



(51) International Patent Classification:
G06F 9/45 (2006.01)

(21) International Application Number:
PCT/US2011/044895

(22) International Filing Date:
21 July 2011 (21.07.2011)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
12/847,925 30 July 2010 (30.07.2010) US

(71) Applicant (for all designated States except US): **APPLE INC.** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **HERNANDEZ PORRAS, Victor, Leonel** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **LATTNER, Christopher, Arthur** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **CHEN, Jia-hong** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **CHRISTOPHER, Eric, Marshall**

[US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **HOOVER, Roger, Scott** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **JOUAUX, Francois** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **MCCALL, Robert, John** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **O'BRIEN, Thomas, John** [US/US]; 1 Infinite Loop, Cupertino, CA 95014 (US). **SOLANKI, Pratik** [IN/US]; 1 Infinite Loop, Cupertino, CA 95014 (US).

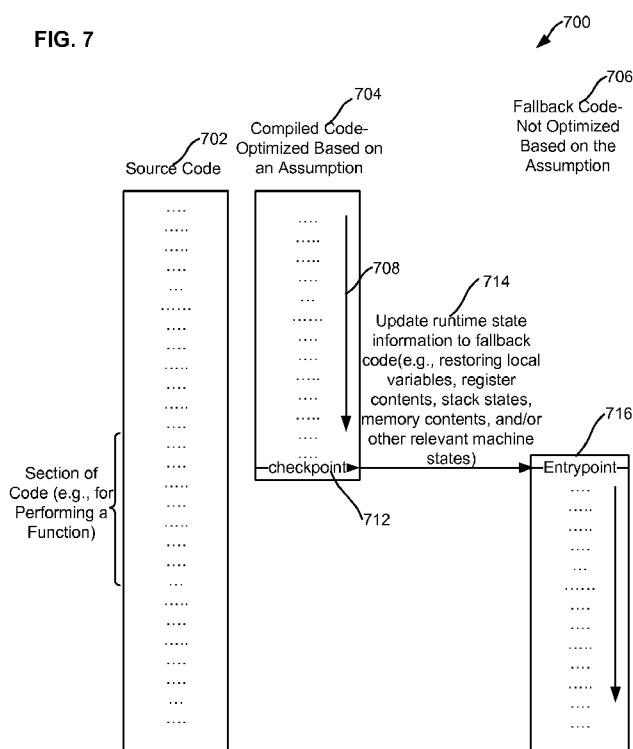
(74) Agent: **CHEN, Linna, R.**; Van Pelt, Yi & James LLP, 10050 N. Foothill Blvd., Suite 200, Cupertino, CA 95014 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

[Continued on next page]

(54) Title: DYNAMIC COMPILER OPTIMISATION

FIG. 7



(57) Abstract: Techniques for processing source code written in a traditionally interpreted language such as JavaScript, or another dynamic and/or interpreted language, are disclosed. In one example, compiled code associated with the source code is constructed and executed. An assumption on which a specific aspect of the compiled code is based (e.g., an optimization) is tested at a checkpoint of the compiled code. A roll over to fallback code is performed if the test indicates the assumption is not true.



(84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK,

SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

DYNAMIC COMPILER OPTIMISATION

BACKGROUND OF THE INVENTION

[0001] Traditionally, computer code written in an interpreted language such as JavaScript is not executed directly by a computer processor, and instead must be interpreted at runtime by an interpreter, runtime environment, and/or other software entity. An example of such a typical configuration is shown in Figure 1, specifically JavaScript 102 is executed at runtime by an interpreter 104 which interprets the JavaScript instructions and provides corresponding machine code instructions to central processing unit (CPU) 106 for execution. An interpreted language such as JavaScript has the advantage of being intuitive, flexible, and easy to program, but such languages may be slow to execute since the code needs to be interpreted line by line at runtime before it can be executed.

[0002] By contrast, in traditional compiled programming languages, source code typically is compiled into machine code prior to runtime, e.g., source code 116 shown being compiled into compiled code 118 at a time earlier or separate from runtime in Figure 1. Machine code comprises very low level instructions, which are typically stored in data storage and subsequently executed directly by a processor, e.g., compiled code 120 shown being provided directly to CPU 124 in Figure 1, with the result that source code that has been compiled to machine code typically executes on a processor more quickly or efficiently than equivalent interpreted code (e.g., JavaScript), which requires an interpreter at runtime to execute the equivalent machine code on the processor.

[0003] More recently, scripts and other code written in interpreted languages such as JavaScript have been precompiled to machine code executable directly by a processor, e.g., pre-compiled JavaScript (i.e., machine code generated prior to runtime based on JavaScript) 140 shown in Figure 1 as being provided to CPU 144 for execution at runtime. However, the gains achieved by compiling code written in an interpreted dynamic language such as JavaScript prior to runtime in the past have been limited by the highly flexible, dynamic, and sometimes unpredictable nature and behavior of code written in such languages. For example, in JavaScript the variable type of a variable may be dynamically assigned or changed at runtime and, as a result, certain optimizations that could otherwise be performed

on a pre-compiled version of such code, for example type-specific optimizations, cannot be performed, resulting in less optimized compiled code.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] Various embodiments of the invention are disclosed in the following detailed description and the accompanying drawings.

[0005] Examples of such typical configurations are shown in Figure 1, specifically JavaScript 102 being executed at runtime by an interpreter 104 which interprets the JavaScript instructions and provides corresponding machine code instructions to a central processing unit 106, source code 116 being compiled into compiled machine code 118 prior to runtime, compiled machine code 120 being provided to CPU 124 for execution at runtime, and pre-compiled JavaScript being provided to CPU 144 for execution at runtime.

[0006] Figure 2 is a flow diagram illustrating an embodiment of a process for processing source code written in a traditionally interpreted language. Figure 3 is a flow diagram illustrating an embodiment of a process for processing source code written in a traditionally interpreted language such as JavaScript.

[0007] Figure 4 is a flow diagram of an embodiment of a process for generating compiled code and fallback code, where the compiled code is based on an assumption and the fallback code is not.

[0008] Figure 5 is a flow diagram of an embodiment of a process for generating compiled code and fallback code, where the compiled code is based on an assumption and the fallback code is not.

[0009] Figure 6 is a flow diagram illustrating an embodiment of a process for optimizing compiled code based on an assumption.

[0010] Figure 7 is flow diagram illustrating an embodiment of a process for rolling over from compiled code that is optimized based on an assumption to fallback code that is not optimized based on the assumption.

[0011] Figure 8 is a block diagram illustrating an embodiment of a system for processing source code written in a traditionally interpreted language such as JavaScript.

[0012] Figure 9 is a block diagram showing an embodiment of a system for processing source code written in a traditionally interpreted language such as JavaScript.

[0013] Figure 10 is a block diagram illustrating an embodiment of a system for processing source code written in a traditionally interpreted language such as JavaScript.

DETAILED DESCRIPTION

[0014] The invention can be implemented in numerous ways, including as a process; an apparatus; a system; a composition of matter; a computer program product embodied on a computer readable storage medium; and/or a processor, such as a processor configured to execute instructions stored on and/or provided by a memory coupled to the processor. In this specification, these implementations, or any other form that the invention may take, may be referred to as techniques. In general, the order of the steps of disclosed processes may be altered within the scope of the invention. Unless stated otherwise, a component such as a processor or a memory described as being configured to perform a task may be implemented as a general component that is temporarily configured to perform the task at a given time or a specific component that is manufactured to perform the task. As used herein, the term 'processor' refers to one or more devices, circuits, and/or processing cores configured to process data, such as computer program instructions.

[0015] A detailed description of one or more embodiments of the invention is provided below along with accompanying figures that illustrate the principles of the invention. The invention is described in connection with such embodiments, but the invention is not limited to any embodiment. The scope of the invention is limited only by the claims and the invention encompasses numerous alternatives, modifications and equivalents. Numerous specific details are set forth in the following description in order to provide a thorough understanding of the invention. These details are provided for the purpose of example and the invention may be practiced according to the claims without some or all of these specific details. For the purpose of clarity, technical material that is known in the technical fields related to the invention has not been described in detail so that the invention is not unnecessarily obscured.

