(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 November 2013 (07.11.2013)

WIPO | PCT

(10) International Publication Number
WO 2013/165461 Al

(51) International Patent Classification:
G06F 9/45 (2006.0 1) *G06F 9/30* (2006.0 1)

(21) International Application Number:
PCT/US20 12/0567 11

(22) International Filing Date:
2 1 September 2012 (21 .09.2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
13/461,757 1 May 2012 (01.05.2012) US

(71) Applicant *(for all designated States except US):* **CONCURIX CORPORATION** [US/US]; 244 Market Street, Kirkland, Washington 98033 (US).

(72) Inventors; and

(71) Applicants *(for US only):* **GOUNARES, Alexander G.** [US/US]; 2 17 7th Ave W., Kirkland, Washington 98033 (US). **GARRETT, Charles D.** [US/US]; 17641 167th Ave NE, Woodinville, Washington 98072 (US).

(74) Agent: **KRAJEC, Russell S.;** Krajec Patent Offices, LLC, 1635 Foxtrail Drive, Suite 321, Loveland, Colorado 80538 (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, BN, 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, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, 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.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available):* ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, 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).

[Continued on nextpage]

(54) Title: RECOMPILING WITH GENERIC TO SPECIFIC REPLACEMENT

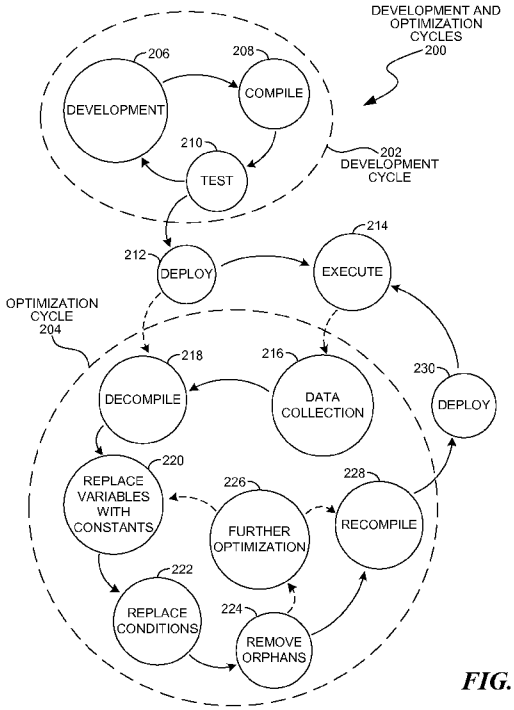

FIG. 2

(57) Abstract: Executable code may be recompiled so that generic portions of code may be replaced with specific portions of code. The recompilation may customize executable code for a specific use or configuration, making the code lightweight and executing faster. The replacement mechanism may replace variable names with fixed values, replace conditional branches with only those branches which are known to be executed, and may eliminate executable code portions that are not executed. The replacement mechanism may comprise identifying known values defined in the executable code for variables, and replacing those variables with the constant value. Once the constants are substituted, the code may be analyzed to identify branches that may be evaluated using the constant values. Those branches may be reformed using the constant value and the rest of the conditional code that may not be accessed may be removed.

WO 2013/165461 A1

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(iv))*

Published:

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

Recompiling with Generic to Specific Replacement

Claim of Priority

[0001] This application claims the benefit priority to U.S. Patent Application No. 13/461,757, filed May 1, 2012, entitled "Recompiling with Generic to Specific Replacement", which is incorporated herein by reference in its entirety.

Background

[0002] Many computer programs are written in a reusable style, where functions may be called using variables that may change each time the function may be called. Such a programming style has many uses, such as being able to test and verify each function independently, as well as having sections of executable code that may be reused over and over.

[0003] In order to create executable code from such a program, a compiler and linker may create and track all of the connections between many different sections of code.

Summary

[0004] Executable code may be recompiled so that generic portions of code may be replaced with specific portions of code. The recompilation may customize executable code for a specific use or configuration, making the code lightweight and executing faster. The replacement mechanism may replace variable names with fixed values, replace conditional branches with only those branches which are known to be executed, and may eliminate executable code portions that are not executed. The replacement mechanism may comprise identifying known values defined in the executable code for variables, and replacing those variables with the constant value. Once the constants are substituted, the code may be analyzed to identify branches that may be evaluated using the constant values. Those branches may be reformed using the constant

value and the rest of the conditional code that may not be accessed may be removed.

[0005] This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter.

Brief Description of the Drawings

[0006] In the drawings,

[0007] FIGURE 1 is a diagram illustration of an embodiment showing a system with code optimization.

[0008] FIGURE **2** is a diagram illustration of an embodiment showing development and optimization cycles.

[0009] FIGURE **3** is a diagram illustration of an embodiment showing an example deployment in a datacenter.

[0010] FIGURE 4 is a flowchart illustration of an embodiment showing a method for optimizing code.

[0011] FIGURE 5 is a flowchart illustration of an embodiment showing a method for optimizing conditions within code.

Detailed Description

[0012] Executable code may be recompiled to make the executable code more specific for an instance that may be executed. The code may be analyzed to identify variables that may be replaced by constants, then branches of the code that can be determined given the constants may be evaluated and streamlined. In some cases, code that cannot be reached once the constants are known may be removed from the executable code.

[0013] Recompiling executable code may attempt to streamline and speed up execution of the code by replacing variables with constants. The executable code may be developed, tested, and operated as a generic code, then

recompiled to speed up execution for a specific execution instance. In some embodiments, the recompiling and optimization may occur at runtime, while other embodiments may recompile and optimize prior to runtime.

[0014] The executable code may be any executable code, including application code, operating system code, services code, or any other set of executable instructions. In many embodiments, executable code that may be written in functional languages or written in functional language style may enable an analysis engine to identify constant values for variables. Since functional languages and programs written in a functional language style may have no mutable state, an analysis engine may readily identify those elements that do not change and may propagate those values throughout the code.

[0015] For the purposes of this specification and claims, the term "executable element" may define a set of instructions that may be executed by a processor. In a typical embodiment, an executable element may be machine level commands that may be sent to a processor. A single computer application may be made up of many executable elements. An executable element may also be referred to as a job, application, code chunk, or other term.

