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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
18 September 2014 (18.09.2014)

WIPO | PCT

(10) International Publication Number
WO 2014/143279 A1

(51) International Patent Classification:
G06F 11/34 (2006.01) *G06F 9/44* (2006.01)

(21) International Application Number:
PCT/US2013/075876

(22) International Filing Date:
17 December 2013 (17.12.2013)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

61/801,298	15 March 2013 (15.03.2013)	US
13/866,014	18 April 2013 (18.04.2013)	US
13/866,020	18 April 2013 (18.04.2013)	US
13/866,022	18 April 2013 (18.04.2013)	US

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

(72) Inventors: **GARRETT, Charles D.**; 17641 167th Ave NE, Woodinville, Washington 98072 (US). **FRASER, Christopher W.**; 2477 Westmont Way West, Seattle, Washington 98199 (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, IR, IS, JP, KE, KG, 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, SA, 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,

[Continued on next page]

(54) Title: BOTTLENECK DETECTOR FOR EXECUTING APPLICATIONS

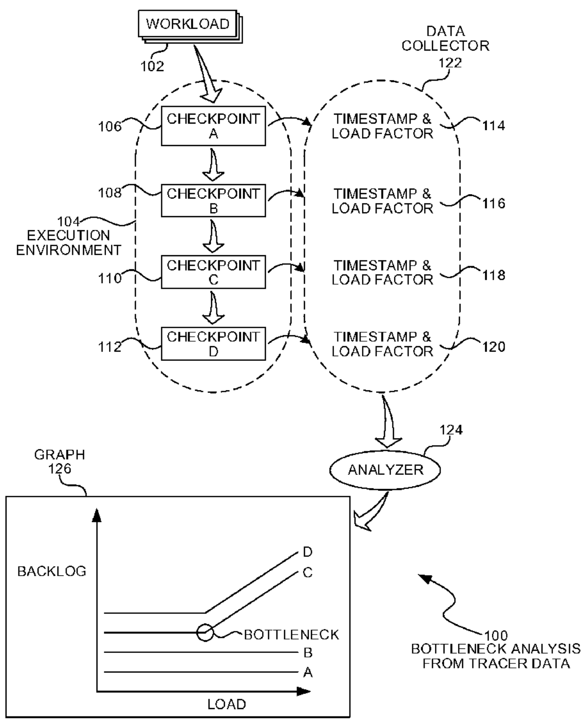

FIG. 1

(57) Abstract: A bottleneck detector may analyze individual workloads processed by an application by logging times when the workload may be processed at different checkpoints in the application. For each checkpoint, a curve fitting algorithm may be applied, and the fitted curves may be compared between different checkpoints to identify bottlenecks or other poorly performing sections of the application. A real time implementation of a detection system may compare newly captured data points against historical curves to detect a shift in the curve, which may indicate a bottleneck. In some cases, the fitted curves from neighboring checkpoints may be compared to identify sections of the application that may be a bottleneck. An automated system may apply one set checkpoints in an application, identify an area for further investigation, and apply a second set of checkpoints in the identified area. Such a system may recursively search for bottlenecks in an executing application.

WO 2014/143279 A1

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, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(iv))*

Published:

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

Bottleneck Detector for Executing Applications

Cross Reference to Related Applications

[0001] This application claims priority to and benefit of United States Patent Application Serial Number 13/866,014 entitled "Bottleneck Detector for Executing Applications" filed 18 Apr 2013, United States Patent Application Serial Number 13/866,020 entitled "Bottleneck Detector Application Programming Interface" filed 18 Apr 2013, and United States Patent Application Serial Number 13/866,022 entitled "Iterative Bottleneck Detector for Executing Applications" filed 18 Apr 2013, the entire contents of which are hereby expressly incorporated by reference for all they disclose and teach.

Background

[0002] Computer applications often have bottlenecks that may limit the throughput or efficiency of an application. Often, bottlenecks may not be fully appreciated when the application code is being written and may only be noticeable when the code may be executed under load.

[0003] The bottlenecks may be an artifact of the application design, poor programming technique, or may be the result of outside constraints on an application. When a bottleneck may be identified, a programmer may be able to investigate the bottleneck and rewrite or otherwise improve the code to increase application performance.