[0016] Figure 2 is a flow diagram illustrating an embodiment of a process 200 for executing software code. In the example shown, a unit of assumption-based compiled code

associated with a unit of source code is executed 202. An assumption on which a specific aspect of the assumption-based compiled code is based (e.g., an optimization) is tested at a checkpoint of the compiled code 204. A roll over or transition to fallback code is performed and the fallback code is executed if the test indicates the assumption is not true 206.

[0017] In some embodiments, the assumption-based compiled code is low level machine readable code compiled from the source code and is specific to the platform that it is compiled for. The assumption-based compiled code is optimized in some embodiments based on an assumption while the fallback code is not optimized based on this assumption. In some embodiments, the assumption is one of a plurality of assumptions based on which the assumption-based compiled code is optimized. In some embodiments, the assumption is an assumption regarding a variable of the source code. In some embodiments, the assumption is an optimistic assumption of a likely form of the input variable, such as a likely value, type, and/or range of the variable.

[0018] In some embodiments, the assumption-based compiled code includes instructions for testing the assumption at the checkpoint, for invoking the fallback code when the assumption is tested not true, and/or for rolling over to the fallback code (e.g., updating the runtime state of the fallback code so that the fallback code can continue correctly from the point where the assumption-based compiled code is rolled over to the fallback code). In some embodiments, the checkpoint is one of a plurality of checkpoints throughout the assumption-based compiled code for testing the assumption and the fallback code is executed when the assumption is tested not true at any one of the checkpoints. In some embodiments, each of the plurality of checkpoints corresponds to a separate unit of fallback code customized to that particular checkpoint. Code execution can continue correctly in the fallback code (from a point corresponding to the particular checkpoint's place in the source code without re-starting the routine) when the assumption is tested to be not true at the particular checkpoint.

[0019] In one example, a variable in a unit of source code can be dynamically assigned or changed at runtime to be an integer, float, or string, however based on analysis (e.g., based on dynamic analysis of past runs and/or static analysis of the code), it is determined that the variable may likely be an integer. Consequently, the assumption-based compiled code is optimized by assuming that the variable will be an integer so that the compiled code is simplified and made more efficient in situations where the assumption holds

true. However, since the assumption-based compiled code is based on the assumption that the variable is an integer, it may not run correctly and/or efficiently in situations where this assumption does not hold true. In contrast, the fallback code is not based on the assumption and will run correctly when the variable is not an integer, although the fallback code may be less efficient than the assumption-based compiled code when the variable is an integer. The fallback code can be used as a backup for the optimized assumption-based compiled code when the assumption does not hold true.

[0020] Figure 3 is a flow diagram illustrating an embodiment of a process 300 for executing software code. In the example shown, a unit of assumption-based compiled code associated with a unit of source code and fallback code associated with the source code are received and/or generated locally on a local computing environment 302. The assumption-based compiled code in some embodiments comprises low level machine readable code that is optimized based on an assumption, compiled based on the source code, and is specific to the platform that it is compiled for. In contrast, the fallback code is not optimized based on the assumption. In some embodiments the fallback code is compiled machine code specific to the platform it is compiled for, but is not optimized or otherwise based on the assumption. In some embodiments the fallback code is the original source code which is then interpreted.

[0021] The assumption-based compiled form of the source code is executed when the source code is invoked 304. For example when a user browses a webpage and invokes the JavaScript behind the webpage, a compiled version of the JavaScript that is optimized based on an assumption is executed. As the assumption-based compiled code is executed, a checkpoint for testing the assumption is encountered and the assumption is tested at the checkpoint 306. For example, the assumption-based compiled code in some embodiments includes checkpoint code configured to test the validity of the assumption, for example at a point in the execution of the assumption-based code that occurs prior to a portion of the code that has been optimized or otherwise based on the assumption. If the assumption is tested to be true at the checkpoint 306, the assumption-based compiled code continues to be executed unless/until the above or in some embodiments another assumption proves false 306 or until execution is done 310. If the assumption is tested to be not true 306, the execution of the assumption-based compiled code is rolled over to the fallback code and the fallback code is executed instead 308. In some embodiments, rollover involves updating the runtime environment of the fallback code so that the fallback code can continue execution correctly

from where the assumption-based compiled code stopped, and the system state is as if it has always been executing the fallback code alone, and the assumption-based compiled code had never been executed. In some embodiments, rollover involves rolling back the runtime environment to a saved state of the assumption-based compiled code and then updating the runtime environment of the fallback code with the saved state. In some embodiments, updating the runtime environment of the fallback code involves restoring the local variables and other relevant machine state, which can include but is not limited to register contents, stack state, and memory contents, while leaving the global variables alone. The checkpoint can be located at any appropriate point in the assumption-based compiled code. For example, if a section of the assumption-based compiled code is optimized based on an assumption, checkpoint(s) for the assumption can be placed before or within the section of code, so that the assumption can be tested before the execution of this section of code and/or during the execution of this section of code. Detection and collapsing of redundant checkpoints can be used to further optimize the assumption-based compiled code.

[0022] Figure 4 is a flow diagram of an embodiment of a process 400 for generating assumption-based compiled code and fallback code associated with a unit of source code, such as source code written in a traditionally interpreted language such as JavaScript. In the example shown, source code is received (e.g., downloaded from or uploaded by another device) 402. An initial compiled version of the source code (e.g., compiled machine code specific to a platform and can be directly read by a processor) is generated 404. Analysis, such as static analysis of the source code and/or initial compiled code, and/or dynamic runtime analysis of the source code and/or initial compiled code, is performed to determine whether an assumption can be made that would allow an optimization of the initial compiled version of the source code (e.g., a section of the initial compiled version of the source code) 406. If an assumption based on which the initial compiled version of the source code can be optimized is found, the initial (or other current) compiled version of the source code is optimized based on the assumption to generate an optimized (or further optimized) compiled version of the source code 408. The optimized (or further optimized) compiled version of the source code includes in some embodiments one or more checkpoints for testing the validity of the assumption and for rolling over to fallback code if the assumption is tested not true. The process determines whether further optimization of the compiled version of the source code based on additional assumptions will be performed 406, 410. If yes, the steps 406, 408, and 410 are repeated to perform such further optimization. Analysis of the source and/or

initially generated compiled code continues, and opportunities to optimize based on a potentially valid assumption (if any) are taken advantage of, until the entire code has been analyzed and no further opportunities for optimization based on an assumption are found 406, 410, at which time the process of Figure 4 ends.

[0023] Figure 5 is a flow diagram for an embodiment of a process 500 for generating assumption-based compiled code and fallback code associated with a unit of source code. In the example shown, source code 502 is used to generate an initial intermediate representation 504 of the source code that is not optimized based on an assumption. The intermediate representation 504 in the example shown goes through successive rounds of optimization to generate subsequent intermediate representations represented in Figure 5 by intermediate representations 506 and 508, the latter of which in this example is used ultimately to generate final assumption-based compiled code 512 that is optimized based on one or more assumptions. In some embodiments, the source code comprises JavaScript and at least a subset of the intermediate representations shown in Figure 5 comprise LLVM or other byte code, or another compiled or otherwise derived representation of the original source code. In the example shown, the final intermediate representation is used to generate assumption-based compiled code 512 that is optimized based on one or more assumptions. The initial intermediate representation 504 is used in this example to generate a compiled code that used as fallback code 510 that is not based on an assumption, e.g., one based on which an optimization of the initial intermediate representation 504 is performed to generate a subsequent, optimized intermediate representation. In some embodiments, for each intermediate representation that is optimized based on an assumption, corresponding fallback code is generated based on an intermediate version (e.g., 1 through n in this example) that is not based on that assumption, e.g., an intermediate version as it existed immediately prior to an optimization based on that assumption being performed. In some cases further or subsequent iterations of optimization may be facilitated by an assumption but may not themselves be based on any new assumption, so that not every intermediate representation would have or would require corresponding fallback code. In some embodiments, one or more assumptions may be made without new compiled fallback code being generated, and instead for example the original source code may be used as a fallback code if that particular assumption is found not to be true at runtime. While in the example shown in Figure 5 multiple iterations of intermediate representation of the source code are generated, in other embodiments compilation may involve more or fewer stages than shown in Figure 5.