[0016] Throughout this specification, like reference numbers signify the same elements throughout the description of the figures.

[0017] When elements are referred to as being "connected" or "coupled," the elements can be directly connected or coupled together or one or more intervening elements may also be present. In contrast, when elements are referred to as being "directly connected" or "directly coupled," there are no intervening elements present.

[0018] The subject matter may be embodied as devices, systems, methods, and/or computer program products. Accordingly, some or all of the subject matter may be embodied in hardware and/or in software (including firmware, resident software, micro-code, state machines, gate arrays, etc.) Furthermore, the subject matter may take the form of a computer program product on a computer-usable or computer-readable storage medium having computer-usable or computer-readable program code embodied in the medium for use by or in connection with an instruction execution system. In the context of this document, a computer-usable or computer-readable medium may be any

medium that can contain, store, communicate, propagate, or transport the program for use by or in connection with the instruction execution system, apparatus, or device.

[0019] The computer-usable or computer-readable medium may be, for example but not limited to, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, device, or propagation medium. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media.

[0020] Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by an instruction execution system. Note that the computer-usable or computer-readable medium could be paper or another suitable medium upon which the program is printed, as the program can be electronically captured, via, for instance, optical scanning of the paper or other medium, then compiled, interpreted, of otherwise processed in a suitable manner, if necessary, and then stored in a computer memory.

[0021] When the subject matter is embodied in the general context of computer-executable instructions, the embodiment may comprise program modules, executed by one or more systems, computers, or other devices. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Typically, the functionality of the program modules may be combined or distributed as desired in various embodiments.

[0022] Figure 1 is a diagram of an embodiment 100 showing a system that may analyze executable code to replace generic code with specific code. Embodiment 100 is a simplified example of the various software and hardware

components that may be used an execution environment for applications and an operating system.

[0023] The diagram of Figure 1 illustrates functional components of a system. In some cases, the component may be a hardware component, a software component, or a combination of hardware and software. Some of the components may be application level software, while other components may be operating system level components. In some cases, the connection of one component to another may be a close connection where two or more components are operating on a single hardware platform. In other cases, the connections may be made over network connections spanning long distances. Each embodiment may use different hardware, software, and interconnection architectures to achieve the functions described.

[0024] Embodiment 100 illustrates a computer system 102 that may create specific code for execution from generic code. The specific code may attempt to replace variables with constants, simplify or remove branches in the generic code, and otherwise streamline the code for execution.

[0025] Widely used programming techniques teach programmers to create functions, routines, or other code blocks in a generic manner. Such functions may accept a variable or set of inputs, perform a function, and return a result. The function may be called from many different locations within an application, and the code may be reused for many different input values. Such a function may be tested and validated as a separate unit of code.

[0026] Another widely used programming technique may be to create branches for different conditions in a program. A branch may determine that one function may be executed when a value of X is received, while another function may be executed when a value of Y is received. When a branch is determined, a function may be called, causing a computer to keep track of all the function calls using a set of internal pointers and a stack storage mechanism.

[0027] Many advanced processors may attempt to identify code that will be executed in the future. Such processors may load the code into a cache memory prior to being executed to avoid a delay when the executable code may be retrieved from main memory. In some cases, code executed from cache may be performed on subsequent clock cycles, while code that may be retrieved from

main memory may cause a processor to wait many clock cycles, sometimes over 100 clock cycles later, before the next instruction may be processed.

[0028] At a code branch point, the advanced processors may not be able to look ahead and determine which branch of the code may be executed next. Code branches often cause slow performance as the next function or execution may wait until the executable code may be retrieved from main memory after the code branch condition may be evaluated.

[0029] An optimization routine for executable code may attempt to minimize or remove branch points by replacing variables with any known constants. Based on the constants, branch points in the executable code may be analyzed to identify branch conditions that may be known. When the branch may be fully determined prior to execution, the branching code may be removed and replaced with a pointer to the code associated with the only branch that may be possible.

[0030] The optimization routine may be performed prior to execution. In some cases, the optimization routine may be part of a just-in-time compiler or some other runtime environment that may prepare and manage applications being executed. In other cases, the optimization routine may be part of an operating system, where an analyzer may decompile, analyze, and recompile code for execution. In still other cases, the optimization routine may be performed in a development environment by analyzing source code.

[0031] The optimization routine may be performed on programs written in functional languages or in a functional language style. Functional language style consolidates blocks of executable code into units that receive input and return an output without otherwise changing state or receiving state that may be defined outside of the function. Functional language style may treat each block of executable code as independent, separate operations that may not depend on any other input except for that which is passed to the function, as well as not changing the state or value of other objects, except for that which is returned by the function. In many functional languages, a message passing mechanism may transmit and receive the values of parameters passed between functions.

[0032] Many functional languages, such as Erlang, Scala, F#, and others may enforce such restrictions on code that may be written and compiled in those

languages. Other languages may also be used to write in a functional language style and yield similar results.

[0033] The optimization may be readily applied to programs written in a functional language style because the state of variables may be defined when messages are passed between functions. As such, the analysis may be performed by analyzing the connections between functional elements.

[0034] In contrast, non-functional languages or applications written in a non-functional language style may allow some variables to be changed, updated, or retrieved within a function, where those variables may not be expressly passed between the functions. In such cases, the analysis for an optimization routine may have more difficulty identifying when variables are changed.

[0035] Some applications may be written in a language or style that may identify very small functions that may be linked together in an application. For example, some Erlang programs of moderate size may include many hundreds of thousands if not millions of individual functions connected by message passing. Such situations may allow for fine-grained analysis of parameters passed between functions.

[0036] In contrast, some applications written in a functional language style may have large blocks of code that may be treated as independent functions. Some such applications may identify only a handful to several hundred functions that may be considered independent functions. Such situations may allow for a more coarse-grained analysis.

[0037] In general, more fine-grained analyses may result in more dramatic improvements in processing speed than more coarse-grained analyses.

