telephone interview with applicants' attorney on October 13, 1999. During the interview, applicants' attorney and the Examiner agreed on an amendment to claim 1 that would further clarify the distinctions between the claim and the cited art. Applicants' attorney and the Examiner also agreed that this amendment rendered all of the pending claims allowable over the cited art, although a subsequent search by the Examiner would be performed.

Based upon the above amendments and remarks, applicants submit that all of the pending

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, D. C. 20005
202-408-4000

OAGOOGLE0000057511

claims are either allowed or clearly allowable, and thus, applicants request the issuance of a Notice of Allowance. Additionally, applicants respectfully request that the Examiner call applicants' attorney if it would expedite prosecution.

Please grant any extensions of time required to enter this response and charge any additional required fees to our deposit account 06-0916.

Respectfully submitted,

FINNEGAN, HENDERSON, FARABOW,
GARRETT & DUNNER, L.L.P.

By: ____________________
Michael L. Kiklis
Reg. No. 38.939

Dated: October 18, 1999

RECEIVED
OCT 19 1999
TC [illegible] ROOM

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, D. C. 20005
202-408-4000

OAGOOGLE0000057512

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR		ATTORNEY DOCKET NO.
09/055,947	04/07/98	YELLIN	F	06502.0046

LM71/0104

FINNEGAN HENDERSON FARABOW GARRETT
& DUNNER
1300 I STREET NW
WASHINGTON DC 20005

EXAMINER	
BOOKER,K	
ART UNIT	**PAPER NUMBER**
2762	

DATE MAILED: 01/04/00

Please find below and/or attached an Office communication concerning this application or proceeding.

Commissioner of Patents and Trademarks

PTO-90C (Rev. 2/95)

1- File Copy

OAGOOGLE0000057513

Notice of Allowability	Application No. 09/055,947	Applicant(s) Yellin et al.
	Examiner Kelvin E. Booker	Group Art Unit 2762

All claims being allowable, PROSECUTION ON THE MERITS IS (OR REMAINS) CLOSED in this application. If not included herewith (or previously mailed), a Notice of Allowance and Issue Fee Due or other appropriate communication will be mailed in due course.

☒ This communication is responsive to *Amendment filed October 18, 1999 (see paper no. 4)*

☒ The allowed claim(s) is/are *1-23*

☐ The drawings filed on ____________ are acceptable.

☐ Acknowledgement is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d).

☐ All ☐ Some* ☐ None of the CERTIFIED copies of the priority documents have been

☐ received.

☐ received in Application No. (Series Code/Serial Number) ____________ .

☐ received in this national stage application from the International Bureau (PCT Rule 17.2(a)).

*Certified copies not received: ____________

☐ Acknowledgement is made of a claim for domestic priority under 35 U.S.C. § 119(e).

A SHORTENED STATUTORY PERIOD FOR RESPONSE to comply with the requirements noted below is set to EXPIRE **THREE MONTHS** FROM THE "DATE MAILED" of this Office action. Failure to timely comply will result in ABANDONMENT of this application. Extensions of time may be obtained under the provisions of 37 CFR 1.136(a)

☐ Note the attached EXAMINER'S AMENDMENT or NOTICE OF INFORMAL APPLICATION, PTO-152, which discloses that the oath or declaration is deficient. A SUBSTITUTE OATH OR DECLARATION IS REQUIRED.

☒ Applicant MUST submit NEW FORMAL DRAWINGS

☒ because the originally filed drawings were declared by applicant to be informal.

☐ including changes required by the Notice of Draftsperson's Patent Drawing Review, PTO-948, attached hereto or to Paper No. ______ .

☐ including changes required by the proposed drawing correction filed on ____________ , which has been approved by the examiner.

☐ including changes required by the attached Examiner's Amendment/Comment.

Identifying indicia such as the application number (see 37 CFR 1.84(c)) should be written on the reverse side of the drawings. The drawings should be filed as a separate paper with a transmittal lettter addressed to the Official Draftsperson.

☐ Note the attached Examiner's comment regarding REQUIREMENT FOR THE DEPOSIT OF BIOLOGICAL MATERIAL.

Any response to this letter should include, in the upper right hand corner, the APPLICATION NUMBER (SERIES CODE/SERIAL NUMBER). If applicant has received a Notice of Allowance and Issue Fee Due, the ISSUE BATCH NUMBER and DATE of the NOTICE OF ALLOWANCE should also be included.

Attachment(s)

- ☒ Notice of References Cited, PTO-892
- ☐ Information Disclosure Statement(s), PTO-1449, Paper No(s). ________
- ☐ Notice of Draftsperson's Patent Drawing Review, PTO-948
- ☐ Notice of Informal Patent Application, PTO-152
- ☒ Interview Summary, PTO-413
- ☐ Examiner's Amendment/Comment
- ☐ Examiner's Comment Regarding Requirement for Deposit of Biological Material
- ☒ Examiner's Statement of Reasons for Allowance

Tariq R. Hafiz
Supervisory Patent Examiner
Technology Center 2700

OAGOOGLE0000057514

Allowable Subject Matter

1. The following is an examiner's statement of reasons for allowance:

the cited prior art, either singly or in combination, fails to anticipate or render obvious the simulation of execution with respect to class initialization, without executing byte codes.

Cierniak, "Briki: An Optimizing Java Compiler", teaches of comparing the execution of byte codes, but fails to address the issue of simulating the process without execution.

Any comments considered necessary by applicant must be submitted no later than the payment of the issue fee and, to avoid processing delays, should preferably accompany the issue fee. Such submissions should be clearly labeled "Comments on Statement of Reasons for Allowance."

Conclusion

2. An inquiry concerning this communication or earlier communications from the examiner should be directed to Kelvin Booker whose telephone number is (703) 308-4088. The examiner can normally be reached on Monday-Thursday from 7:00 AM-5:30 PM EST.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Tariq Hafiz, can be reached on (703) 305-9643. The fax phone number for the organization where this application or proceeding is assigned is (703) 308-1396.

OAGOOGLE0000057515

An inquiry of a general nature or relating to the status of this application proceeding should be directed to the receptionist whose telephone number is (703) 305-3900.

Kelvin E. Booker

Patent Examiner

Group Art Unit 2762

Tariq R. Hafiz
Supervisory Patent Examiner
Technology Center 2700

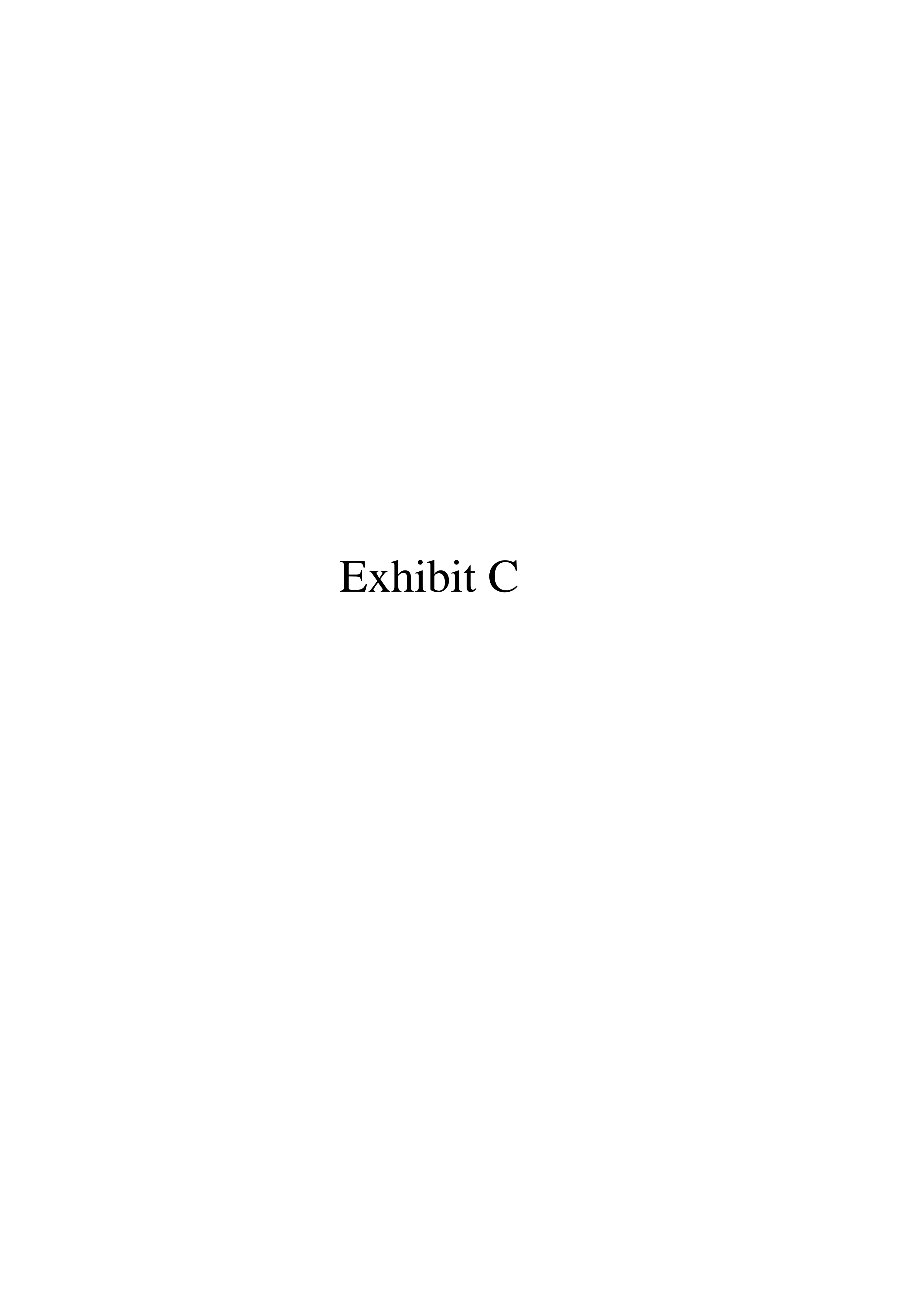


Exhibit C

EXEMPLARY LIST OF SUN PATENTS

The following patents are examples of Sun patents claiming carrier waves or including carrier waves in definitions of "computer-readable medium" or similar terms.