[0024] Figure 6 is a flow diagram illustrating an embodiment of a process 600 for optimizing compiled code based on an assumption (e.g., the step 408 of Figure 4). In the example shown, the compiled code associated with the source code is optimized based on an assumption 602. Code for testing the assumption is true is added to the compiled code 604. Code for rolling over to fallback code, if the assumption is found at runtime not to be true, is added to the compiled code 606. In various embodiments, the process 600 is repeated for each assumption that is made.

[0025] Figure 7 is flow diagram illustrating an embodiment of a process 700 for rolling over from a unit of assumption-based compiled code to fallback code. In the example shown, assumption-based compiled code 704 associated with a unit of source code 702 is optimized based on an assumption. The compiled code 704 includes code configured to execute a roll over to fallback code 706 that is not optimized based on the assumption, beginning at an associated entry point 716. In the example shown, the fallback code 706 is compiled code of the source code 702 and is not optimized based on the assumption. In the example shown, when the source code 702 is invoked (e.g., when a user browses a webpage containing the source code), the assumption-based compiled code 704 is executed 708. At a checkpoint 712, the assumption is tested. If the assumption is tested true, execution of the assumption-based compiled code 704 continues (not shown in Figure 7). If the assumption is tested false, rollover code included in the assumption-based compiled code 704 executes a rollover to fallback code 706, which may involve updating a runtime environment associated with the fallback code 706 to a state in which it would have been had the fallback code 706 been executing all along 714. Updating the runtime state may involve restoring local variables, register contents, stack states, memory contents, and/or other relevant machine states, which may be specific to the particular architectures/ABI involved. Execution of the fallback code 706 begins and continues from entry point 716 in the fallback code 706.

[0026] Figure 8 is a block diagram illustrating an embodiment of a system 800 for generating compiled code that is based on (e.g., optimized based on) an assumption and fallback code that is not based on the assumption. In the example shown, source code 802 is provided to a parser and initial compiler 804, which parses the source code and generates an initial intermediate representation. In various embodiments, the initial intermediate representation may be bytecode or another derived representation of the original source code. The initial intermediate representation is provided to an optimizing second stage compiler

806, which performs one or more rounds of optimization on the initially generated intermediate representation and provides as output a second intermediate representation of the source code 802, e.g., low level virtual machine (LLVM) or another low level intermediate representation. The optimization may be based on one or more assumptions, for example as discussed above in connection with Figures 2 & 3. The system also includes a machine code generator 808 that is configured to use the initial and/or the second (and/or other subsequent) intermediate representations to generate device-specific or computing-platform-specific machine code, e.g., machine code appropriate to processors of the x86, ARM, or other architectures. The generated machine code may be cached in the data storage and made available to each of one or more devices or systems, for example at a subsequent device synchronization time with the one or more devices or systems, machine code appropriate for that device or system. In some embodiments, the machine code generated based on the second intermediate representation and the machine code generated based on the initial intermediate representation are used as the compiled code optimized based on an assumption and the fallback code, respectively, for the processes discussed in reference to Figures 2 & 3.

[0027] Figure 9 is a block diagram of a computer system 900 used in some embodiments to process software code as described herein. Figure 9 illustrates one embodiment of a general purpose computer system. Other computer system architectures and configurations can be used for carrying out the processing described herein. Computer system 900, made up of various subsystems described below, includes at least one microprocessor subsystem (also referred to as a central processing unit, or CPU) 902. That is, CPU 902 can be implemented by a single-chip processor or by multiple processors. In some embodiments CPU 902 is a general purpose digital processor which controls the operation of the computer system 900. Using instructions retrieved from memory 910, the CPU 902 controls the reception and manipulation of input data, and the output and display of data on output devices.

[0028] CPU 902 is coupled bi-directionally with memory 910 which can include a first primary storage, typically a random access memory (RAM), and a second primary storage area, typically a read-only memory (ROM). A removable mass storage device 912 provides additional data storage capacity for the computer system 900. Storage 912 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. A fixed mass storage 920 can also provide additional data storage capacity. A common example of mass storage 920 is a hard disk drive.

[0029] In addition to providing CPU 902 access to storage subsystems, bus 914 can be used to provide access other subsystems and devices as well. In the described embodiment, these can include a display monitor 918, a network interface 916, a keyboard 904, and a pointing device 906, as well as an auxiliary input/output device interface, a sound card, speakers, and other subsystems as needed. The pointing device 906 may be a mouse, stylus, track ball, or tablet, and is useful for interacting with a graphical user interface. The network interface 916 allows CPU 902 to be coupled to another computer, computer network, or telecommunications network using a network connection as shown.

[0030] The computer system shown in Fig. 9 is but an example of a computer system suitable to implement techniques described herein. Other computer systems suitable for such use may include additional or fewer subsystems.

[0031] Figure 10 is a block diagram illustrating an embodiment of a system for processing a source code written in a traditionally interpreted dynamic language such as JavaScript. In the example shown, a device 1002 such as a phone, PDA, portable computer, or other device is directly connected up to a network via land wire and/or wirelessly through a gateway 1004 (in various embodiments through one or more other networks, nodes, and/or devices not shown in Figure 10), and via the network with one or more web or other devices (e.g., server 1008, 1010). In various embodiments, the device 1002 is configured to access web pages, web applications, applets, widgets, or other resources available for download from servers such as servers 1008 and 1010, all or some of which resources may include JavaScript or other traditionally interpreted code. In various embodiments, the device 1002 is configured to execute various computer codes such as a source code (e.g., a source code written in traditionally interpreted dynamic language such as JavaScript), a compiled code of a source code (e.g., compiled code that is based on or not based on an assumption), and/or other derived code of the source code such as a device independent intermediate representation of the source code (e.g., intermediate representation). In various embodiments, the device 1002 is configured to in some instances download computer code from another device or system (e.g., server 1008, 1010). In various embodiments, the device 1002 is configured to, upon the initial download of the source code written in interpreted

language such as JavaScript, generate, optimize, and/or cache an intermediate and/or other compile representation of the downloaded source code. In various embodiments, the device is configured to generate, optimize, and/or cache (e.g., prior to runtime or dynamically at runtime) a compiled machine code of the source code that is specific to the computing platform it is generated for and/or which is based on an assumption, as described above. In some embodiments, the device 1002 implements one or more of the processes of Figures 2-6. In various embodiments, the device is configured to cache the downloaded or generated code in data storage. In various embodiments, the device 1002 is configured to make the cached computer codes available to other devices (e.g., server 1008, 1010, device 1014, and personal computer 1012) upon synchronization with the one or more devices.

[0032] Although the foregoing embodiments have been described in some detail for purposes of clarity of understanding, the invention is not limited to the details provided. There are many alternative ways of implementing the invention. The disclosed embodiments are illustrative and not restrictive.