Summary

[0004] A bottleneck detector may analyze individual workloads processed by an application by logging times when the workload may be processed at different checkpoints in the application. For each checkpoint, a curve fitting algorithm may be applied, and the fitted curves may be compared between different checkpoints to identify bottlenecks or other poorly performing sections of the application. A real time implementation of a detection system may compare newly captured data points

against historical curves to detect a shift in the curve, which may indicate a bottleneck. In some cases, the fitted curves from neighboring checkpoints may be compared to identify sections of the application that may be a bottleneck. An automated system may apply one set checkpoints in an application, identify an area for further investigation, and apply a second set of checkpoints in the identified area. Such a system may recursively search for bottlenecks in an executing application.

[0005] An application programming interface may receive workload identifiers and checkpoint identifiers from which bottleneck detection may be performed. Workloads may be tracked through various checkpoints in an application and timestamps collected at each checkpoint. From these data, bottlenecks may be identified in real time or by analyzing the data in a subsequent analysis. The workloads may be processed by multiple devices which may comprise a large application. In some cases, the workloads may be processed by different devices in sequence or in a serial fashion, while in other cases workloads may be processed in parallel by different devices. The application programming interface may be part of a bottleneck detection service which may be sold on a pay-per-use model, a subscription model, or some other payment scheme.

[0006] A bottleneck detector may use an iterative method to identify a bottleneck with specificity. An automated checkpoint inserter may place checkpoints in an application. When a bottleneck is detected in an area of an application, the first set of checkpoints may be removed and a new set of checkpoints may be placed in the area of the bottleneck. The process may iterate until a bottleneck may be identified with enough specificity to aid a developer or administrator of an application. In some cases, the process may identify a specific function or line of code where a bottleneck occurs.

[0007] 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

[0008] In the drawings,

[0009] FIGURE 1 is a diagram illustration of an embodiment showing a method for performing bottleneck analysis.

[0010] FIGURE 2 is a diagram illustration of an embodiment showing a network environment with devices to perform bottleneck analysis.

[0011] FIGURE 3A is a diagram illustration of an example embodiment showing a backlog as a function of load for various checkpoints.

[0012] FIGURE 3B is a diagram illustration of an example embodiment showing a backlog as a function of time for various checkpoints.

[0013] FIGURE 3C is a diagram illustration of an example embodiment showing raw data plot of observations for various checkpoints.

[0014] FIGURE 4 is a flowchart illustration of an embodiment showing a method for collecting data.

[0015] FIGURE 5 is a flowchart illustration of an embodiment showing a method for analyzing data in real time.

[0016] FIGURE 6 is a flowchart illustration of an embodiment showing a method for automated bottleneck detection.

[0017] FIGURE 7 is a flowchart illustration of an embodiment showing a method for iterating to pinpoint a bottleneck.

Detailed Description

Bottleneck Detection Using Timestamps

[0018] Bottlenecks in executing programs may be identified by analyzing the timestamps taken as workloads pass certain checkpoints. A bottleneck may be identified when the time to execute a workload between two checkpoints increases at a greater rate than the load increases.

[0019] A bottleneck detector may capture timestamps as workloads pass checkpoints within an application, then analyze the timestamps to identify bottlenecks. The workloads may be any type of memory object, message, process, thread, or other application element that may be operated upon through a sequence of operations. Throughout the course of the workload, various checkpoints may be placed to track the progress of the workload. For each workload, a sequence of timestamps may be collected for each of the checkpoints that the workload may pass.

[0020] In a lightly loaded application, each workload may typically be executed in approximately the same elapsed time. As the load increases to the point where a bottleneck may occur, workloads may take an increasing amount of time at a bottleneck.

[0021] In a simple analogy, workloads may be visualized as cars travelling on a multilane highway. When there is very little traffic, each car may pass mile posts at approximately the same time from the previous mile post. If there was a bottleneck in the highway, due to construction or an accident, the number of lanes may constrict, forcing the cars to slow down through the bottleneck. In such a case, the time from one mile post to another for each car that may pass through the bottleneck. This effect may be measured and detected as a bottleneck.

[0022] The timestamps may be taken at various locations within an application. In some cases, an application may be decorated with function calls that may transmit a workload identifier and capture a timestamp for each checkpoint. The function calls may be placed throughout an application to track a workload's progress through the application. In such an embodiment, the executable code for an application may be changed to include the checkpoint function calls. Such embodiments may make such changes in source code, intermediate code, binary code, or other phases of an application.