Filing Date	Patent Number	Example Citation
7/1/96	US 5,953,522	17:32-34 (Claim “15. The computer program product of claim 14 wherein the computer readable medium is a data signal embodied in a carrier wave.”)
4/23/97	US 5,903,899	17:30–39 (“Such implementation may comprise a series of computer instructions either fixed on a tangible medium, such as a computer readable media, e.g. diskette, CD-ROM, ROM, or fixed disk, or transmittable to a computer system, via a modem or other interface device, such as communications adapter connected to a network over a medium. Medium can be either a tangible medium, including but not limited to optical or analog communications lines, or may be implemented with wireless techniques, including but not limited to microwave, infrared or other transmission techniques.”)
6/30/97	US 5,978,588	10:54–58 (Claim “3. A computer data signal embodied in a carrier wave and representing sequences of instructions which, when executed by a processor, cause said processor to compile a source program into an object program, optimally placing code blocks of the source program, by performing the steps of . . .”)
10/6/97	US 5,970,249	16:51–53 (Claim “25. A computer-readable medium as recited in claim 24 wherein the computer program code devices are embodied in a carrier wave.”)
12/11/97	US 6,047,377	5:7–15 (“Common forms of computer-readable media include, for example, a floppy disk, a flexible

		disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.")
12/11/97	US 6,044,467	5:26–34 ("Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.")
12/12/97	US 5,946,489	11:1–7 ("Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described below, or any other medium from which a computer can read.")
5/27/98	US 5,983,021	6:44–51 ("Also, although aspects of one embodiment are depicted as being stored in memory 303, one skilled in the art will appreciate that systems and methods consistent with the present invention may be stored on other computer-readable media, such as secondary storage devices, like hard disks, floppy disks, and CD-ROM; a carrier wave received from the Internet 302; or other forms of ROM or RAM.")
6/29/98	US 6,115,715	16:42–47 (Claim "15. A computer data signal embodied in a carrier wave and representing sequences of instructions arranged to update a Java system configuration database having a client schema containing persistent and transient data, and

		a server schema containing persistent data, the sequence of instructions comprising: . . .")
6/30/98	US 6,272,517	13:41–45 (Claim "22. A computer data signal embodied in a carrier wave and representing sequences of instructions which, when executed by a processor, cause the processor to share a time quantum between threads in a process by performing the steps of: . . .")
6/30/98	US 6,271,838	5:16–53 ("The term "computer-readable medium" as used herein refers to any media that participates in providing instructions to processor 604 for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device 610. Volatile media includes dynamic memory, such as Memory 606. Transmission media includes coaxial cables, copper wire, and fiber optics, including the wires that comprise bus 602. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications. Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punch cards, papertape, any other physical medium with patterns of holes, a RAM, PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read. Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor 604 for execution. For example, the instructions may initially be carried on magnetic disk of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system 600 can receive the data on the telephone line and use an infra-red transmitter to

		convert the data to an infra-red signal. An infra-red detector coupled to bus 602 can receive the data carried in the infra-red signal and place the data on bus 602. Bus 602 carries the data to memory 606, from which processor 604 retrieves and executes the instructions. The instructions received by memory 606 may optionally be stored on storage device 610 either before or after execution by processor 604.”)
10/15/99	US 6,853,868	10:10–15 (“The computer-readable medium can also be distributed as a data signal embodied in a carrier wave over a network of coupled computer systems so that the computer-readable code is stored and executed in a distributed fashion.”)
5/19/00	US 6,542,920	17:50–18:3 (“The term "computer-readable medium" as used herein refers to any medium that participates in providing instructions to processor 604 for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic to disks, such as storage device 610. Volatile media includes dynamic memory, such as main memory 606. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus 602. Transmission media can also take the form of acoustic or electromagnetic waves, such as those generated during radio-wave, infra-red, and optical data communications. Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.”)
5/19/00	US 6,938,085	14:4–11 (“Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical

		medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.”)
9/19/01	US 6,499,049	16:45–52 (“Additionally, although aspects of the present invention are described as being stored in memory, one skilled in the art will appreciate that these aspects can also be stored on or read from other types of computer-readable media, such as secondary storage devices, like hard disks, floppy disks, or CD-ROM; a carrier wave from the Internet; or other forms of RAM or ROM, either currently know or later developed.”)
4/1/02	US 6,983,455	10:1–8 (“Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.”)
5/21/03	US 6,952,760	4:47–61 (“Although aspects of methods, systems, and articles of manufacture consistent with the present invention are depicted as being stored in memory 106, one skilled in the art will appreciate that these aspects may be stored on or read from other computer-readable media, such as secondary storage devices, like hard disks, floppy disks, and CD-ROMs; a carrier wave received from a network such as the Internet; or other forms of ROM or RAM either currently known or later developed.”)
4/29/04	US 6,980,916	6:39–53 (“Software programming code, which embodies aspects of the present invention, is typically maintained in permanent storage, such as a computer readable medium. In a client/server environment, such software programming code may be stored on a client or a server. The software programming code may be embodied on any of a

		variety of known media for use with a data processing system, This includes, but is not limited to, magnetic and optical storage devices such as disk drives, magnetic tape, compact discs (CD's), digital video discs (DVD's), and computer instruction signals embodied in a transmission medium with or without a carrier wave upon which the signals are modulated. For example, the transmission medium may include a communications network, such as the Internet.")
7/28/05	US 7,278,132	5:27–31 ("Storage 112 may also include computer-readable media such as magnetic tape, flash memory, signals embodied on a carrier wave, PC-CARDS, portable mass storage devices, holographic storage devices, and other storage devices.") 6:34–38 ("The computer-readable medium can also be distributed as a data signal embodied in a carrier wave over a network of coupled computer systems so that the computer-readable code is stored and executed in a distributed fashion.")

More examples may be found by searching the U.S. Patent and Trademark Office online database. U.S. Patent Full-Text Database Manual Search, *available at* http://patft.uspto.gov/netahtml/PTO/search-adv.htm. One such search, which identified 460 candidate patents as of March 17, 2011, is:

AN/"Sun Microsystems" and "carrier wave" and "computer-readable medium"