[0033] WHAT IS CLAIMED IS:

CLAIMS

1. A method for processing computer code, comprising:
executing compiled code associated with a unit of source code;
performing at runtime a test to determine whether an assumption on which the
5 compiled code is based is true; and
rolling over to and executing fallback code associated with the source code if the test
indicates the assumption is not true.
2. The method of claim 1, wherein the compiled code is optimized based on the
assumption.
- 10 3. The method of claim 1, where the compiled code includes instructions for testing the
assumption.
4. The method of claim 1, where the compiled code includes instructions for rolling
over to the fallback code if the assumption is tested to be not true.
5. The method of claim 1, wherein the fallback code comprises the original source code.
- 15 6. The method of claim 1, wherein the compiled code comprises a first compiled version
of the source code and the fallback code comprises a second compiled version of the source
code.
7. The method of claim 1, wherein rolling over comprises:
updating a runtime environment with which the fallback code is associated so that the
20 fallback code can be executed correctly from where the compiled code left off.
8. The method of claim 1, wherein the source code comprises code written in a
traditionally interpreted programming language.
9. The method of claim 1, wherein the source code comprises JavaScript.
10. A system for processing a unit of source code, comprising:
25 a processor configured to:
execute compiled code associated with a unit of source code;
perform at runtime a test to determine whether an assumption on which the compiled
code is based is true; and
roll over to and execute a fallback code associated with the source code if the test
30 indicates the assumption is not true; and

a memory coupled to the processor and configured to provide the processor with instructions.

11. The system of claim 10, wherein the compiled code is optimized based on the assumption.

5 12. The system of claim 10, where the compiled code includes instructions for testing the assumption.

13. The system of claim 10, where the compiled code includes instructions for rolling over to the fallback code if the assumption is tested to be not true.

14. The system of claim 10, wherein the fallback code comprises the original source code.

10 15. The system of claim 10, wherein the compiled code comprises a first compiled version of the source code and the fallback code comprises a second compiled version of the source code.

16. A computer program product for processing source code, the computer program product being embodied in a computer readable storage medium and comprising computer
15 instructions for:

executing compiled code associated with a unit of source code;

performing at runtime a test to determine whether an assumption on which the compiled code is based is true; and

rolling over to and execute a fallback code associated with the source code if the test
20 indicates the assumption is not true.

17. A method of processing source code, comprising:

providing an assumption-based compiled code that comprises a compiled version of the source code and that is based at least in part on an assumption;

providing checkpoint code configured to test at runtime whether the assumption is
25 true; and

providing roll over code configured to invoke execution of a fallback code associated with the source code, and cease execution of the assumption-based compiled code, if the test performed by the checkpoint code at runtime indicates the assumption is not true.

18. The method of claim 17, wherein the checkpoint code is included in the assumption-
30 based compiled code.

19. The method of claim 17, wherein the fallback code comprises the source code.

20. The method of claim 17, wherein the assumption-based compiled code comprises a first compiled version of the source code and the fallback code comprises a second compiled version of the source code.

21. The method of claim 17, wherein providing one or more of the assumption-based
5 compiled code, the checkpoint code, and the roll over code comprises generating such code based on one or both of the source code and an intermediate representation thereof.

22. A system, comprising:

a memory configured to store source code; and

a processor coupled to the memory and configured to:

10 generate based on the source code an assumption-based compiled code that is based at least in part on an assumption;

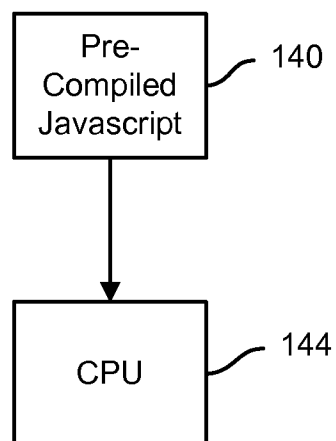
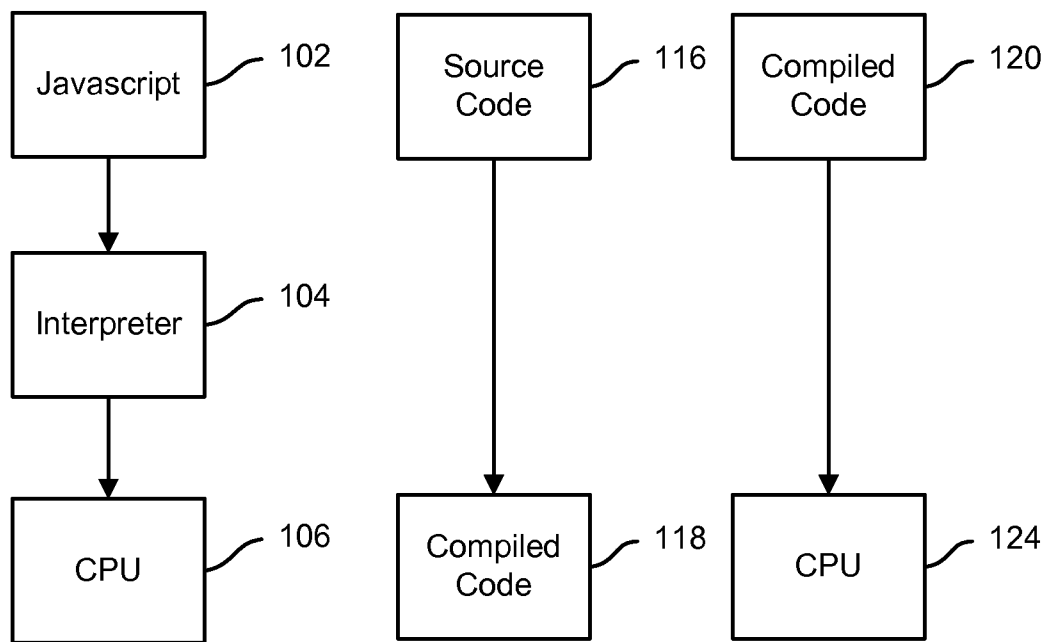
generate checkpoint code to test at runtime whether the assumption is true; and

15 generate roll over code configured to invoke a fallback code associated with the source code, and cease execution of the assumption-based compiled code, if the test performed by the checkpoint code at runtime indicates the assumption is not true.

23. The system of claim 22, wherein the compiled code is optimized based on the assumption.

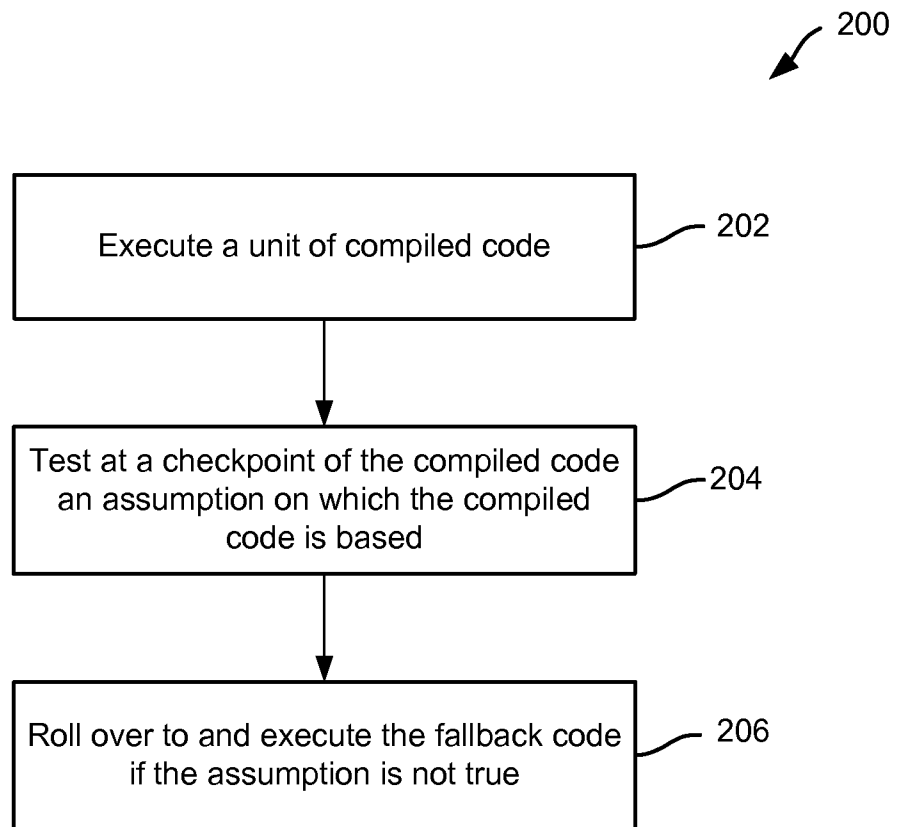
24. The system of claim 22, wherein the fallback code comprises the original source code.