[0023] In some embodiments, an instrumented execution environment may identify checkpoints and set events at each checkpoint to capture timestamp data. In such embodiments, the checkpoints may be created and managed without changing the executable code of an application. Such embodiments may have an identification system for creating and setting the checkpoints, as well as a detection and collection system that may detect the checkpoint and collect related data.

[0024] The application may operate on a single device or across multiple devices. In a single device embodiments, the application may execute on a single hardware platform, which may have multiple processors and various memory and peripheral components. In a single device embodiment, the device may have a clock from which timestamps may be taken.

[0025] In a multiple device application, the application may consist of similar or different software components that may operate on different devices. For example, some applications may operate in a computer cluster, where each device may execute a similar instance of an application to the other devices. In another example, several devices may process workloads in series, where one device may process a workload which may be passed to another device for additional processing. In such embodiments, a synchronized clock may be used to coordinate timestamps that may be gathered from multiple devices.

[0026] A bottleneck detection system may use various time series techniques for capturing, analyzing, and displaying bottleneck information in real time or near-real time. With each data collection event, a set of statistical parameters may be gathered and summed, which may enable other statistical analyses to be performed. The statistical parameters may be lightweight enough to be calculated and updated with minimal computer processing overhead, and a separate analysis routine may analyze the data for bottlenecks in an offline or near-real time basis.

Application Programming Interface for Bottleneck Detection

[0027] An application programming interface may receive workload identifiers and checkpoint identifiers from applications being analyzed for bottlenecks. The application programming interface may receive and store timestamped data. In some embodiments, the application programming interface may analyze and display the data in real time or near-real time. In other embodiments, a detailed analysis may be performed on historical data.

[0028] The application programming interface may operate in several different architectures. In one architecture, a programmer may add function calls within an application, where the function calls may communicate with an application programming interface locally or over a network connection. In another architecture, an execution environment may have alerts or other monitoring functions that may transmit information to the application programming interface when each checkpoint is reached. In some such architectures, the execution environment may be an integrated development environment with code editors, compilers, debugging tools, and other components.

[0029] The application programming interface may operate as a programmatic gateway to accept data in real time, and an accompanying analysis and rendering engines may identify bottlenecks and may generate visualizations of the data. In some cases, the analysis engines may identify bottlenecks automatically and generate an alert or other report. In other cases, the analysis engine may generate graphs or other visualizations that may be displayed as data are received or using a secondary analysis.

[0030] The application programming interface may be one component of a paid service for application developers. The service may be a subscription based service, pay-per-use service, or have some other payment mechanism.

Automatic Bottleneck Detection with Automated Checkpoint Selection

[0031] An automated bottleneck detection system may use a recursive mechanism to isolate and identify a bottleneck in an application. A first set of checkpoints may be used to identify a portion of an application that contains a bottleneck, then a second set of checkpoints may be deployed within the identified portion. From analysis of the second set of checkpoints, the location of a bottleneck may be refined. Such a process may iterate until a bottleneck is defined with a high degree of specificity.

[0032] An automated bottleneck detection system may include an automated mechanism for identifying and placing checkpoints in an application. In some embodiments, such a mechanism may insert function calls or otherwise decorate an application in source code, intermediate code, binary code, or some other form. In some cases, an automated bottleneck detection system may attempt to identify natural breaks or other elements in an application into which to insert checkpoints. In some cases, an automated mechanism for inserting checkpoints may place checkpoints at locations that may not be natural breaks.

[0033] In some cases, an automated bottleneck detection system may use a set of predefined checkpoint function calls that may be inserted automatically or inserted by a programmer. In such embodiments, the automated bottleneck detection system may turn on a first subset of checkpoint function calls, identify the general area of a bottleneck, then turn on a second subset of checkpoint function calls that are nearer to the bottleneck to hone in on the bottleneck location.

[0034] Throughout this specification and claims, the terms “profiler”, “tracer”, and “instrumentation” are used interchangeably. These terms refer to any mechanism that may collect data when an application is executed. In a classic definition, “instrumentation” may refer to stubs, hooks, or other data collection mechanisms that may be inserted into executable code and thereby change the executable code, whereas “profiler” or “tracer” may classically refer to data collection mechanisms that may not change the executable code. The use of any of these terms and their derivatives may implicate or imply the other. For example, data collection using a “tracer” may be performed using non-contact data collection in the classic sense of a “tracer” as well as data collection using the classic definition of “instrumentation” where the executable code may be changed. Similarly, data collected through “instrumentation” may include data collection using non-contact data collection mechanisms.