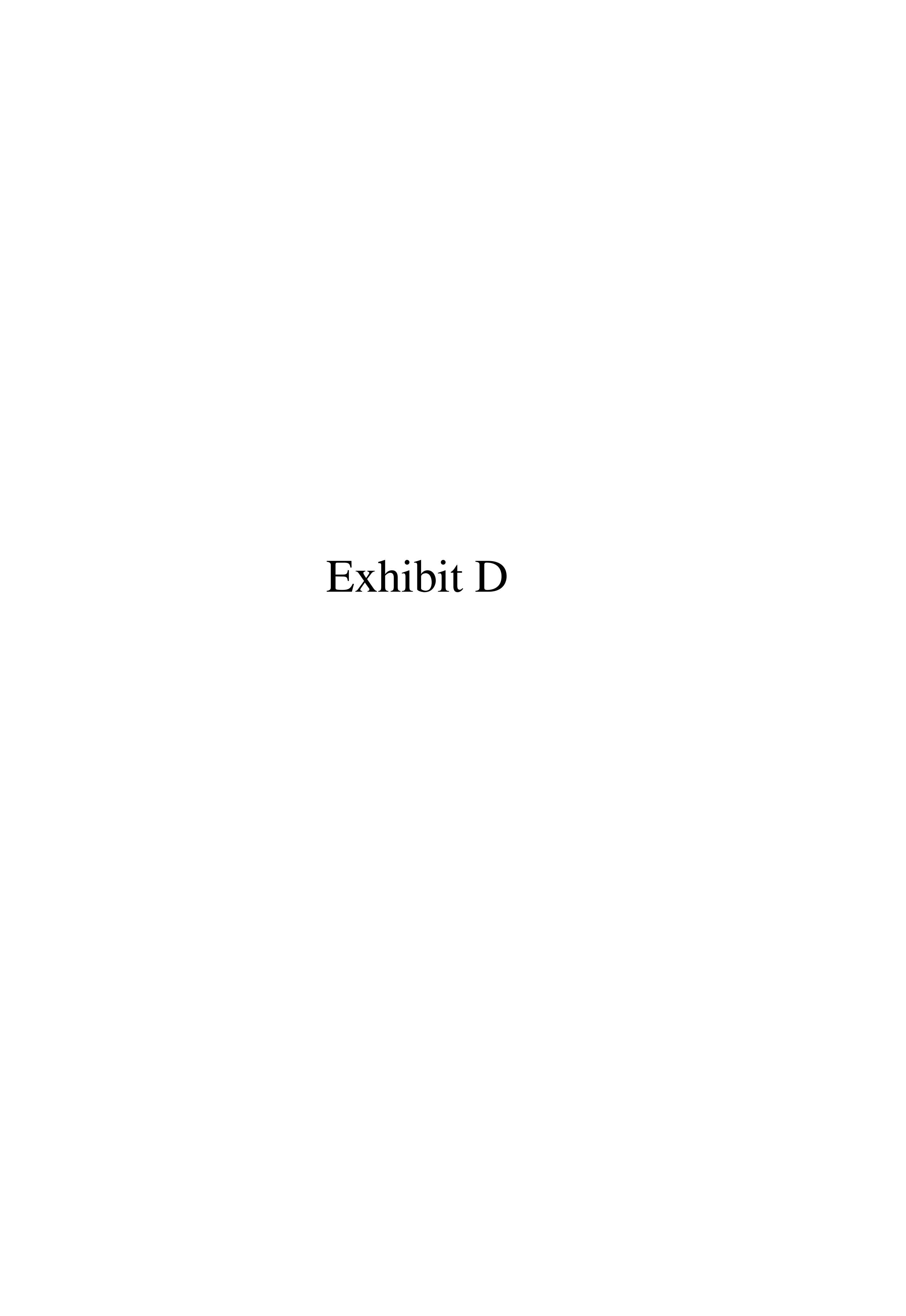


Exhibit D

Subject Matter Eligibility of Computer Readable Media

Subject Matter Eligibility of Computer Readable Media

The United States Patent and Trademark Office (USPTO) is obliged to give claims their broadest reasonable interpretation consistent with the specification during proceedings before the USPTO. See In re Zletz, 893 F.2d 319 (Fed. Cir. 1989) (during patent examination the pending claims must be interpreted as broadly as their terms reasonably allow). The broadest reasonable interpretation of a claim drawn to a computer readablle medium (also called machine readable medium and other such variations) typically covers forms of non-transitory tangible media and transitory propagating signals per se in view of the ordinary and customary meaning of computer readable media, particularly when the specification is silent. See MPEP 2111.01. When the broadest reasonable interpretation of a claim covers a signal per se, the claim must be rejected under 35 U.S.C. § 101 as covering non-statutory subject matter. See In re Nuijten, 500 F.3d 1346, 1356-57 (Fed. Cir. 2007) (transitory embodiments are not directed to statutory subject matter) and Interim Examination Instructions for Evaluating Subject Matter Eligibility Under 35 U.S.C. § 101, Aug. 24, 2009; p. 2.

The USPTO recognizes that applicants may have claims directed to computer readable media that cover signals per se, which the USPTO must reject under 35 U.S.C. § 101 as covering both non-statutory subject matter and statutory subject matter. In an effort to assist the patent community in overcoming a rejection or potential rejection under 35 U.S.C. § 101 in this situation, the USPTO suggests the following approach. A claim drawn to such a computer readable medium that covers both transitory and non-transitory embodiments may be amended to narrow the claim to cover only statutory embodiments to avoid a rejection under 35 U.S.C. § 101 by adding the limitation "non-transitory" to the claim. Cf. Animals - Patentability, 1077 Off. Gaz. Pat. Office 24 (April 21, 1987) (suggesting that applicants add the limitation "non-human" to a claim covering a multi-cellular organism to avoid a rejection under 35 U.S.C. § 101). Such an amendment would typically not raise the issue of new matter, even when the specification is silent because the broadest reasonable interpretation relies on the ordinary and customary meaning that includes signals per se. The limited situations in which such an amendment could raise issues of new matter occur, for example, when the specification does not support a non-transitory embodiment because a signal per se is the only viable embodiment such that the amended claim is impermissibly broadened beyond the supporting disclosure. See, e.g., Gentry Gallery, Inc. v. Berkline Corp., 134 F.3d 1473 (Fed. Cir. 1998).

DAVID J. KAPPOS
Under Secretary of Commerce for
Intellectual Property and
Director of the United States Patent
and Trademark Office

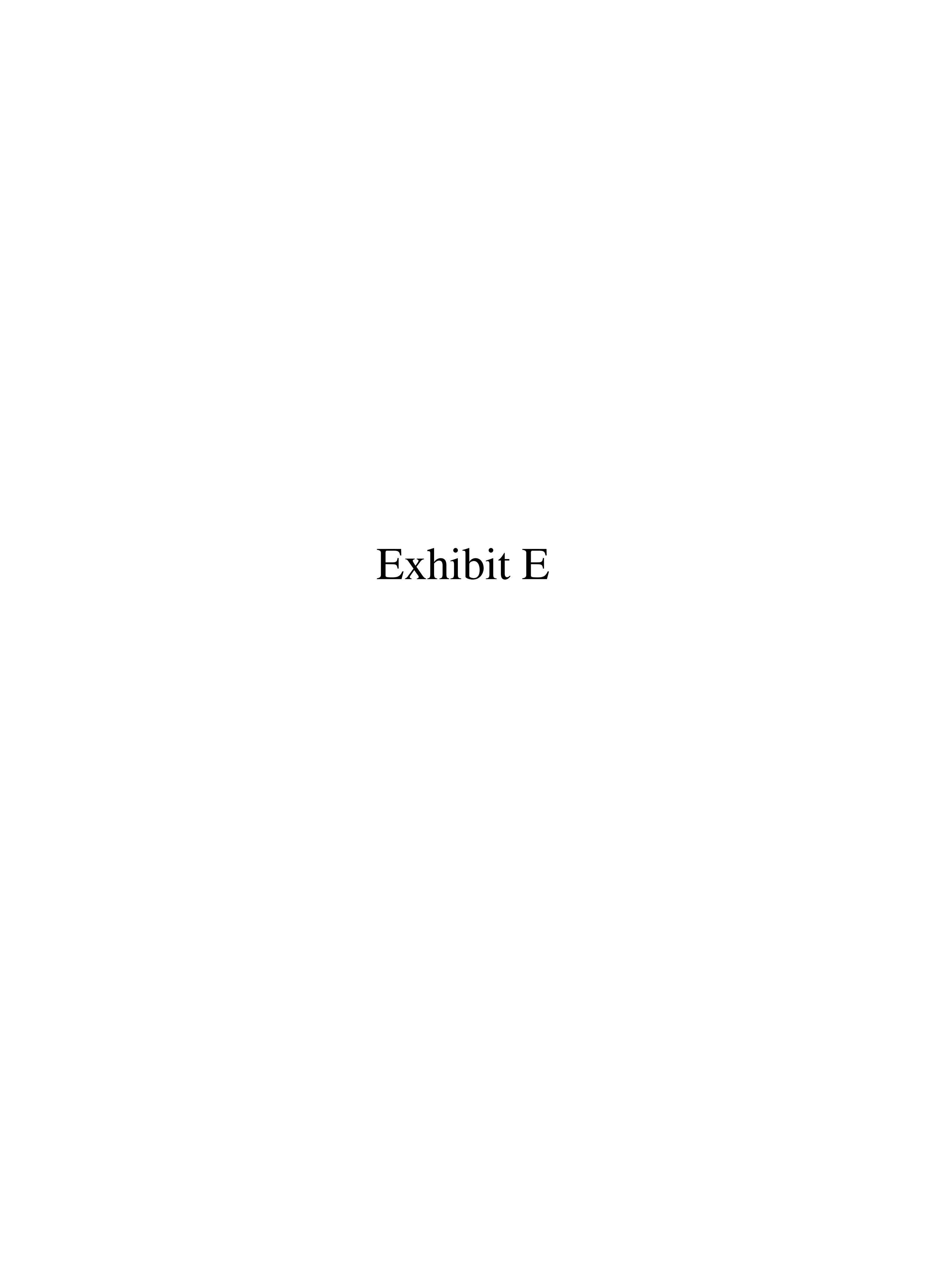


Exhibit E

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(12), 1253–1268 (OCTOBER 1998)

Tailored Compression of Java Class Files

R. NIGEL HORSPOOL AND JASON CORLESS
Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6
(email: nigelh@csr.uvic.ca)

SUMMARY

Java class files can be transmitted more efficiently over a network if they are compressed. After an examination of the class file structure and obtaining statistics from a large collection of class files, we propose a compression scheme that is tailored to class files. Our scheme achieves significantly better compression than commonly used methods such as ZIP. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: data compression; Java; class file

INTRODUCTION

The Java programming language[1] and its implementation using a Java Virtual Machine (JVM)[2] have greatly simplified the task of developing web-based application programs. In this and in other roles, Java has been a runaway success. When a Java program is compiled, it is translated into a collection of *class* files. Each class file contains a variety of components, including instructions for the JVM as well as data constants, interface specifications and other information. When a remote user executes a Java applet, the class files are downloaded over the internet onto the user's machine and interpretively executed by a copy of the JVM on that machine. An alternative is that the class files might be translated on the user's machine into native machine code using a Just-In-Time (JIT) Java compiler.