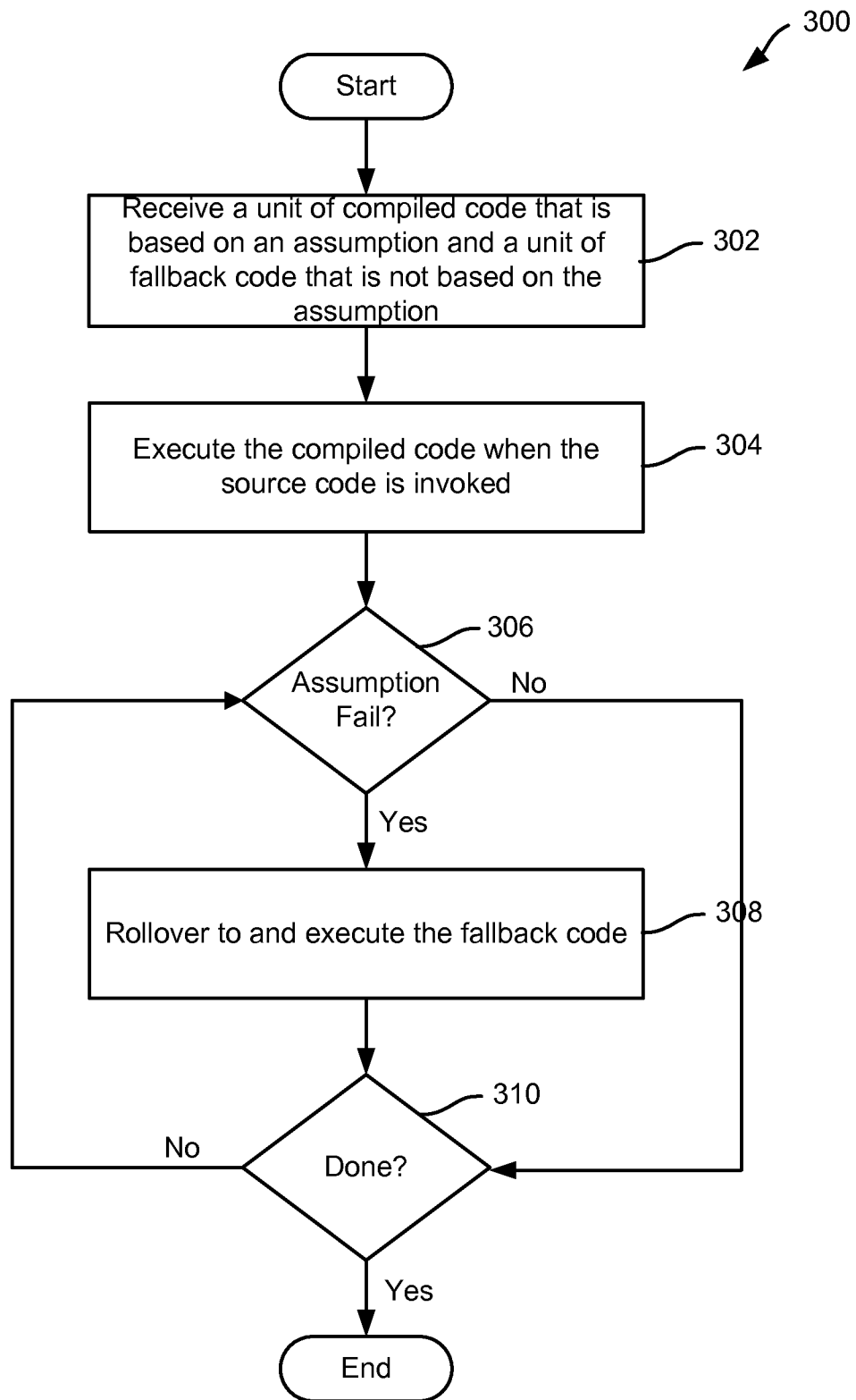
20 25. The system of claim 22, wherein the compiled code comprises a first compiled version of the source code and the fallback code comprises a second compiled version of the source code.