[0038] The device 102 is illustrated having hardware components 104 and software components 106. The device 102 as illustrated represents a conventional computing device, although other embodiments may have different configurations, architectures, or components.

[0039] In many embodiments, the device 102 may be a server computer. In some embodiments, the device 102 may still also be a desktop computer, laptop computer, netbook computer, tablet or slate computer, wireless handset, cellular telephone, game console or any other type of computing device.

[0040] The hardware components 104 may include a processor 108, random access memory 110, and nonvolatile storage 112. The hardware components 104 may also include a user interface 114 and network interface 116. The processor 108 may be made up of several processors or processor cores in some embodiments. The random access memory 110 may be memory that may be readily accessible to and addressable by the processor 108. The nonvolatile storage 112 may be storage that persists after the device 102 is shut down. The nonvolatile storage 112 may be any type of storage device, including hard disk, solid state memory devices, magnetic tape, optical storage, or other type of storage. The nonvolatile storage 112 may be read only or read/write capable.

[0041] The user interface 114 may be any type of hardware capable of displaying output and receiving input from a user. In many cases, the output display may be a graphical display monitor, although output devices may include lights and other visual output, audio output, kinetic actuator output, as well as other output devices. Conventional input devices may include keyboards and pointing devices such as a mouse, stylus, trackball, or other pointing device. Other input devices may include various sensors, including biometric input devices, audio and video input devices, and other sensors.

[0042] The network interface 116 may be any type of connection to another computer. In many embodiments, the network interface 116 may be a wired Ethernet connection. Other embodiments may include wired or wireless connections over various communication protocols.

[0043] The software components 106 may include an operating system 118 on which various applications and services may operate. An operating system may provide an abstraction layer between executing routines and the hardware components 104, and may include various routines and functions that communicate directly with various hardware components.

[0044] The operating system 118 may be defined using various functional components 120. The components 120 may be blocks of code that operate in a functional language style. In some cases, the components 120 may be written in a functional language.

[0045] An analyzer 122 may include a decompiler 124 and compiler 126 and may create optimized executable code using the operating system components 120 as well as applications 144. The optimized executable code may be incorporated into the device's executing code and caused to run.

[0046] Embodiment 100 illustrates an embodiment where a single device may optimize code that runs on the device. Embodiment 300 presented later in this specification illustrates an embodiment where optimized code may be created and deployed to other devices.

[0047] The device 102 may have an execution environment 128 that may analyze and optimize application 144 for execution. An execution environment may be a virtual machine or other environment that may manage execution of applications.

[0048] The execution environment 128 may include a just-in-time compiler 130. Some programming environments may create applications in source code, and then compile the source code into intermediate code. The intermediate code may then be compiled at runtime into executable code by a just-in-time compiler 130. The just-in- time compiler 130 may operate with a linker 132 to create a full set of executable code.

[0049] The execution environment 128 may have a data collector 134 and an analyzer 136 that may create optimized code. The data collector 134 may determine the value of certain constants prior to execution, and the analyzer 136 may create the optimized code by inserting the constant values and analyzing any branches or conditions within the code.

[0050] A development environment 138 may include some programming tools that a developer or programmer may use to create the applications 144. The development environment 138 may include an editor 140 and compiler 142.

[0051] Figure 2 is a diagram illustration of an embodiment 200 showing steps that may create optimized code. Embodiment 200 may show the development cycle 202 and optimization cycle 204 that may result in optimized executable code.

[0052] In the development cycle 202, a programmer may develop code in step 206, compile code in step 208, and test the code in step 210. The

development cycle 202 may iterate many times until the programmer may deploy the code in step 212.

[0053] The deployed code from step 212 may be executed in step 214 without optimization. In such a case, the deployed code may perform all of its functions as designed. After optimization, the optimized executable code may perform the same functions, but with a much faster throughput.

[0054] The optimization cycle 204 may collect data in step 216 to determine any known parameter values. In some embodiments, the data collection step 216 may be a separate application, function, or mechanism to collect information for optimization. For example, an optimization that may include operating system functions may perform a data collection to identify the available hardware, network addresses, and other information that may be used to optimize the performance of the operating system. Such a data collection may be performed by a routine that may be part of an optimization mechanism.

[0055] In some embodiments, a data collection step 216 may be performed as a profiled or instrumented analysis of the executable code. In such embodiments, a profiler may collect data while the code may be executing. The data may be analyzed to identify variables that have known or consistent values.

[0056] The executable code may be decompiled in step 218. Some embodiments may decompile machine code or intermediate code to create a higher level code that may be more easily analyzed. Other embodiments may access and optimize source code, intermediate code, machine code, or some other form of code, and such embodiments may not perform step 218.

[0057] Variables may be replaced by constants in step 220. The variables may be those variables determined from step 216 or from analysis of the executable code. The replacements performed in step 220 may remove references to a parameter and replace those references with the actual value of the parameter.

[0058] The replacement of step 220 may provide some performance improvements for executable code. In many cases, a reference to a variable value may cause an executing processor to perform a lookup or memory access for that variable. When the variable name may be replaced by a constant value,

the processing overhead of accessing the variable value may be avoided, thereby increasing performance.

[0059] An analysis of conditional statements may identify conditional statements that may be replaced in step 222. The conditional statements that may be replaced may be those for which the variable values are known and the condition may be evaluated at the time the optimization cycle 204 may be executed. The replacement of conditional statements may remove conditional statements and replace those statements with direct jumps or other pointers to the portion of code that may be executed when the condition is satisfied.

[0060] The replacement of conditional statements may provide increased performance, as a processor may be able to 'look ahead' and prepare to execute the code when the condition is satisfied. When a conditional statement exists, a processor may not be able to 'look ahead', as the condition may not be known. In such a situation, the processor may evaluate the condition, then may have to wait as the next set of instructions may be retrieved from main memory prior to continuing execution. By eliminating the wait time associated with processing a condition, performance may be increased.

[0061] Orphan code may be identified and removed in step 224. Orphan code may be portions of the executable code that may not be reached during execution. Orphan code may be identified when a condition is replaced in step 222, which may leave an orphaned code block that may not be reached when the condition is removed.