It is clearly advantageous for the Java class files to be made as small as possible. The smaller the file, the shorter the transmission time to deliver the file to its destination. If the user is being charged for connect time or for the number of data packets delivered, then smaller files would also have an economic benefit. Transmission of a class file in a compressed format and decompressing the file on the user's machine should improve overall performance provided that the decompression process consumes reasonable amounts of computation, and does not occupy excessive amounts of main memory.

There are many general purpose data compression programs which could be used to reduce the size of a Java class file. Some examples of widely available compression programs are *gzip*, *zip*, and *compress*.[3] However, they tend not to be as effective on Java class files as on file formats. The reason is that these compression programs work by finding repetitions and by coding the repeated patterns of data in an efficient

CCC 0038–0644/98/121253–16$17.50
© 1998 John Wiley & Sons, Ltd.

Received 10 September 1997
Revised 19 March 1998
Accepted 17 April 1998

manner. The longer the input, the more opportunity there is for finding repetitions and the better the compression usually becomes. Unfortunately, a typical class file for a Java program is quite short, perhaps just a few hundred bytes in size, and the file is organized into several sections. Each section contains data in a different format. It is unlikely that the data in one section of a class file will repeat any patterns of bytes encountered in a previous section of the file. Given the short size of a class file and the fragmented nature of each file, a general-purpose compression algorithm has too little opportunity to adapt to a class file section before the end of that section is reached.

There has been some work on compression methods that are specifically targeted to executable files or tuned for such files. Recently, Ernst *et al*.[4] reported a compressed 'wire' representation that reduces SPARC code to 21 per cent of its original size. However, as we will argue, their techniques would be relatively ineffective on Java class files because of their small size. Yu's[5] approach achieves excellent compression on executable files and is well suited to its main task of compressing software that is being distributed on floppy disks. It too benefits from having reasonably large quantities of data to compress and would also be ineffectual if applied separately to small files like the Java class files. Yu's algorithm is based on LZSS[3,6] with a non-greedy matching heuristic and, in that regard, is similar to gzip.

If we wish to achieve good compression on Java class files, the only realistic approach is to develop a compression program which has been customized to the Java class file format. Such a program would waste very little time adapting itself to a particular file and would be able to achieve some compression on even the smallest files.

In this paper, we report on a compression/decompression program named clazz which was developed specifically for Java class files. Our program outperforms both gzip and bzip2[7] by a wide margin. The bzip2 method is close to being the best available general-purpose compression method for text files. Detailed information about the clazz program and the experimental methodology is available in the second author's MSc dissertation.[8]

The following sections of this paper provide an overview of the Java class file structure, then explain how we developed a customized compression method, and finally report on how well our method works on a collection of class files.

STRUCTURE OF JAVA PROGRAMS AND JAVA CLASS FILES

A Java program could comprise a stand-alone application program or an applet to be invoked from a web page. In either form, the program will have been constructed as a collection of Java classes. It is a requirement of some of the standard Java compilers that each class declared as 'public' must be compiled separately. In effect, a Java source code file should contain a declaration of exactly one public class plus, optionally, some declarations of private classes. Each source code file is translated by the compiler into a so-called *class file*. For example, if an application program is constructed from source code files named Main.java, One.java and Two.java then the compilation command

```
javac Main.java One.java Two.java
```

using Sun's JDK implementation on a Sun workstation will create three files with the names Main.class, One.class and Two.class.

A group of related class files may be stored together on the computer's hard drive as a *package*. It is a common practice to use zip file format[9] for this collection of files, thus making it simpler to treat a package as a single entity while also saving disk storage. The Java Archive (JAR) format also provides a mechanism for grouping class files into a single entity for shipping over a network connection. It too is based on the zip file format, although the capability of compressing members of the archive does not appear to be used by Sun when distributing their class file libraries. If the compression capabilities of the zip file or JAR file format are used, it should not be forgotten that the zip file or JAR file is constructed from independently created class files, and that individual class files can be extracted from the collection and used. The files are compressed independently in order to facilitate extraction and replacement of a single file. Consequently, no benefit to compression performance is obtained by combining several small files into a single archive.

A Java program is normally executed by invoking the Java interpreter, specifying to it the class file where execution is to begin. A less common alternative, but one that is growing in popularity as JIT compiler technology develops, is to translate the bytecode of each method into native code when it is invoked for the first time. For a stand-alone application, execution begins at a standard method named main; for applets, the class file has to implement other standard methods. While the program executes, it will occasionally make a reference to a class type defined in another file and which has not been previously accessed. In this case, the corresponding class file must be dynamically loaded before execution can continue. For a program invoked as an applet from a web browser, dynamic loading will typically require that the class file be fetched from a host computer elsewhere on the network.

Regardless of the architecture of the client computer where the Java program is to run, a class file always has the same format. The instructions in the class file are instructions for a Java Virtual Machine (JVM).[2] The client computer would execute the class file either by using an interpreter for the JVM or a JIT compiler. However, to maintain architecture neutrality, JIT translation is performed on the client computer. The class file is in the standard format and not in native code form when it is fetched over the network.

A slightly simplified picture of the overall layout of a class file is shown in Figure 1. What should be observed from the picture is that the class file is organized as a series of sections. One section in particular, the *Methods Section*, is itself organized as a series of method entries, where each entry is itself subdivided into sections. We explain the structure of some of the more important sections below.

The Constant Pool

Constants occurring in a Java source code file are converted by the compiler into entries in the constant pool. The compiler may also create many additional entries for constants which do not explicitly appear in the source code but which are needed at execution time. As represented in a class file, each entry in the constant pool consists of a tag byte followed by a group of bytes that contain the value of the constant. For example, a UTF8 string constant containing 10 characters would be stored as a 13-byte entry in the constant pool. The first byte is a tag with value 1,

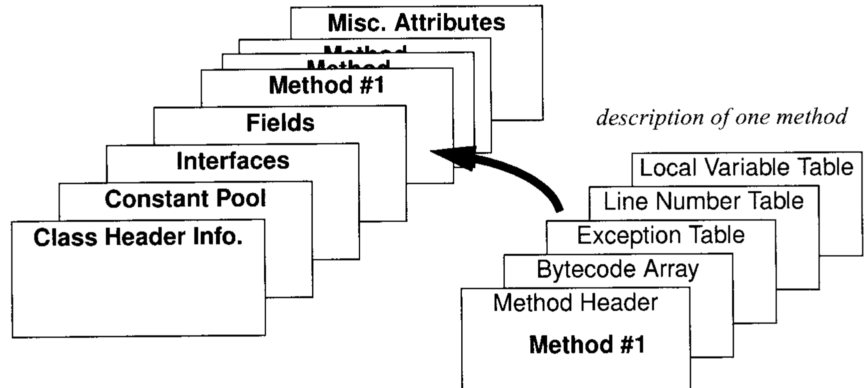


Figure 1. Class file layout

the next two bytes hold the string length, and the following 10 bytes hold the characters of the string constant.

The constant pool produced for the small sample program of Figure 2 is shown in Figure 3. Each entry is written as a tag name followed, in square brackets, by the additional information required for that tag.

The relatively large size of the constant pool compared to the original program should be noted. Many field names, class names, and type signatures are included as character strings.

```
class Small {
        static final int CUTOFF = 8;

        public int foobar( int a, int b ) {
                int val = a + b;
                if (val > CUTOFF) {
                        return 4*val;
                } else {
                        return 3*val;
                }
        }

        static void main( String args[] ) {
                Small s = new Small();
                System.out.print("Return value is:");
                System.out.println(s.foobar(4,3));
        }
}
```

Figure 2. A sample Java source file

```
1:  String [39]
2:  Integer [8]
3:  Class [42]
4:  Class [23]
5:  Class [35]
6:  Class [32]
7:  Methodref [6, 13]
8:  Methodref [5, 13]
9:  Methodref [4, 16]
10: Methodref [6, 17]
11: Fieldref [3, 14]
12: Methodref [4, 15]
13: NameAndType [40, 43]
14: NameAndType [33, 41]
15: NameAndType [20, 34]
16: NameAndType [18, 19]
17: NameAndType [36, 22]
18: Utf8 [7, "println"]
19: Utf8 [4, "(I)V"]
20: Utf8 [5, "print"]
21: Utf8 [13, "ConstantValue"]
22: Utf8 [5, "(II)I"]
23: Utf8 [19, "java/io/PrintStream"]
24: Utf8 [10, "Exceptions"]
25: Utf8 [15, "LineNumberTable"]
26: Utf8 [1, "I"]
27: Utf8 [10, "SourceFile"]
28: Utf8 [14, "LocalVariables"]
29: Utf8 [4, "Code"]
30: Utf8 [10, "Small.java"]
31: Utf8 [6, "CUTOFF"]
32: Utf8 [5, "Small"]
33: Utf8 [3, "out"]
34: Utf8 [21, "(Ljava/lang/String;)V"]
35: Utf8 [16, "java/lang/Object"]
36: Utf8 [6, "foobar"]
37: Utf8 [4, "main"]
38: Utf8 [22, "([Ljava/lang/String;)V"]
39: Utf8 [16, "Return value is:"]
40: Utf8 [6, "<init>"]
41: Utf8 [21, "Ljava/io/PrintStream;"]
42: Utf8 [16, "java/lang/System"]
43: Utf8 [3, "()V"]
```