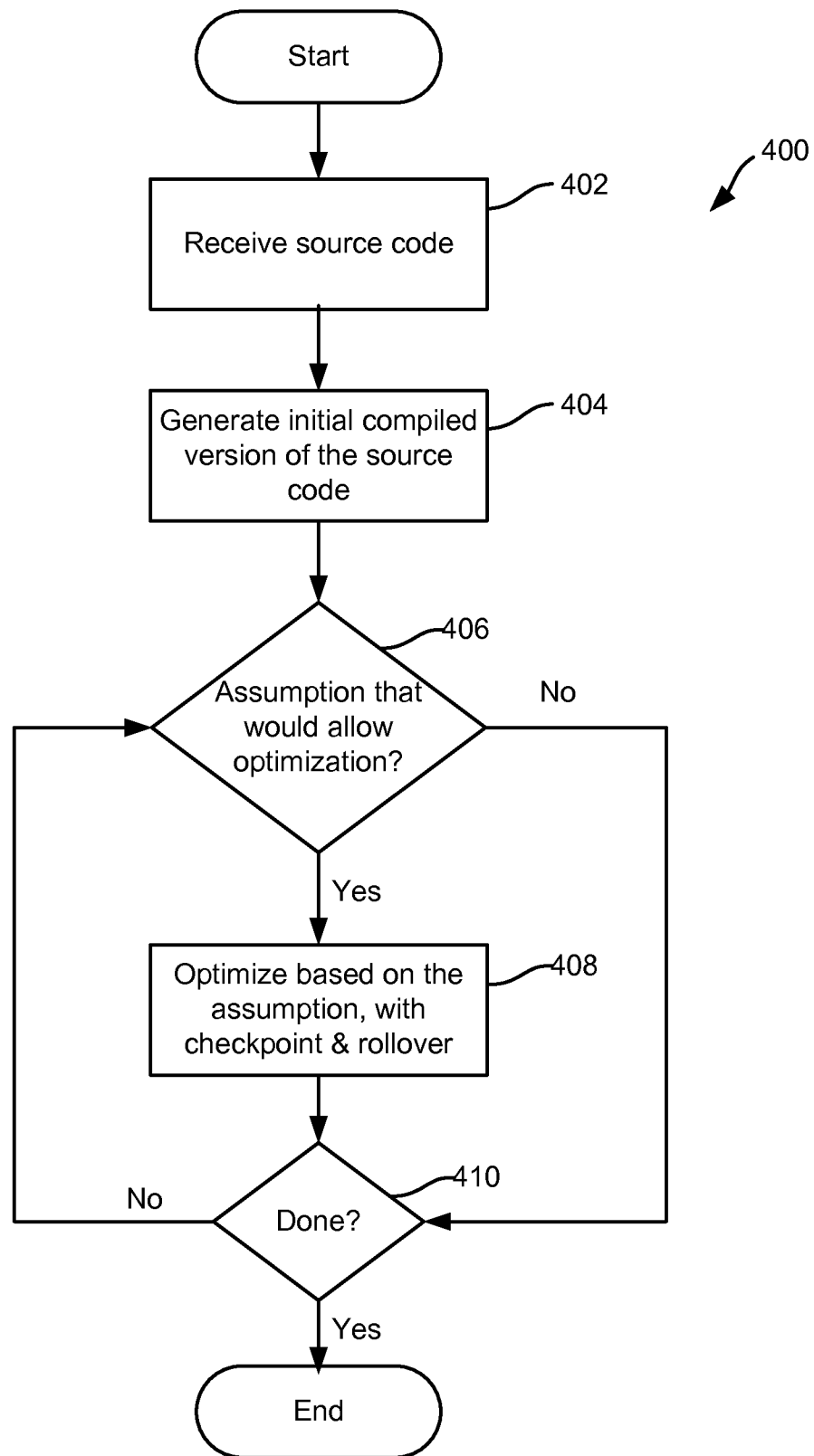


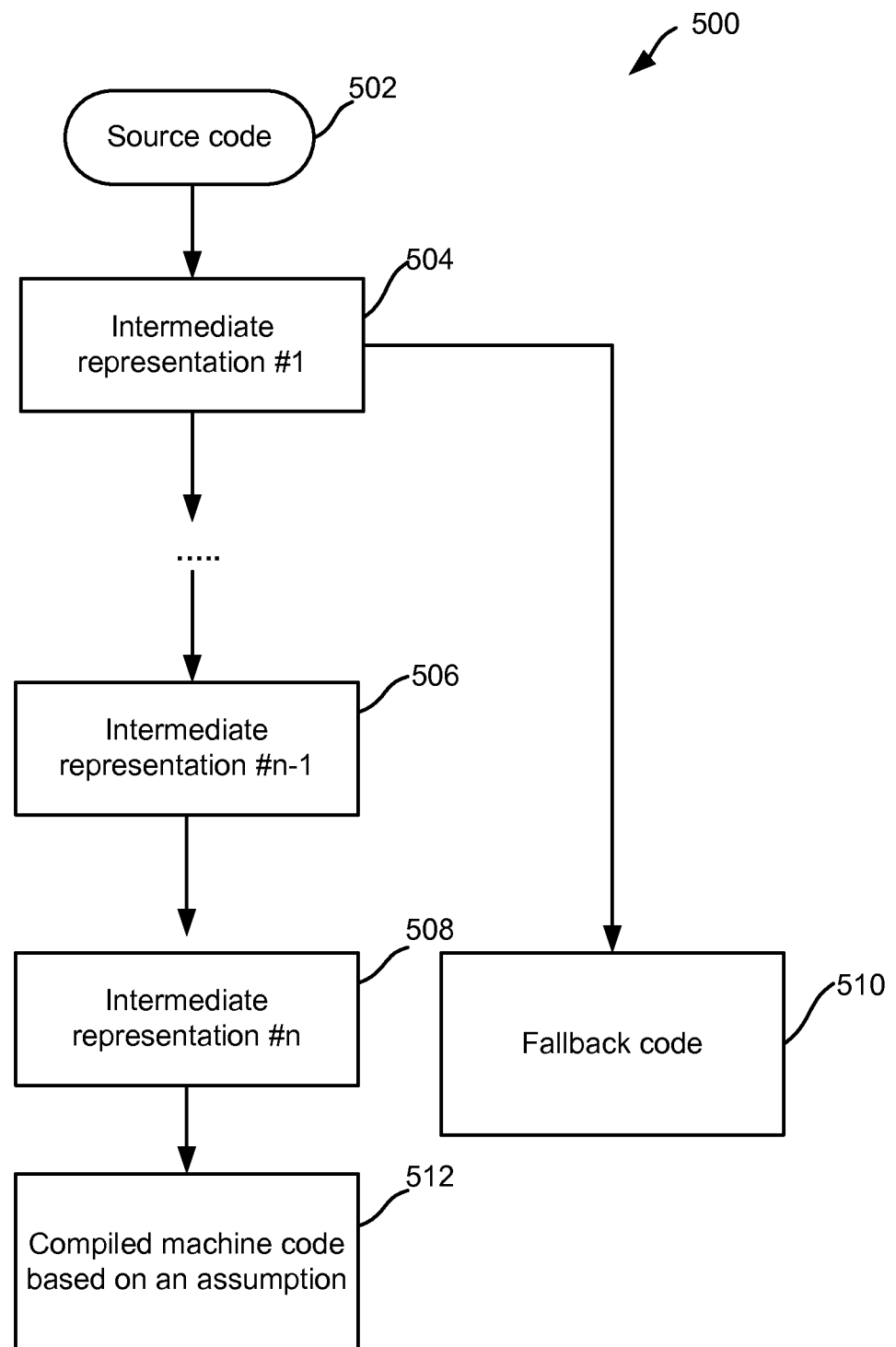
PRIOR ART

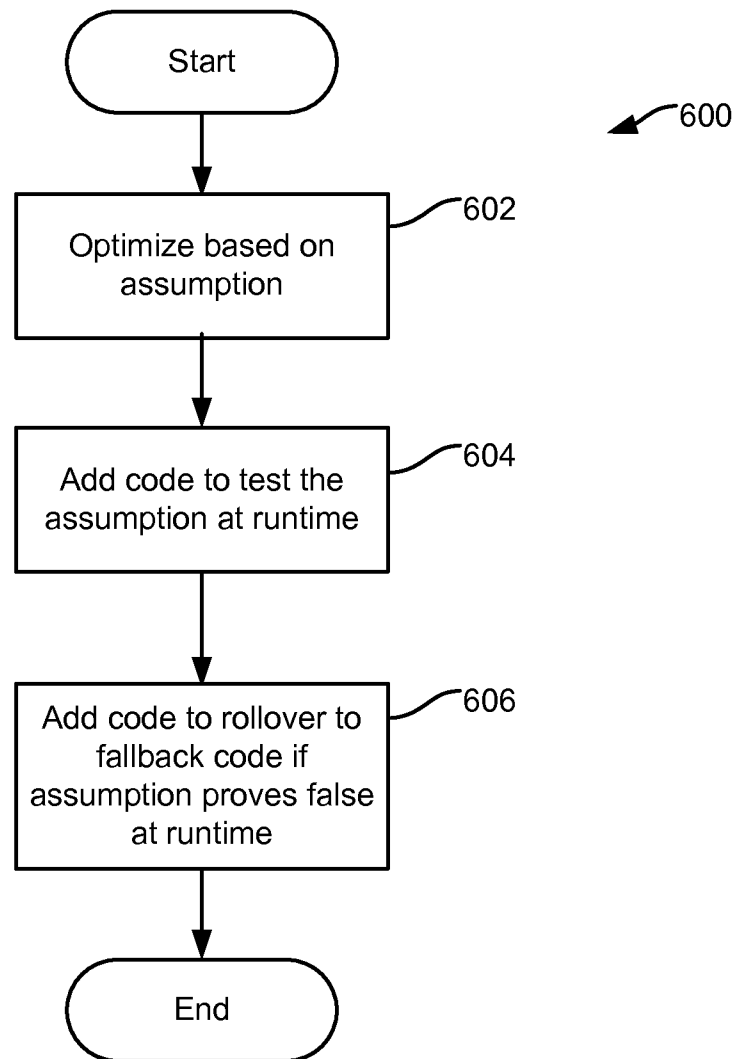
FIG. 1

**FIG. 2**

**FIG. 3**