[0062] In some embodiments, the process of step 224 may involve creating a control flow graph of the executable code, then analyzing the control flow graph to identify orphaned code.

[0063] The process may iterate in step 226 to perform further optimization, in which the process may return to step 220.

[0064] Once the optimization is ready for execution, the code may be recompiled in step 228 and deployed in step 230. In some embodiments, the optimization cycle 204 may be performed repeatedly. In one such embodiment, a profiler may gather further data in the data collection step 216 and cause the optimization to be performed again.

[0065] Figure 3 is a diagram illustration of an example embodiment 300 showing a customized deployment of code for an enterprise application. Embodiment 300 is one example of how a code optimization mechanism may be used to deploy customized code to servers in a server farm.

[0066] Embodiment 300 illustrates a server farm 302 that may have several servers 304, 306, 308, and 310. In many embodiments, a server farm may have many hundreds, thousands, or even hundreds of thousands of servers. The servers may be connected to a network 312 which may also be connected to a manager 314.

[0067] The manager 314 may create and distribute operating systems and applications to the various servers in the form of optimized executable code. While the manager 314 may be illustrated as a single device, other embodiments may have one or more functions described for the manager 314 to be performed by separate devices.

[0068] The manager 314 may operate on a hardware platform 316, which may be similar to the hardware platform 106 of embodiment 100.

[0069] The manager 314 may have a code generator 324 which may create customized executable code using various applications 326 and a full copy of an operating system 328. The code generator 324 may create a customized executable that is optimized for each specific server. In many cases, different servers may perform different tasks, and as such, the executable code may be optimized for that task.

[0070] The code generator 324 may combine the operating system 328 and applications 326 into a single executable, and may perform at least some optimization on the code using, as an example, the process described in embodiment 200. The code generator 324 may create an optimized version of code as described by a system manager 322.

[0071] A system manager 322 may determine which server may execute which code, then pass the configuration to the code generator 324. The code generator 324 may then create a configured operating system 320 that may contain an operating system and applications. In some embodiments, the configured operating system 320 may contain only an operating system and applications for a particular server may be downloaded separately.

[0072] In some cases, the code generator 324 may operate on decompiled code, intermediate code, or source code, and may use one or more compilers 330 to create the executable code.

[0073] A Pre-eXecution Environment (PXE) boot server 318 may receive requests from the various servers at boot time. As part of the communication, a server may request bootable code from the PXE boot server 318. The system manager 322 may determine which version of the configured operating system 320 may be transmitted to the server, and the PXE boot server 318 may transmit the bootable code to the server. In such an embodiment, a set of pre-configured operating systems 320 may be available prior to receiving a request from a server.

[0074] In some embodiments, the manager 314 may receive a request for bootable code and then the system manager 322 may cause the code generator 324 to create optimized code. In such an embodiment, the system manager 322 may determine a configuration after the request is received.

[0075] Embodiment 300 is an example of a distribution mechanism that may be deployed in a datacenter environment. In other embodiments, a code generator may create optimized code for desktop computers, slate and tablet computers, mobile telephones, hand held scanning devices, sensor devices, or any other computing device. In many cases, the optimization mechanism may speed up the execution of the code while maintaining the same functionality of non-optimized code.

[0076] Figure 4 is a flowchart illustration of an embodiment 400 showing a method for optimizing code for execution. Embodiment 400 illustrates the operations of an analyzer, such as the analyzers 122 and 136 of embodiment 100, as well as the operations of a code generator, such as the code generator 324 of embodiment 300.

[0077] Other embodiments may use different sequencing, additional or fewer steps, and different nomenclature or terminology to accomplish similar functions. In some embodiments, various operations or set of operations may be performed in parallel with other operations, either in a synchronous or asynchronous manner. The steps selected here were chosen to illustrate some principles of operations in a simplified form.

[0078] Embodiment 400 illustrates a method for optimization where constants may replace variables, and a condition optimization may be performed. An example of a condition optimization may be found in embodiment 500 presented later in this specification.

[0079] In block 402, the code to be optimized may be received. The code may include application code and operating system code. In some embodiments, the code may be only operating system code, while other embodiments may process only application code.

[0080] Data may be collected about the deployment in block 404. The data may identify any constants that may be collected and used by the executable code.

[0081] The code may be decompiled in block 406, and each variable may be identified in block 408. Each variable may be analyzed in block 410.

[0082] For each variable in block 410, an attempt may be made in block 412 to determine the value for the variable. If the value is not known in block 414, the process may return to block 410. If the value is known in block 414, and the value is not constant in block 416, the process may return to block 410.

[0083] If the value is known in block 414 and the value is constant in block 416, each instance of the variable may be analyzed in block 418. For each instance in block 418, the variable may be replaced by the value in block 420.

[0084] The process of blocks 408-420 may put constant values in place of the variables. At this point, a condition optimization may be performed in block 422. An example of a condition optimization is illustrated in embodiment 500 presented later in this specification.

[0085] After the optimization may be performed, the updated code may be stored in block 424, recompiled in block 426, and deployed in block 428 to be executed in block 430.

[0086] Figure 5 is a flowchart illustration of an embodiment 500 showing a method for analyzing and optimizing conditions in code. Embodiment 500 illustrates the operations that may be performed in block 422 of embodiment 400.

[0087] Other embodiments may use different sequencing, additional or fewer steps, and different nomenclature or terminology to accomplish similar

functions. In some embodiments, various operations or set of operations may be performed in parallel with other operations, either in a synchronous or asynchronous manner. The steps selected here were chosen to illustrate some principles of operations in a simplified form.

[0088] Embodiment 500 illustrates a method for optimization where conditions may be analyzed and removed when the conditions are known. Embodiment 500 may also perform orphan removal.

[0089] The code may be received in block 502 with variables replaced by constants. The code may be searched to identify a condition in block 504.

[0090] If the condition cannot be evaluated using the known constants in block 506 and another condition exists in the code in block 514, the process may return to block 504.