Figure 3. Constant pool entries

The Methods Section

Each entry in the Methods Section is a variable-length structure that describes one method in the class. In addition to some information about the class (its name, access permissions, etc.), the entry normally contains both a *Code* attribute and an *Exceptions* attribute. The Code attribute contains an array of the bytecode that is to be interpretively executed by the JVM when this method is invoked. It also includes a table providing information about exception handlers in the code and, potentially, a *LineNumberTable* and a *LocalVariableTable* which would enable a debugger to relate the bytecode and the local variables of the method to its source code. The Exceptions attribute is normally small.

A COMPRESSED ‘WIRE’ REPRESENTATION FOR JAVA?

A recent paper[4] describes an approach that compresses intermediate code from the lcc C compiler to as little as 21 per cent of the corresponding executable code for the SPARC architecture. This is a compression rate of 4·9 to 1. At first sight, it would appear that a similar approach should be effective for Java. The Class file format is similar in nature to the intermediate code, or IR code, generated by the front-end of a conventional compiler.

The ‘wire’ format for C code is explained in Ref. 4. It is based on a heuristic tree pattern matching method for compressing IR code that is described in Ref. 10. It involves splitting the sequence of tree patterns into separate streams and then applying three different compression methods to the streams—Move-To-Front encoding, Huffman coding and gzip compression.

If we were to develop a similar approach for Java, we would need to re-engineer the Java compiler so that it generates trees instead of the linearized byte code and

where constants appear as leaves in those trees instead of having been separated out into a special Constants section. In principle, it should be possible though inefficient to construct such trees from the information in the Class file.

The reason why we consider such an approach to be unappealing however is provided by the compression results reported in Ref. 4. These results are reproduced in Table I. As can be seen, wire format achieves its reported 4·9 to 1 compression ratio only on the largest file. For the smallest of the three files reported in the paper, the compression ratio is worse than that produced by gzip. This file has an uncompressed size of 60 KB, which is already much larger than the average for Java Class files. Consequently, we conclude that a Java wire format would rarely achieve better compression than gzip. The tailored approach that we have developed invariably achieves better compression than gzip.

DEVELOPING A TAILORED SOLUTION FOR JAVA CLASS FILES

Our review of the structure of the class file should have brought out the importance of the constant pool section. First, every class file will almost necessarily include many 'standard' character strings containing names and type signatures for methods in the Java class libraries. These strings will have a major influence on small class files, and the class files in package libraries do tend to be small. Second, the other sections of the class file all contain indexes into the constant pool. The net result is that the constant pool occupies 50–90 per cent of the entire file size when measured on a test collection of about 1000 class files. A chart that shows the contributions of the constant pool to the total file size is shown in Figure 4. The chart should be read as follows: the height of the column centred around 0·7, for example, represents the number of class files in our collection where the constant pool occupied between 65 per cent and 75 per cent of the whole file, i.e. 380 of our 1000 class files had constant pools that represented between 65 per cent and 75 per cent of the whole file.

Another important observation is that character string constants dominate the constant pool. These are strings represented in the UTF8 format. Each string is represented by a two-byte length immediately followed by the bytes that comprise the string constant. Together, the string constants account for about 40–80 per cent of the entire file size. Figure 5 graphically shows the size contribution fo UTF8 strings to the overall class file size. It should be abundantly clear that a good compression scheme for Java class files would have to perform well on UTF8 constant strings.

After the constant pool, we found the next most significant component of the class file to be the code attribute. Figure 6 shows the contribution of bytecode to the overall class file size in our experimental measurements.

Table I. SPARC code compression reported by Ernst *et al*.

Executable file	Original size	Gzipped size	Wire code size
lcc	315,636	75,928	64,475
gcc	1,381,304	380,451	287,260
agrep	61,036	15,936	16,013

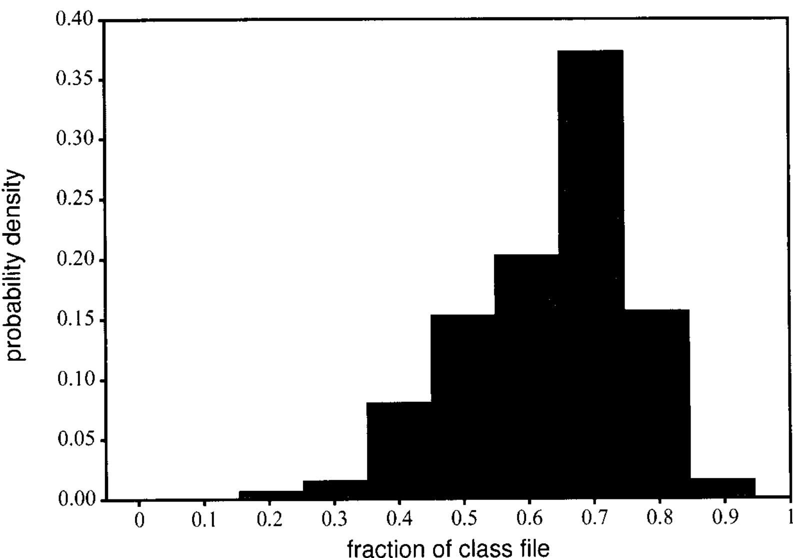


Figure 4. Distribution of constant pool contributions

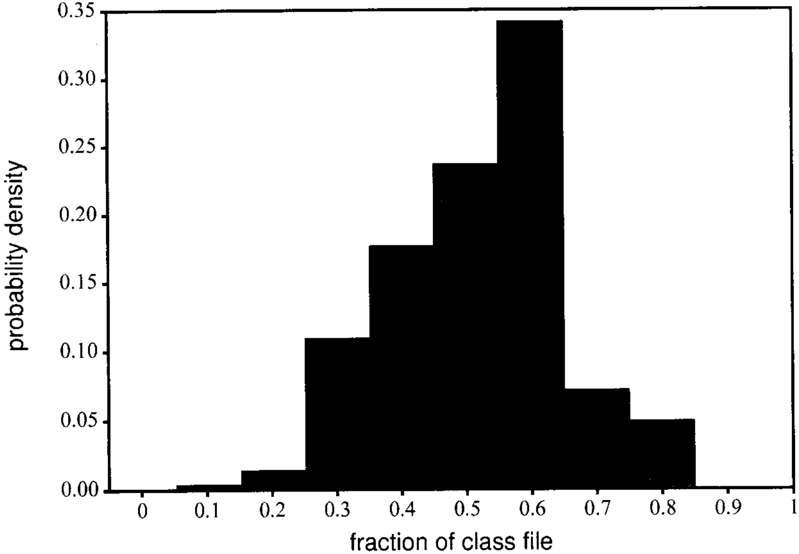


Figure 5. Distribution of UTF8 string constant contributions

The third most important component of the class file is the LineNumberTable Atribute. It is not necessarily present in a class file because it is needed only for debugging and error reporting. If the table is present, it holds one entry for every time the line number of the corresponding source code changes when making a sequential scan through the bytecode array. Its size should be roughly proportional to the bytecode array size.

The other components of the class file did not make significant contributions to

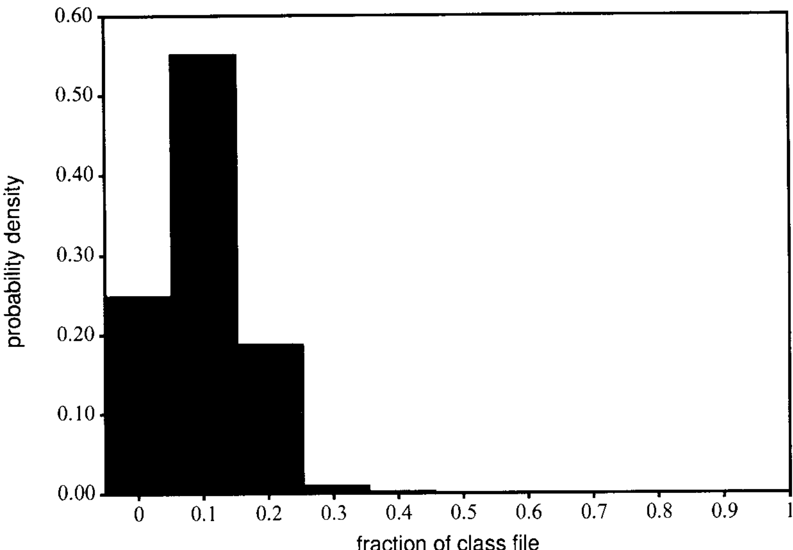


Figure 6. Contribution of Bytecode to class file size

overall file size in our experimental measurements. We therefore do not consider it necessary to provide tailored compression schemes for them.