**FIG. 4**

**FIG. 5**

**FIG. 6**

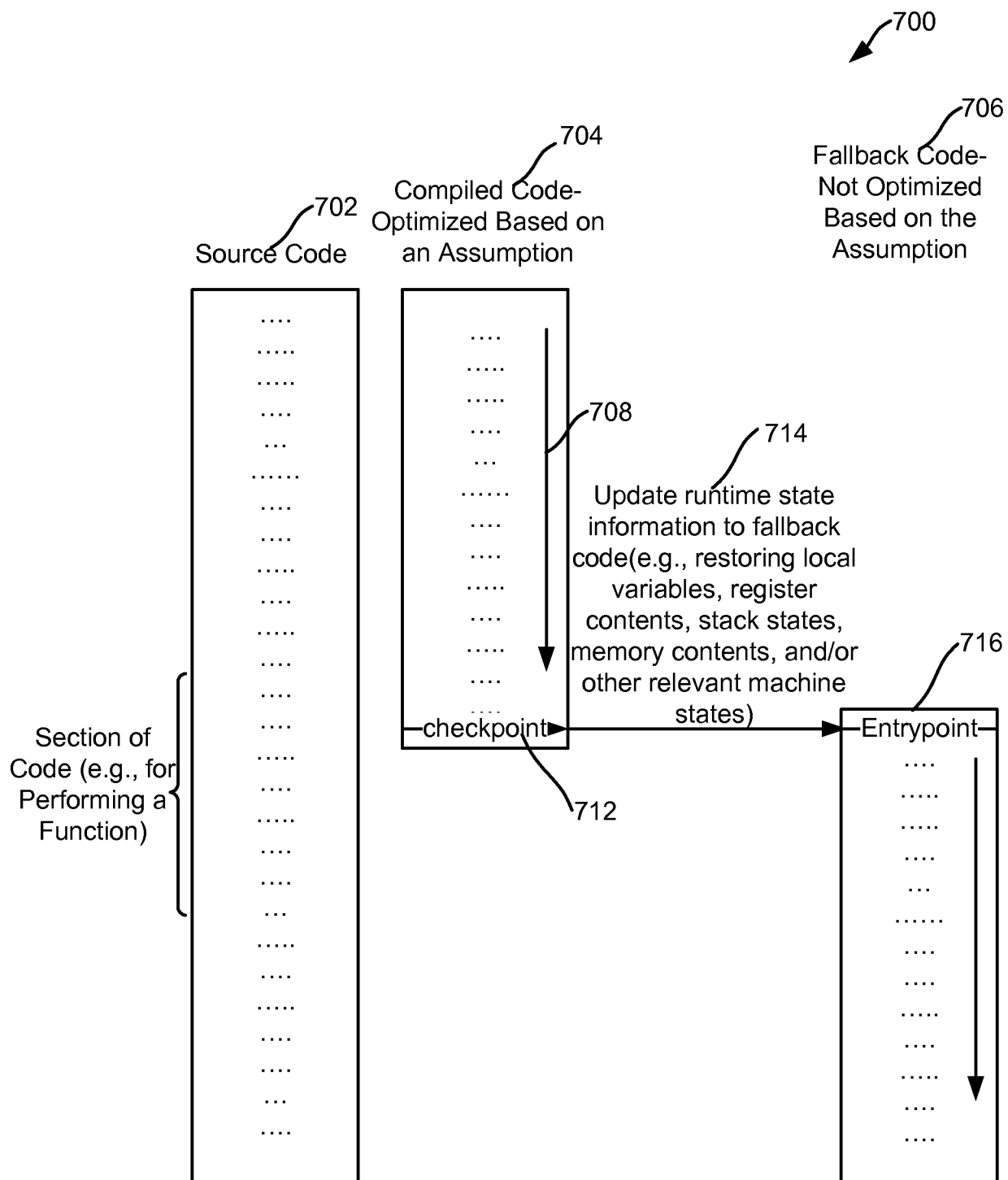


FIG. 7

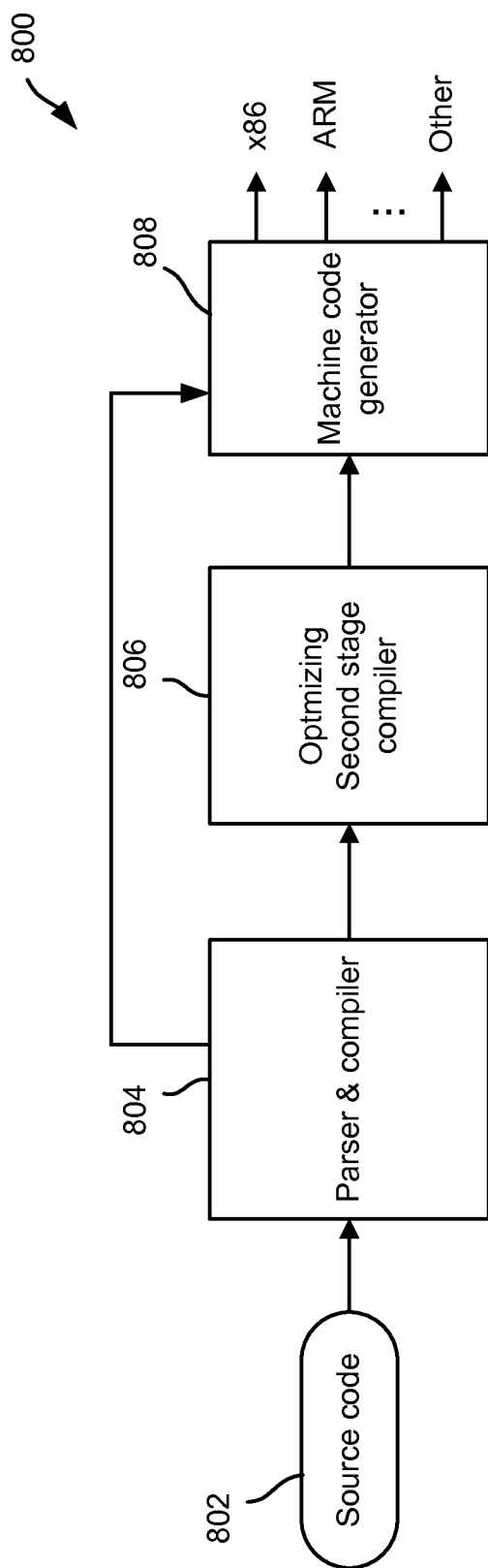
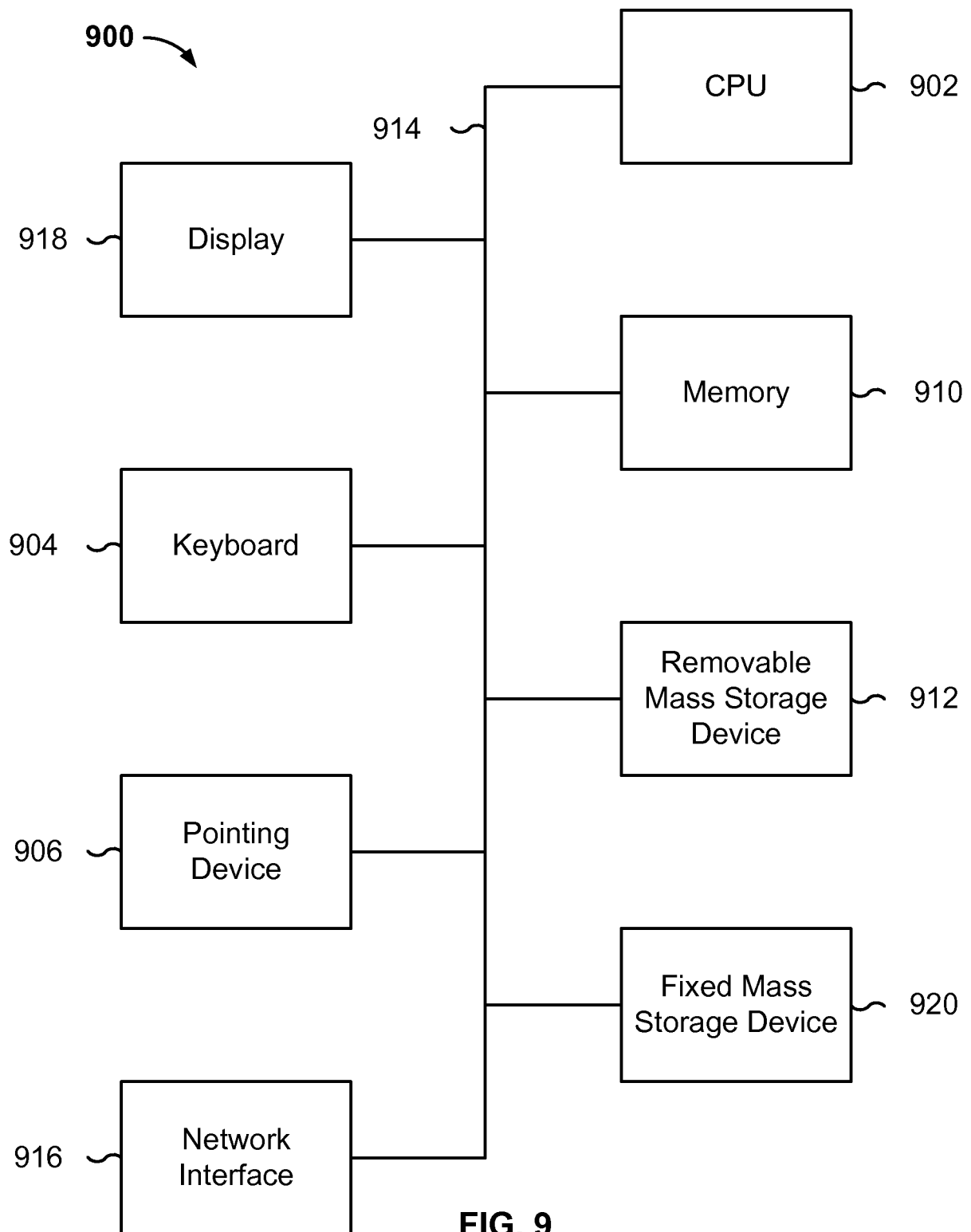