[0091] If the condition can be determined given the constants in block 506, the condition statement may be removed in block 508 and replaced with a jump to the executable code associated with the condition. If an orphan code is created in block 510, the orphan may be removed in block 512, otherwise the process may return to block 514 to process another condition.

[0092] After the optimization has been performed, the code may be updated in block 516.

[0093] The foregoing description of the subject matter has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the subject matter to the precise form disclosed, and other modifications and variations may be possible in light of the above teachings. The embodiment was chosen and described in order to best explain the principles of the invention and its practical application to thereby enable others skilled in the art to best utilize the invention in various embodiments and various modifications as are suited to the particular use contemplated. It is intended that the appended claims be construed to include other alternative embodiments except insofar as limited by the prior art.

CLAIMS

What is claimed is:

1. A method comprising:

receiving a set of executable code;

decompiling said set of executable code;

determining a first constant value for a first variable in said executable code;

for each occurrence of said first variable, replacing said first variable with said first constant value to create a first modified set of executable code; and

recompiling said first modified set of executable code to create a second set of executable code.

2. The method of claim 1 further comprising:

executing said second set of executable code.

3. The method of claim 1 further comprising:

analyzing said first modified set of executable code to identify conditions that are satisfied when said first variable is defined at said first constant value;

for each of said conditions, removing said conditions from said first modified set of executable code.

4. The method of claim 3 further comprising:

identifying an orphan code in a first condition, said first condition being satisfied when said first variable is defined at said first constant value; and

removing said orphan code from said first modified set of executable code.

5. The method of claim 4, said executable code being application code.

6. The method of claim 5, said decompiling being performed in an executable environment, said executable environment being separate from an operating system.

7. The method of claim 5, said decompiling being performed in an operating system.

8. The method of claim 5, said decompiling being performed in a server, said executable code being transmitted to a client device for execution.

9. The method of claim 8, said second set of modified executable code comprising operating system code.

10. A system comprising:

a processor;

a code analyzer operating on said processor, said code analyzer that:

receives a set of executable code;

decompiles said set of executable code;

determines a first constant value for a first variable in said executable code;

for each occurrence of said first variable, replaces said first variable with said first constant value to create a first modified set of executable code; and

recompiles said first modified set of executable code to create a second set of executable code.

11. The system of claim 10, said code analyzer being comprised in an execution environment.

12. The system of claim 10, said code analyzer being executed after receiving a request for executing said executable code.

13. The system of claim 10, said executable code comprising application executable code.

14. The system of claim 10, said executable code comprising operating system executable code.

15. The system of claim 14, said executable code further comprising application executable code.

16. The system of claim 10, said code analyzer being comprised in an operating system.

17. The system of claim 16, said second set of executable code being executed on a second system having a second processor.

18. A method comprising:

receiving a set of executable code, said executable code comprising application code;

decompiling said set of executable code;

identifying links from said application code to operating system code and incorporating said operating system code into said set of executable code;

determining a first constant value for a first variable in said executable code;

for each occurrence of said first variable, replacing said first variable with said first constant value to create a first modified set of executable code; and

recompiling said first modified set of executable code to create a second set of executable code.

19. The method of claim 18 further comprising:

downloading said second set of executable code to a client device.

20. The method of claim 19, said second set of executable code being the complete set of executable code executing on said client device.

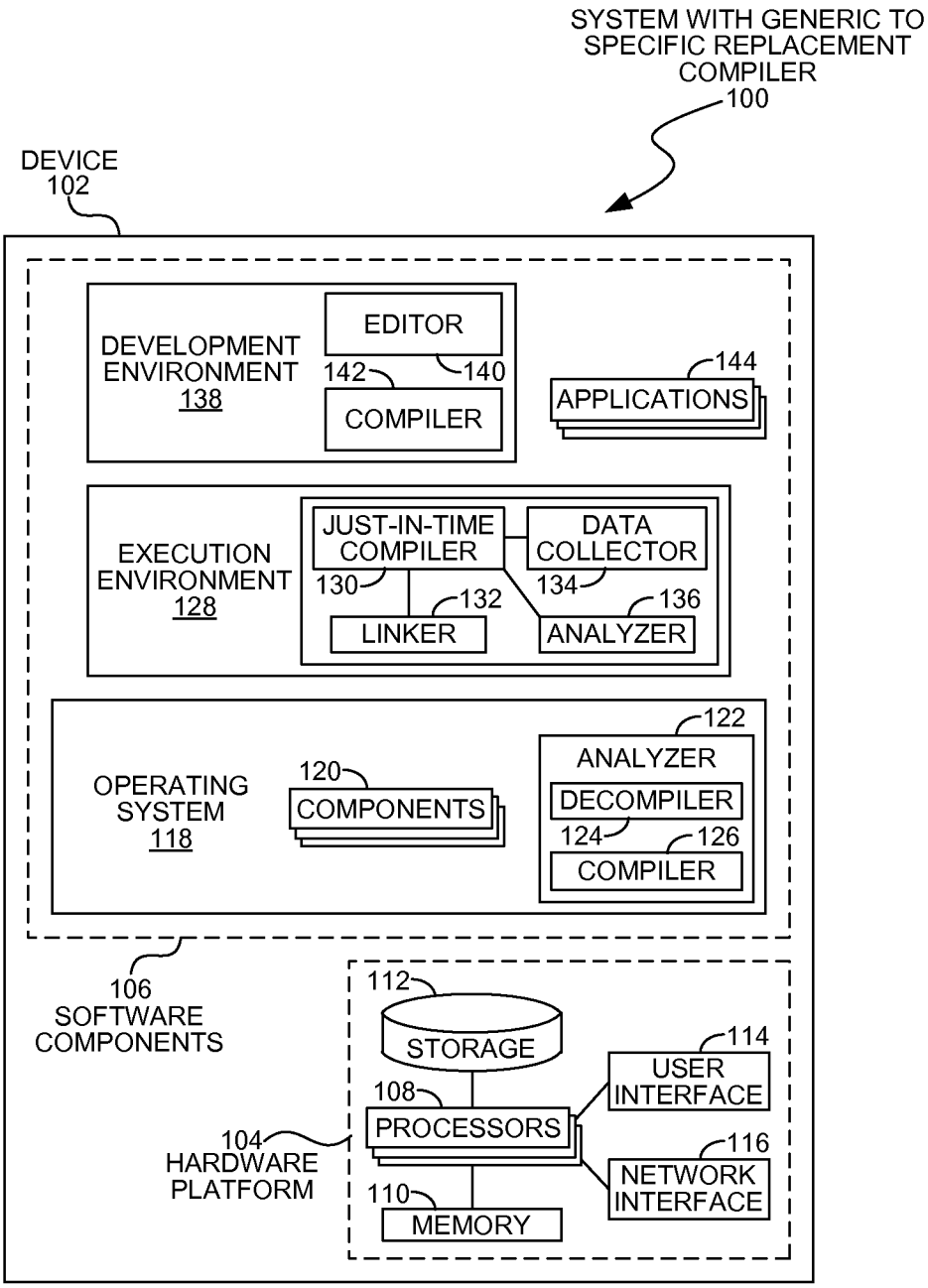
SYSTEM WITH GENERIC TO SPECIFIC REPLACEMENT COMPILER
100
DEVICE
102
DEVELOPMENT ENVIRONMENT
138
EDITOR
142
140
COMPILER
144
APPLICATIONS
EXECUTION ENVIRONMENT
128
JUST-IN-TIME COMPILER
130
DATA COLLECTOR
134
132
LINKER
136
ANALYZER
OPERATING SYSTEM
118
120
COMPONENTS
122
ANALYZER
DECOMPILER
124
126
COMPILER
106
SOFTWARE COMPONENTS
112
STORAGE
114
USER INTERFACE
108
PROCESSORS
104
HARDWARE PLATFORM
116
NETWORK INTERFACE
110
MEMORY