A general compression program such as gzip is lossless in the sense that a decompressed file will be *identical* to the original file. A key observation is that a compression program for Java class files does not need to be perfectly identical to the original. It is good enough if the decompressed file *executes* in the same way as the original. That is, we only need to preserve semantic equivalence and not textual equivalence between the two files. For example, one of our biggest improvements to space efficiency comes from reordering the constant pool. As long as all indexes into the constant pool are adjusted to reflect the new order, the bytecode should execute in exactly the same way as before.

We note that re-ordering the constant pool may cause the bytecode component of the class file to *increase* in size. This is because the LDC (load constant) instruction has an operand which is a one byte index into the constant pool. If re-ordering should cause a constant used as an operand of LDC to move out of the first 256 positions in the constant pool, then the wide form of the instruction, LDC_W, must be used instead and that occupies more memory. However, the risk is worth taking. In our experiments, the effect was observed only rarely and caused only tiny increases in the size of the decompressed file.

We now explain our transformations on the significant parts of the class file.

Constant Pool Entries

Our initial transformation is to reorder the entries of the constant pool so that all entries of the same type are grouped together and so that UTF8 strings are sorted by their lengths. The result of this reordering on the example constant pool of Figure 3 is shown in Figure 7.

There are three important benefits from the reordering. First, we no longer need

1:	NameAndType [29, 20]	**23:**	Utf8 [4, "main"]
2:	NameAndType [19, 42]	**24:**	Utf8 [5, "print"]
3:	NameAndType [24, 41]	**25:**	Utf8 [5, "(II)I"]
4:	NameAndType [30, 21]	**26:**	Utf8 [5, "Small"]
5:	NameAndType [28, 25]	**27:**	Utf8 [6, "CUTOFF"]
6:	Methodref [16, 1]	**28:**	Utf8 [6, "foobar"]
7:	Methodref [15, 1]	**29:**	Utf8 [6, "<init>"]
8:	Methodref [14, 4]	**30:**	Utf8 [7, "println"]
9:	Methodref [16, 5]	**31:**	Utf8 [10, "Small.java"]
10:	Methodref [14, 3]	**32:**	Utf8 [10, "Exceptions"]
11:	Fieldref [13, 2]	**33:**	Utf8 [10, "SourceFile"]
12:	String [38]	**34:**	Utf8 [13, "ConstantValue"]
13:	Class [39]	**35:**	Utf8 [14, "LocalVariables"]
14:	Class [40]	**36:**	Utf8 [15, "LineNumberTable"]
15:	Class [37]	**37:**	Utf8 [16, "java/lang/Object"]
16:	Class [26]	**38:**	Utf8 [16, "Return value is:"]
17:	Integer [8]	**39:**	Utf8 [16, "java/lang/System"]
18:	Utf8 [1, "I"]	**40:**	Utf8 [19, "java/io/PrintStream"]
19:	Utf8 [3, "out"]	**41:**	Utf8 [21, "(Ljava/lang/String;)V"]
20:	Utf8 [3, "()V"]	**42:**	Utf8 [21, "Ljava/io/PrintStream;"]
21:	Utf8 [4, "(I)V"]	**43:**	Utf8 [22, "([Ljava/lang/String;)V"]
22:	Utf8 [4, "Code"]		

Figure 7. Reordered constant pool entries

to associate a tag byte with each entry when outputting the constant pool in its compressed form. The decoding program needs to know only how many entries there are of each type in order to reconstruct a constant pool that contains the proper tag bytes. Thus, the compression program outputs a simple count of how many entries there are of each type before outputting the entries of that type. We used a start-step-stop code with parameters (1, 3, 16) to encode the count. (Start-step-stop codes are variable-length codes where small integers are encoded in a few bits while larger integers require more bits for their encoding. The scheme is explained in more detail below.) Since there are so few counts to encode in each class file, the choice of the particular encoding scheme is not critical.

The second benefit comes from encoding constant pool entries that contain references to other constant pool entries. For example, an entry of type Fieldref is normally coded as a tag byte followed by two 16-bit indexes. The first index always references a constant pool entry of type Class and the second always references an entry of type NameAndType. Since our reordered constant pool has grouped all the Class and NameAndType entries together, we can replace the first index with a number that represents the position within the group of Class entries, and similarly for the second index. These relative indexes will almost always have much smaller values than the original index numbers and we can therefore encode them using fewer bits. We encoded each index into the group of Class entries using a fixed-length binary code with $\lceil \log N_{\mathrm{Class}} \rceil$ bits, where N_{class} is the number of entries of type Class in the constant pool.

The third benefit is that reordering the UTF8 strings into order of increasing length means that we can encode the string lengths in a more efficient manner. If we look at an arbitrary string constant, other than the first, in Figure 7, we can see that its length is almost always identical to the length of the preceding entry or is

only very slightly longer. In other words, if we encode the length of a string as the difference between its length and that of the preceding string, we will be encoding much smaller numbers. Encoding differences rather than the values themselves is known as *delta coding*. Since a typical class file contains relatively many string constants, it is worthwhile to devise a scheme for encoding the deltas (differences) as efficiently as possible. To that end, we determined the distribution of string length deltas for our collection of class files. We then determined which start-step-stop code matched that distribution best. The result is shown in Figure 8. The solid line shows the distribution of delta values; the dashed line shows what the distribution should be to perfectly match the (0, 1, 16) start-step-stop code that we picked as being the closest match.

After extracting and encoding the length prefixes, the group of string constants becomes a block of text that contains a substantial amount of repetition. (Observe, for example, the repetition of the substring 'java' in Figure 7.) This block of text is well-suited for standard text compression algorithms. For convenience, we used the ZLIB library functions[11,12] to compress the text. ZLIB implements the same compression algorithm as used in gzip, a method that is very similar to the *deflate* compression method supported in the zip file format.[9] The maximum compression option for ZLIB/deflate was used here, as in all our uses of this compression method.

Other kinds of entries in the constant pool, such as integer or floating-point constants, occurred so infrequently in our sample files that there was very little benefit from devising special coding schemes for them. We therefore left their representations unchanged.

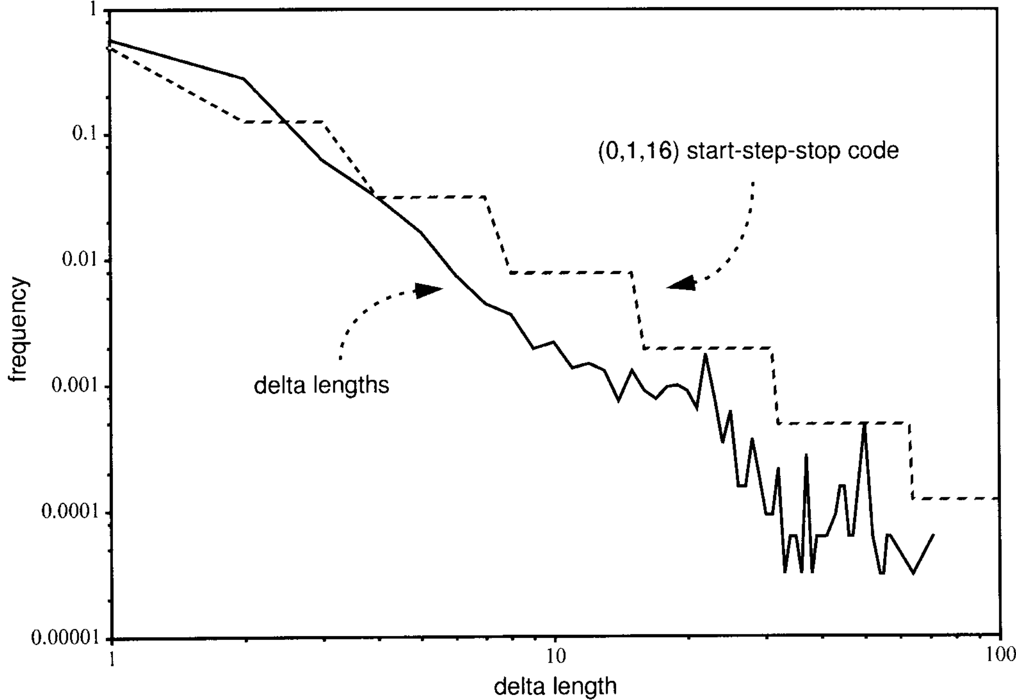


Figure 8. Distribution of delta string lengths

Code Attribute

The bytecode part of the class file contains the patterns of JVM instructions generated by the Java compiler for the constructs in the source program. Unless the compiler is sophisticated and optimizes these patterns extensively, there will necessarily be repetitions of the coding patterns. However, we were unable to find a fast and effective way to exploit these patterns. It would be easier and preferable for the compiler to generate such patterns in a compact form directly, rather than having to rediscover the patterns by analyzing the bytecode. Such is the approach of the *Slim Binaries* format of Kistler and Franz.[13]

In keeping with our desire to preserve the existing bytecode format, we chose to perform only two simple transformations on the bytecode and then apply the ZLIB compression algorithm[11,12] to it. The first transformation was to separate the opcodes and the operands into two separate arrays. By separating out the opcodes, any repeated patterns of opcodes will become apparent and amenable to compression by a general-purpose method.