FIG. 8

**FIG. 9**

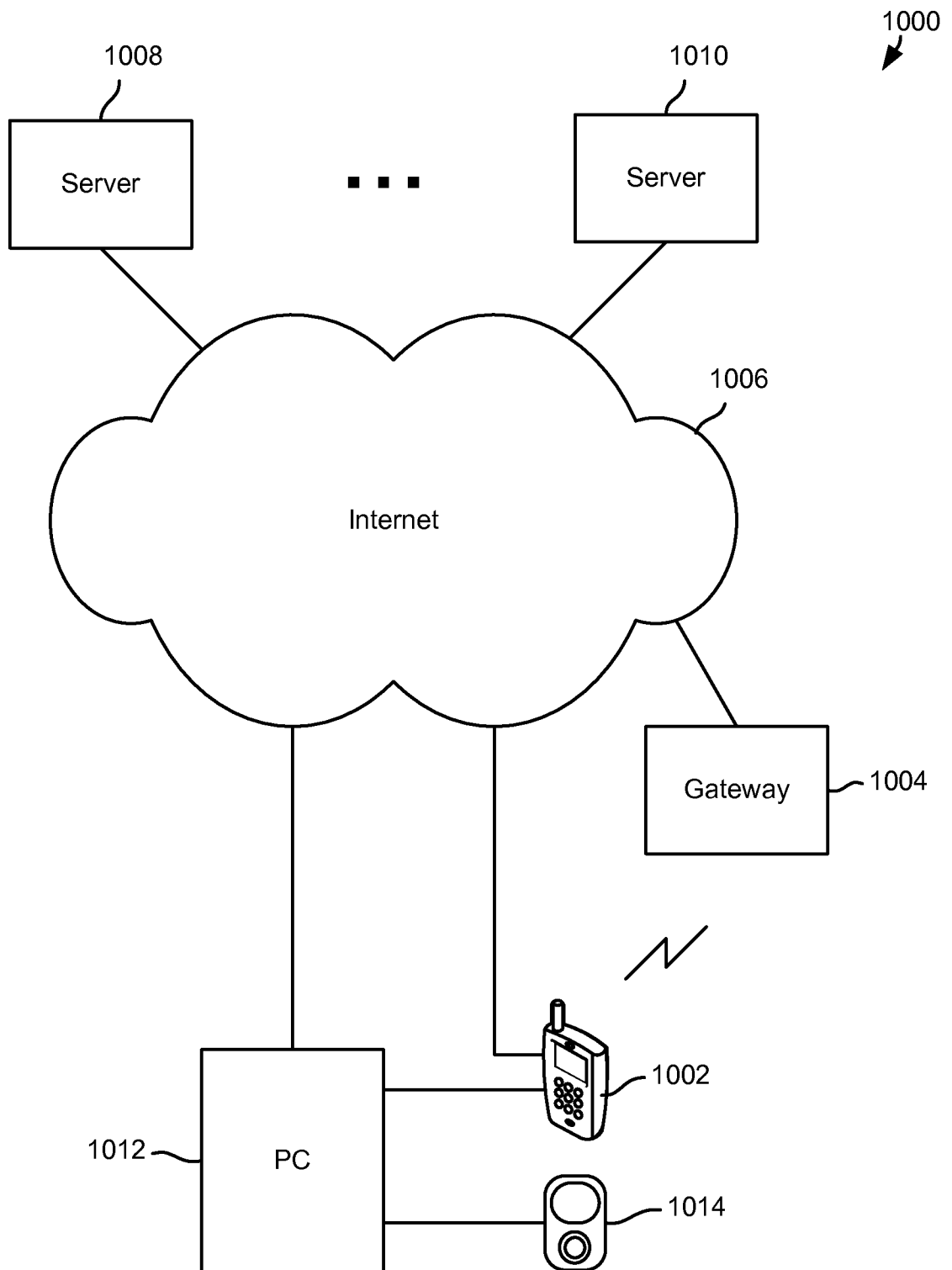


FIG. 10

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2011/044895

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/45
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EP0-Internal, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2008/127134 A1 (TIRUMALAI PARTHA P [US] ET AL) 29 May 2008 (2008-05-29)	1-4,6, 10-13, 15-18, 20-23,25
Y	abstract; figures 4-7 page 1, paragraph 11 - page 1, paragraph 12 page 2, paragraph 23 - page 2, paragraph 30 page 2, paragraph 34 - page 3, paragraph 37 page 4, paragraph 46 - page 4, paragraph 48; claims 1-5,7-11,13-17 ----- -/--	5,7-9, 14,19,24

☒ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search

22 September 2011

Date of mailing of the international search report

30/09/2011

Name and mailing address of the ISA/

European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040,
Fax: (+31-70) 340-3016

Authorized officer

Lelait, Sylvain

INTERNATIONAL SEARCH REPORT

International application No

PCT/US2011/044895

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	ANDREAS GAL ET AL: "Trace-based just-in-time type specialization for dynamic languages", PROCEEDINGS OF THE 2009 ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, PLDI '09, vol. 44, 1 January 2009 (2009-01-01), page 465, XP55007784, New York, New York, USA ISSN: 0362-1340, DOI: 10.1145/1542476.1542528 ISBN: 978-1-60-558392-1	1-4, 10-13, 16-18,22
Y	abstract page 465, right-hand column, paragraph 2 - page 466, left-hand column, paragraph 2 page 466, right-hand column, paragraph 3 - page 468, left-hand column, paragraph 4; figure 2 page 468, left-hand column, paragraph 8 - page 469, left-hand column, paragraph 5 page 469, right-hand column, paragraph 2 - page 469, right-hand column, paragraph 6 page 471, right-hand column, last paragraph - page 472, left-hand column, paragraph 13 page 472, left-hand column - page 473, left-hand column, paragraph 6 -----	5,7-9, 14,19,24
X	US 6 658 656 B1 (THOMPSON CAROL L [US]) 2 December 2003 (2003-12-02)	1-4,6, 10-13, 15-17, 20-23,25 7
Y	abstract; figures 2,4 column 1, line 64 - column 2, line 17 column 2, line 53 - column 3, line 65 column 4, line 11 - column 5, line 35; claims 1-4,6,7,9-11,13-15 -----	
Y	MAXIME CHEVALIER-BOISVERT ET AL: "Optimizing Matlab through Just-In-Time Specialization", 20 March 2010 (2010-03-20), COMPILER CONSTRUCTION, SPRINGER BERLIN HEIDELBERG, BERLIN, HEIDELBERG, PAGE(S) 46 - 65, XP019138830, ISBN: 978-3-642-11969-9 page 49, paragraph 1 - page 53, paragraph 2; figure 1 -----	5,7,14, 19,24

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No

PCT/US2011/044895

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008127134 A1	29-05-2008	NONE	
US 6658656 B1	02-12-2003	US 2004034814 A1	19-02-2004