FIG. 1

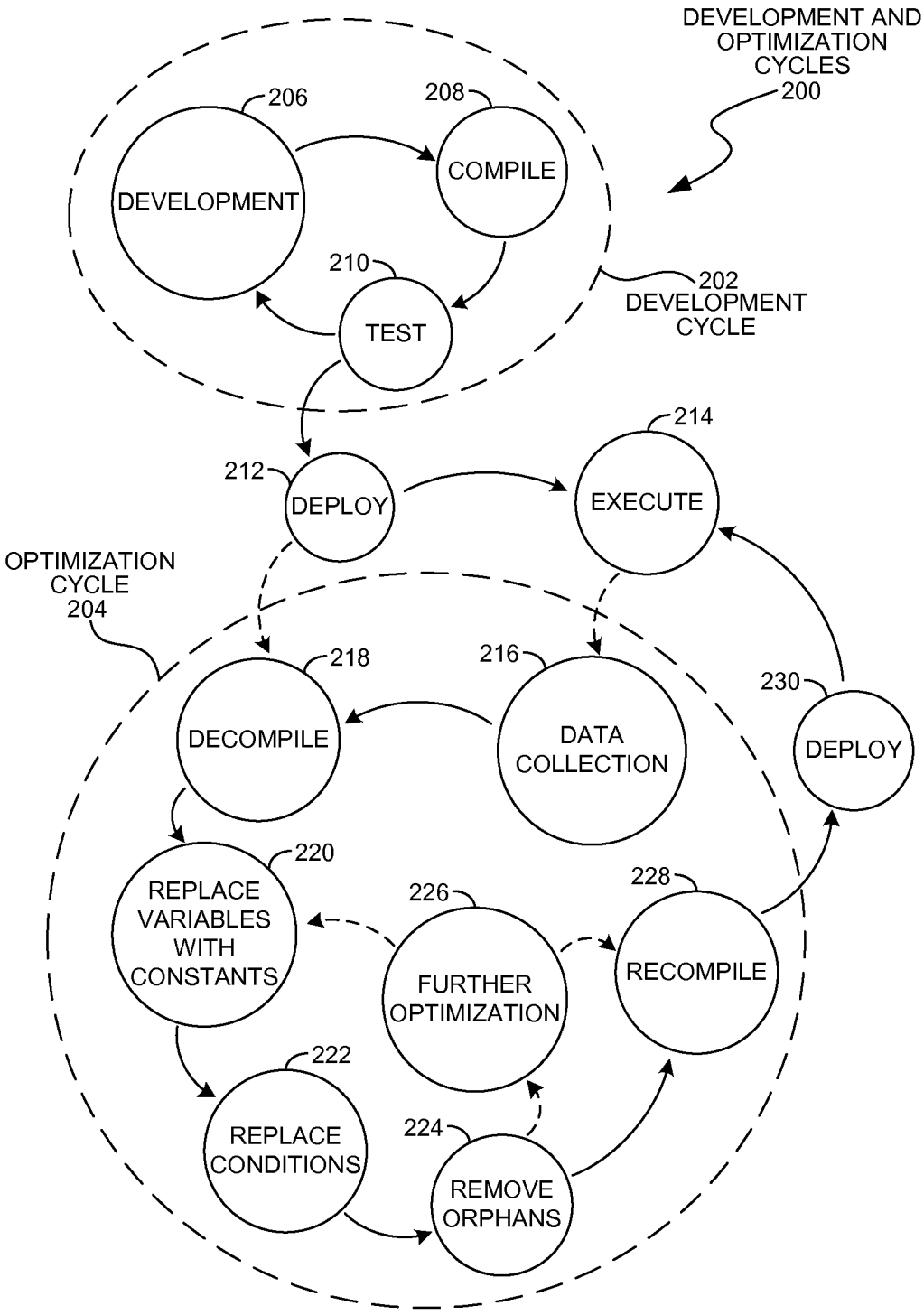
DEVELOPMENT AND OPTIMIZATION CYCLES
200
206
208
DEVELOPMENT
COMPILE
210
TEST
202
DEVELOPMENT CYCLE
212
DEPLOY
214
EXECUTE
OPTIMIZATION CYCLE
204
218
216
230
DECOMPILE
DATA COLLECTION
DEPLOY
220
228
226
REPLACE VARIABLES WITH CONSTANTS
FURTHER OPTIMIZATION
RECOMPILE
222
224
REPLACE CONDITIONS
REMOVE ORPHANS