The second transformation concerned operands of branching instructions. The operands of branching instructions are the addresses of other instructions in the bytecode array for the method being executed. They are normally implemented as 2-byte offsets. For example, if index positions 103–105 of the array hold the ifnonnull branching instruction, and its target is an instruction at index position 124, then bytes 103 and 104 will hold the value 21 (computed as 124–103). Such a representation is redundant because not every position in the bytecode array represents the start of an instruction—many JVM instructions occupy two or more bytes. We eliminated the redundancy by replacing byte-offsets with instruction-offsets in our compressed file format.

Following the re-encoding of all instruction-relative offsets, we separately compress the two arrays created from the bytecode using the ZLIB routines.

LineNumberTable Attribute

Each entry in the LineNumberTable contains a code array index and a corresponding source statement number. The indexes can only refer to the starts of JVM instructions. Therefore, space can be saved by converting these indexes into instruction numbers. Further compression is achieved by using delta coding. Both the instruction numbers and the statement numbers form slowly increasing sequences in our collection of sample class files. Presumably a sophisticated Java compiler could re-order code and thus break the property that statement numbers only increase through the code array; however, statements would be likely to be moved in groups and delta coding would still achieve good results. We used (2, 2, 16) start-step-stop codes for both the instruction number differences and statement number differences.

Start-Step-Stop Codes

We make extensive use of start-step-stop codes[6] to encode various kinds of integers in our compressed class files. Such codes have the general property that small integers receive shorter codes than large integers. The codes are generated in a systematic manner that permits rapid conversions between an integer and its encoded representation.

The codes have three parameters which control the range of integers that can be represented and the rate at which the bit string encoding grows in length. The encoding would be optimal if the range exactly matches the range of numbers that we need to represent and if the number of bits used to encode an integer k is logarithmically related to the frequency with which k needs to be encoded. That is, if $\mathrm{len}(k)$ is the number of bits used to encode k, and if Freq_k is the frequency of occurrences of k, then we would desire that

$$\mathrm{len}(k) = -\log(\mathrm{Freq}_k)$$

should hold. In practice, we can only choose a start-step-stop code that approximately matches the frequency distribution. Huffman coding[6] will usually produce better compression, but the compression and decompression algorithms are more complicated and require that a coding table be provided.

The underlying number representation used by a start-step-stop code is the usual binary. However, the encoder and decoder must agree on how many bits comprise the binary number. Rather than using a fixed, predetermined, number of bits, the start-step-stop code prefixes the binary number with a code that specifies the number of bits in the binary part. This prefix code is implemented as a unary number; unary being a scheme that can be decoded without knowing the number of bits in advance. For example, the unary code for the integer 5 is 111110, constructed as five 1-bits and terminated by a 0-bit. If the unary number is m, then the number of bits in the immediately following binary part of the code is $a + b \times m$ where a is the start parameter and b is the stop parameter. The stop parameter c is the maximum value that the unary prefix is allowed to encode. Knowledge of this value is used to optimize the way in which the unary code is written (its final 0 bit can be safely dropped). As an example, the table of start-step-stop codes for $(1, 2, 5)$ coding is shown in Table II. To make the codes easier to interpret, the prefix part of each code is underlined.

EXPERIMENTAL RESULTS

A C implementation of our tailored compression approach was programmed. We named this program *clazz*. Compression results for some representative class files

Table II. (1,2,5) Start-Step-Stop codes

Integer	Code	Integer	Code
0	0 0	8	10 110
1	0 1	9	10 111
2	10 000	10	11 00000
3	10 001	11	11 00001
4	10 010	12	11 00010
5	10 011	…	…
6	10 100	40	11 11110
7	10 101	41	11 11111

Table III. Compression results for representative class files

File	Original size	ZLIB deflated size	bzip2 size	clazz size
AudioClip.class	233	184	225	95
Component.class	24,622	11,154	11,269	10,050
Enumeration.class	261	203	254	107
HashTableEntry.class	630	404	458	229
Integer.class	3,733	1,919	2,113	1,610
Object.class	1,452	787	923	576

are shown in Table III. In this table, we compare the compression of our clazz program against two general-purpose text compression programs—the *deflate* method of the ZLIB library (with maximum compression selected as an option) and bzip2. The ZLIB program is relatively fast and could therefore be considered as a good candidate for compressing Java class files. In this table, we show the results for some of the largest files as well as for some of the smallest files. We can observe that ZLIB and bzip2 perform better on large class files but quite poorly on small files, where they have little opportunity to adapt to the file characteristics. Our clazz program, on the other hand, achieves significant compression for all file sizes and always outperforms both competitors. Its better performance with small file sizes is marked.

Compression results for two collections of class files are shown in Table IV. Both class file collections were taken from the Metrowerks Codewarrior distribution. The first 50 classfile members of the Swing/Rose library and the first 128 members of the standard Java class library were used. The files were compressed separately. Again, clazz outperformed the ZLIB deflate method and bzip2 by a significant margin. Expressed as compression ratios, clazz is achieving a reduction to 35–38 per cent of the original size, versus 46–51 per cent for ZLIB and 51–56 per cent for bzip2.

Since the clazz program applies a variety of compression methods to different components of the class file, it is interesting to observe how well each component is compressed. We observed the following:

- Tag bytes attached to entries in the constant pool accounted for 2–6 per cent of the size of our sample class files. Our reordering of the entries and replacing the tags with counts, using start-step-stop codes, reduced the contribution of tags to insignificance.

Table IV. Compression results for collections of class files

Library	Average Sizes (in bytes)			
	Original file	ZLIB/deflate	bzip2	clazz
50 members of Rose class library	4047·2	1881·8	2063·6	1431·5
128 members of MW class library	2405·5	1221·9	1354·7	920·6

- The length fields of UTF8 strings were reduced from 2 bytes to an average of 2·7 bits, i.e. to 17 per cent of their original size.
- The entire constant pool was reduced, on average, to 31 per cent of its original size, even though we made no attempt to compress entries for integer constants or floating-point constants.
- A simpler method to compress the constant pool would be to reorder the entries and remove the superfluous tag bytes, as explained above, and then apply the ZLIB compression routine. This achieves somewhat worse compression than that produced by our more complicated approach. For example, the file Integer.class which is compressed to 1610 bytes with our method would be compressed to 1761 bytes instead. We consider this difference to be worth the price of the more complicated method.
- Our attempts to compress the bytecode arrays were successful only for larger class files. In many cases, methods contained fewer than 20 bytes of bytecode. On average, each method had its bytecode reduced to 59 per cent of its original size. The best compression, observed for those methods with the most bytecode, reduced the bytecode to 26 per cent of its original size.
- The LineNumberTable attribute, when present in the class file, was compressed, on average, to 33 per cent of its original size. (Production code would not normally contain this attribute.)
- Reordering the constant pool and making corresponding changes throughout other sections of the class file indeed has no effect when executed by the JVM. Spot checks with several files yielded no discernible difference in behaviour at execution time.

Execution times for compressing and decompressing representative class files are shown in Table V. All times are measured in seconds and were obtained with a 120 MHz Intel Pentium CPU. Compression times are quite competitive with the

Table V. Execution times for compression and decompression

File	Size (bytes)	Execution times (in seconds)		
		ZLIB/deflate	bzip2	clazz
AudioClip.class	233	0·049 0·019	0·577 0·385	0·049 0·025
Component.class	24,622	0·368 0·088	1·340 0·577	0·338 0·370
Enumeration.class	261	0·052 0·024	0·538 0·373	0·052 0·025
HashTableEntry.class	630	0·053 0·026	0·563 0·376	0·058 0·030
Integer.class	3,733	0·105 0·032	0·747 0·417	0·103 0·070
Object.class	1,452	0·057 0·026	0·598 0·381	0·067 0·037

ZLIB library, while decompression times are only a little worse. Our timings could undoubtedly be further improved with a more careful implementation. We observe too that the decompression time could be greatly reduced by integrating decompression with the Java class loader. One reason is that our compressed format has eliminated the need for one step of the bytecode verification process that is performed before the bytecode is executed. The verifier must check that every branch address, every entry point and every exception handler begins at the start of a bytecode instruction. Our compressed file format guarantees that this property must hold. A second reason is that the decompression program re-constructs the class file as an organized collection of data structures in memory as an intermediate step. This is work that the class loader would also perform.

CONCLUSIONS

The class file compression strategy, implemented as the clazz program, achieves much better compression than general-purpose compression programs while retaining full compatibility with the JVM architecture. A key insight is that the reconstructed file does not need to be identical to the original—it need be only semantically equivalent. Our implementation is not as fast as the competing compression programs, but that issue could be alleviated or eliminated if we were to re-implement the program more carefully and if we could combine the decompression code with the Java class loader.