FIG. 2

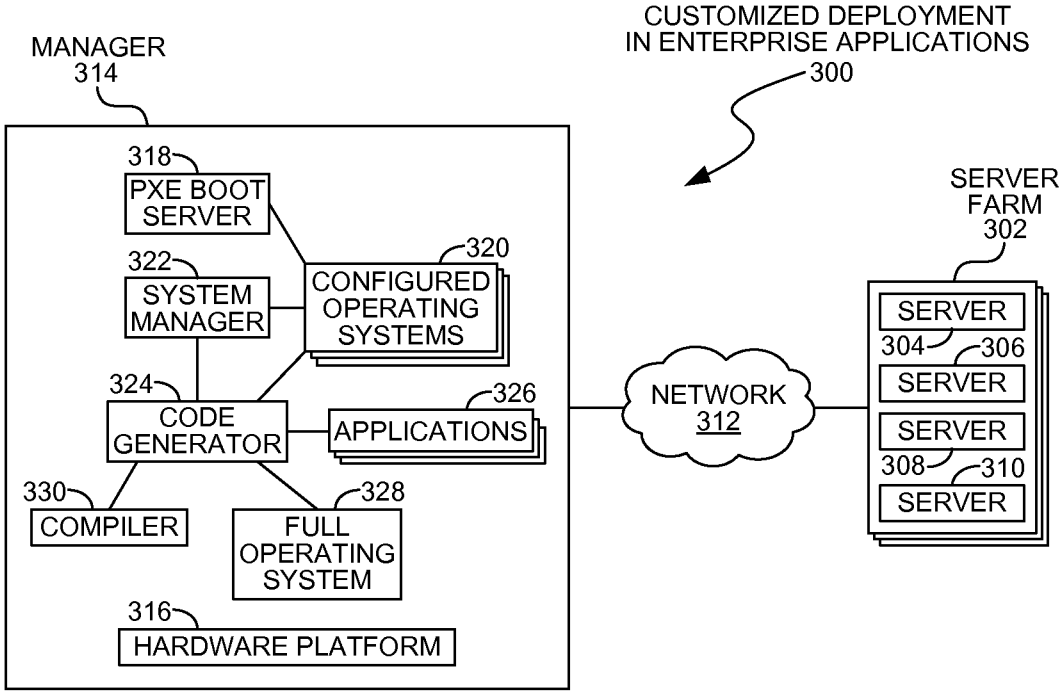
CUSTOMIZED DEPLOYMENT
IN ENTERPRISE APPLICATIONS
300
MANAGER
314
318
PXE BOOT SERVER
322
SYSTEM MANAGER
320
CONFIGURED OPERATING SYSTEMS
324
CODE GENERATOR
326
APPLICATIONS
330
COMPILER
328
FULL OPERATING SYSTEM
316
HARDWARE PLATFORM
NETWORK
312
SERVER FARM
302
SERVER
304
SERVER
306
SERVER
308
SERVER
310

FIG. 3

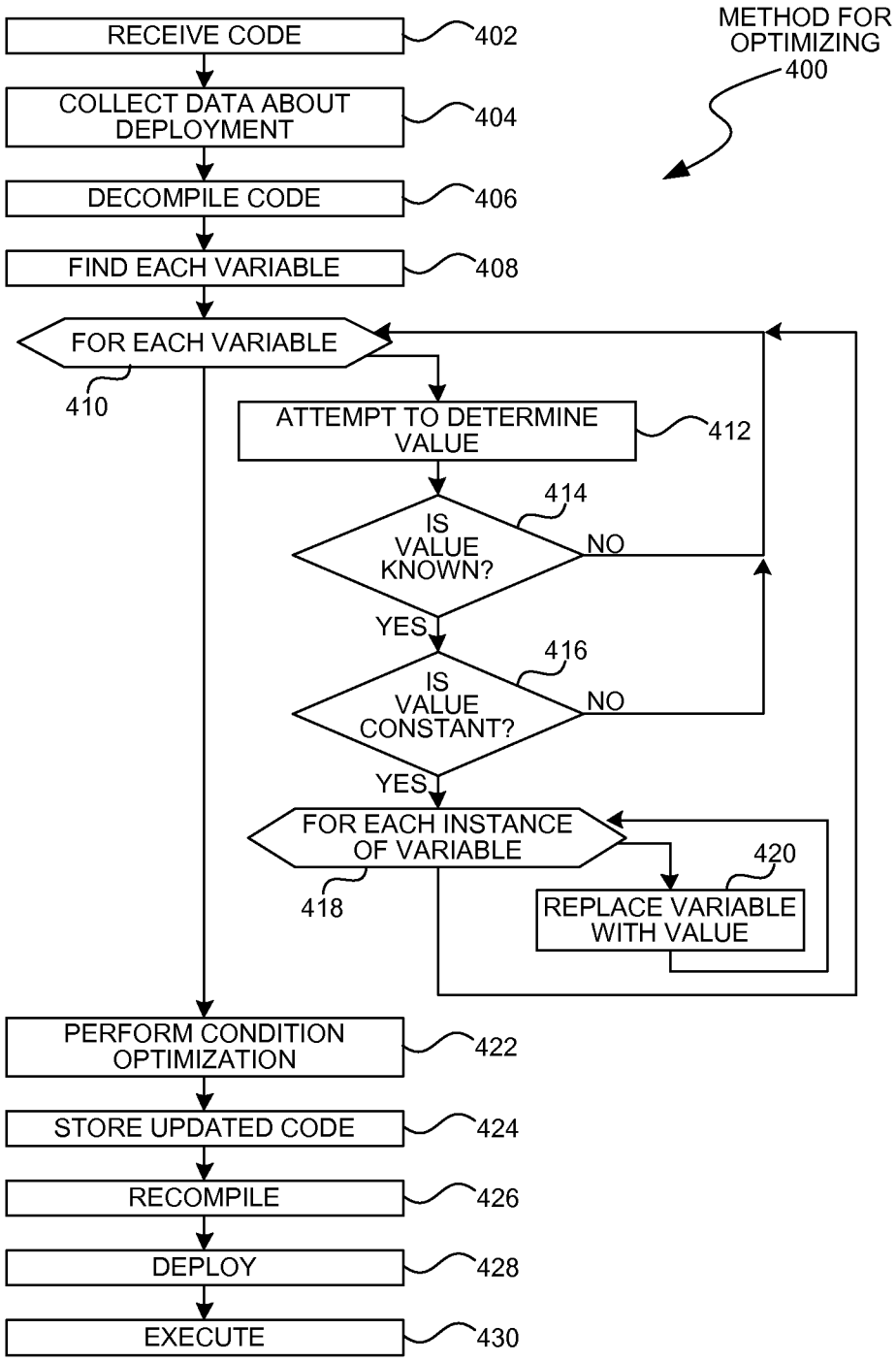
METHOD FOR OPTIMIZING
400
RECEIVE CODE
402
COLLECT DATA ABOUT DEPLOYMENT
404
DECOMPILE CODE
406
FIND EACH VARIABLE
408
FOR EACH VARIABLE
410
ATTEMPT TO DETERMINE VALUE
412
IS VALUE KNOWN?
414
NO
YES
IS VALUE CONSTANT?
416
NO
YES
FOR EACH INSTANCE OF VARIABLE
418
REPLACE VARIABLE WITH VALUE
420
PERFORM CONDITION OPTIMIZATION
422
STORE UPDATED CODE
424
RECOMPILE
426
DEPLOY
428
EXECUTE
430

FIG. 4

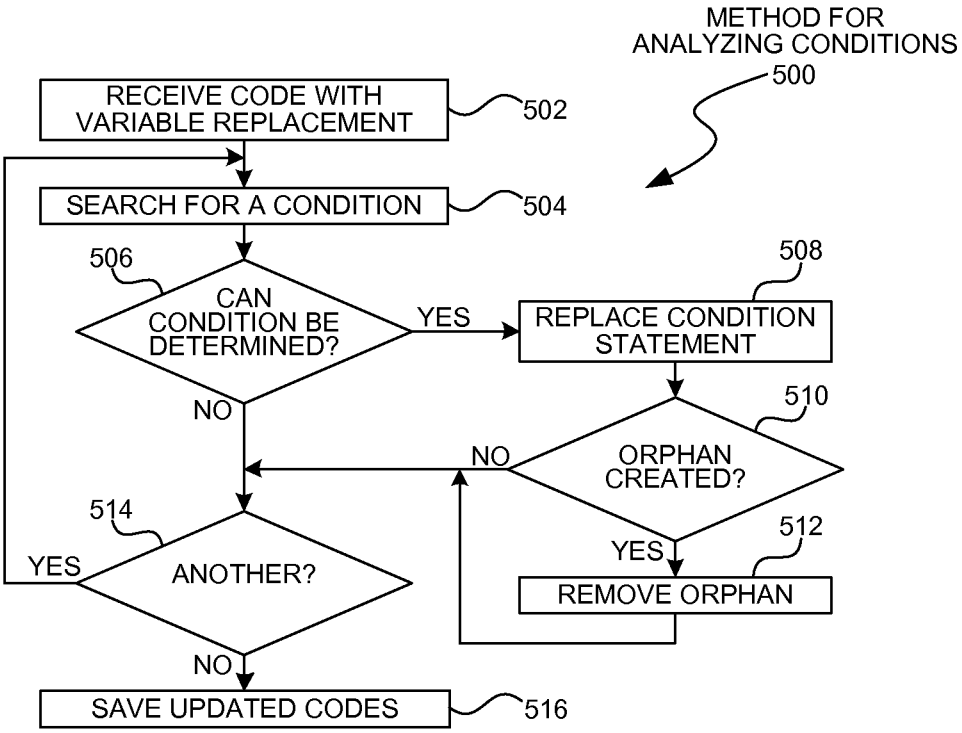
METHOD FOR ANALYZING CONDITIONS
500
RECEIVE CODE WITH VARIABLE REPLACEMENT
502
SEARCH FOR A CONDITION
504
506
CAN CONDITION BE DETERMINED?
YES
508
REPLACE CONDITION STATEMENT
NO
510
ORPHAN CREATED?
NO
514
ANOTHER?
YES
YES
512
REMOVE ORPHAN
NO
SAVE UPDATED CODES
516

FIG. 5

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2012/056711

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/45(2006.01)i, G06F 9/30(2006.01)i

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 9/45; G06F 9/44

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: "optimization, recompile"

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2004-0088690 A1 (HAYIM SHAUL) 06 May 2004 See abstract, paragraphs [0025]-[0026], [0032]-[0048], claim 1-22, and figrur	1-2,10-17
A	es 2-4.	3-9,18-20
Y	US 06151701 A (HUMPHREYS GREG et al.) 21 November 2000	1-2,10-17
A	See abstract, column 5, lines 1-22, column 6, line 22-43, and figures 4-8.	3-9,18-20
A	US 2009-0089771 A1 (GILL BINNY S. et al.) 02 April 2009 See abstract, paragraphs [0039]-[0040], and figures 2-3.	1-20
A	US 2008-0209395 A1 (ERNST BRETT) 28 August 2008 See abstract, paragraphs [0029]-[0043], and figures 3~4.	1-20

☐ 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 application or patent 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 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	Date of mailing of the international search report
22 MARCH 2013 (22.03.2013)	**22 MARCH 2013 (22.03.2013)**

Name and mailing address of the ISA/KR	Authorized officer
Korean Intellectual Property Office 189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701, Republic of Korea	JI, Jeong Hoon
Facsimile No. 82-42-472-7140	Telephone No. 82-42-481-5688

Form PCT/ISA/210 (second sheet) (My 2009)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2012/056711

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US **2004-0088690** A1	**06.05.2004**	**None**	
US **06151701** A	**21.11.2000**	**None**	
US **2009-008977 1** A1	**02.04. 2009**	US **2009-089760** A1	**02. 04. 2009**
US **2008-0209395** A1	**28.08.2008**	**None**	