A longer term and more drastic way of achieving greater compression would involve a complete re-design of the class file structure of the JVM instruction set. The slim binaries proposal,[13] for example, provides a very compact alternative format for bytecode along with the constants used in that code. Yet another possibility would be to design a new JAR file format where members of the archive share a common string constants table. Class files belonging to the same package typically duplicate many string constants, representing member names and method signatures.

ACKNOWLEDGEMENTS

Financial support from Natural Sciences and Engineering Research Council of Canada, in the form of a scholarship for the second author and a research grant for the first author, is gratefully acknowledged. Comments provided by the reviewers were invaluable in improving the experimental results.

REFERENCES

1. K. Arnold and J. Gosling, *The Java Programming Language,* Addison-Wesley, 1997.
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
3. M. Nelson and J.-L. Gailly, *The Data Compression Book, 2nd Edition*. M & T Books, 1995.
4. J. Ernst, W. Evans, C. W. Fraser, S. Lucco and T. A. Proebsting, 'Code compression', *Proceedings of PLDI'97, ACM Conference on Programming Languages, Design and Implementation*, 1997, pp. 358–365.
5. T. L. Yu, 'Data compression for PC software distribution', *Software—Practice and Experience*, **26**(11), 1181–1195 (1996).
6. T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression*, Prentice-Hall, 1990.
7. J. Seward, 'The Bzip2 home page', URL: http://www.muraroa.demon.co.uk and mirrored at http://www.digistar.com/bzip2 in North America (1998).
8. J. Corless, 'Compression of Java Class Files', *MSc Thesis*, Department of Computer Science, University of Victoria, 1997.
9. Info-ZIP 'General format of a ZIP file', Info-ZIP note 970311, URL: http://www.cdrom.com/pub/infozip/doc/ (1997).

10. C. W. Fraser and T. A. Proebsting, 'Custom instruction sets for code compression', URL: http://www.cs.arizona.edu/people/todd/papers/pldi2.ps (1995).
11. L. P. Deutsch and J.-L. Gailly, 'ZLIB compressed data format specification, version 3.3', URL: http://quest.jpl.nasa.gov/zlib/rfc-zlib.html (1996).
12. The Zlib home page, URL: http://www.cdrom.com/pub/infozip/zlib/, 1998.
13. T. Kistler and M. Franz, 'A tree-based alternative to Java byte-codes', *Technical Report 96-58*, Department of Information and Computing Science, University of California at Irvine, 1996.

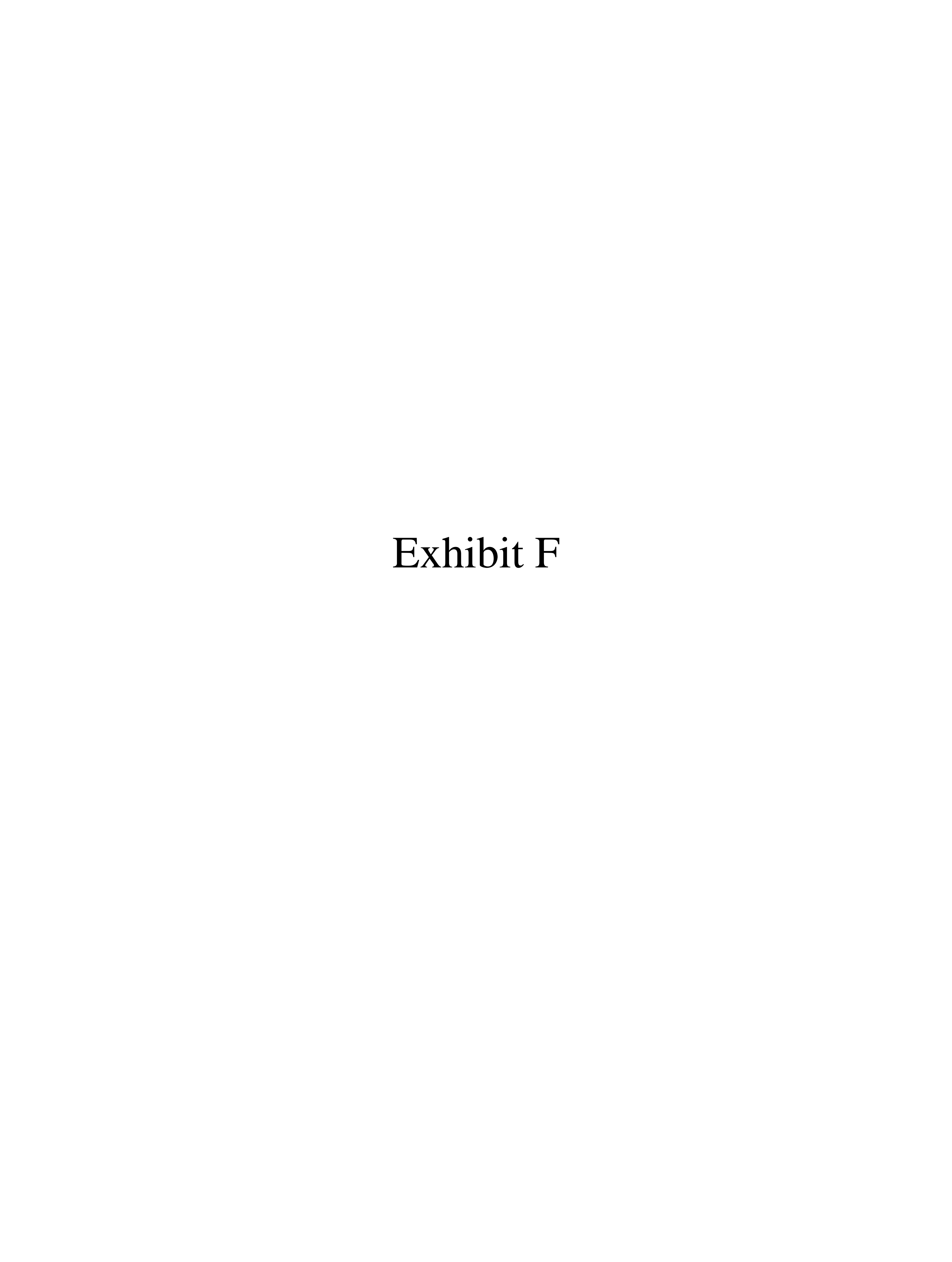


Exhibit F

PARTIAL PROSECUTION HISTORY SUMMARY

U.S. Patent No. RE38,104 (“the ‘104 patent”) is a continuation of reissue patent application Ser. No. 08/755,764 (now U.S. Patent No. RE36,204, “the ‘204 patent”), which was a reissue of U.S. Patent No. 5,367,685 (“the ‘685 patent”). ‘104 patent, first page, 1:9–11. This summary focuses on the use of the term “computer-readable medium” and is arranged chronologically starting with the application that issued as the ‘685 patent.

‘685 Patent

Dec. 22, 1992: The original patent application was filed listing ten claims directed to, a) “a method for generating executable code,” and b) “an apparatus for generating executable code.” ‘104 patent, first page, 1:9–11, Ex. B-2, OAGOOGLE0000057197–99. None of these claims included the term “computer-readable medium.” *Id.*

May 26, 1994: An Examiner interview was conducted with a summary submitted by the Examiner stating: “changes made as per Examiner Amendment.” Ex. B-2, OAGOOGLE0000057220.

May 26, 1994: The Examiner issued a Notice of Allowability along with an Examiners Amendment adding an “s” to the word “instruction” in independent claim 1 and deleting the last two lines of dependent claim 10. Ex. B-2,OAGOOGLE0000057222.

Nov. 22, 1994: The ‘685 patent was issued by the USPTO. ‘104 Patent, first page.

‘204 Patent

Nov. 21, 1996: An application was filed seeking a broadening reissue of the ‘685 patent. ‘204 patent, first page. The reissue application contained new claims, including claims 29–34, which recited the term “computer-readable medium.” Ex. B-2, OAGOOGLE0000059223.

Sept. 24, 1997: The Examiner issued an Office Action rejecting all claims of the reissue application because “[t]he reissue oath or declaration filed with this application is defective because it fails to particularly specify the errors and/or how the errors relied up on arose or

occurred as required under 37 CFR 1.175(a)(5)." Ex. B-2, OAGOOGLE0000059222–24.

Apr. 27, 1999: The '685 patent was reissued as the '204 patent without amendment to claims 29–34 added on Nov. 21, 1996.

'104 Patent

Mar. 3, 1999: A continuation application was filed based on the still pending reissue application (which issued as the '204 patent). '104 patent, first page. A preliminary amendment was filed with the continuing reissue application. The preliminary amendment cancelled all existing claims and added new claims 36–43, of which claim 36 recited the term "computer-readable medium." Ex. B-3, OAGOOGLE0000059421–25.

Apr. 29, 2003: The '685 patent was again reissued as the '104 patent, including claim 12, which recites the term "computer-readable medium." '104 patent, first page, 7:15–28.