[0035] Further, data collected through “profiling”, “tracing”, and “instrumentation” may include any type of data that may be collected, including performance related data such as processing times, throughput, performance counters, and the like. The collected data may include function names, parameters passed, memory object names and contents, messages passed, message contents, registry settings, register contents, error flags, interrupts, or any other parameter or other collectable data regarding an application being traced.

[0036] Throughout this specification and claims, the term “execution environment” may be used to refer to any type of supporting software used to execute an application. An example of an execution environment is an operating system. In some illustrations, an “execution environment” may be shown separately from an operating system. This may be to illustrate a virtual machine, such as a process virtual machine, that provides various support functions for an application. In other embodiments, a virtual machine may be a system virtual machine that may include its own internal operating system and may simulate an entire computer system. Throughout this specification and claims, the term “execution environment” includes operating systems and other systems that may or may not have readily identifiable “virtual machines” or other supporting software.

[0037] Throughout this specification, like reference numbers signify the same elements throughout the description of the figures.

[0038] In the specification and claims, references to “a processor” includes multiple processors. In some cases, a process that may be performed by “a processor” may be actually performed by multiple processors on the same device or on different devices. For the purposes of this specification and claims, any reference to “a processor” shall include multiple processors which may be on the same device or different devices, unless expressly specified otherwise.

[0039] 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.

[0040] 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.

[0041] 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.

[0042] 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.

[0043] 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.

[0044] Figure 1 is a diagram of an embodiment 100 showing a bottleneck detection mechanism in its operational parts. Embodiment 100 is merely one example of a bottleneck detection system that may be performed on an application.

[0045] A set of workloads 102 may be executed by an application in an execution environment 104. The workloads may be any units of work that may be traced, tracked, or otherwise monitored on a series of checkpoints. As each workload 102 is executed, the various checkpoints 106, 108, 110, and 112 may capture a respective set of timestamps 114, 116, 118, and 120. The timestamps may include load factors and other data.

[0046] A data collector 122 may gather and transmit the data to an analyzer 124, which may generate a graph 126 or provide other output. In some cases, the analyzer 124 may produce output in real time, while in other cases, the analyzer 124 may process the various data elements after collection has completed.

[0047] The bottleneck detection of embodiment 100 may identify bottlenecks by measuring the difference in timestamps between various checkpoints for each workload. A bottleneck may be identified when the time difference between two checkpoints grows for each successive workload. In such a condition, the downstream process may be processing fewer workloads than may be coming in, causing the bottleneck.

[0048] The workloads may be any unit of work that may be tracked through an application. In some cases, the unit of work may be captured in a data item that may be passed from one portion of executable code to another. In other cases, the unit of work may be a process, transaction, thread, or other executable or data element that may undergo transformations or changes by several portions of an application.

[0049] In some embodiments, the workloads may be passed from one device to another. For example, a high performance computing environment may use message passing to transmit workloads from one device to another. Other system, such as computer cluster arrangements, may use multiple devices to handle workloads in parallel or in series.

[0050] The workloads may or may not be independent of each other. In some embodiments, the workloads may be clearly delineated and independent. Other embodiments may have workloads that may interact with each other or are not fully independent. In many cases, a more reliable bottleneck detection may occur with independent workloads.

[0051] At each checkpoint, a timestamp and workload identifier may be captured. In many embodiments, a load factor and other data may also be gathered. The timestamp may be a ‘wall clock’ time that may reflect the actual time a checkpoint may have been encountered. Such a timestamp may be useful in cases where a workload may be processed by multiple devices.

[0052] In some embodiments, the timestamp may be an elapsed time from some designated start time. For example, some embodiments may start a clock when a workload encounters a first checkpoint, then collect elapsed time from the first checkpoint for each subsequent checkpoint. Other embodiments may determine elapsed time from the preceding checkpoint.

[0053] With each checkpoint and timestamp, a workload identifier may be captured. The workload identifier may be a mechanism to link subsequent checkpoint timestamps to each other. In some cases, a workload identifier may have a natural and meaningful name. In other cases, an arbitrary name may be assigned to workloads, one example of which may be to assign consecutive numbers as workload identifiers.

[0054] A load factor may be collected with the timestamp. The load factor may be any indicator for the ‘busy-ness’ or amount of work attempting to be processed by a system. In some embodiments, the load factor may be collected by a different data collection mechanism and matched to the data collected from the checkpoints by the timestamps or other mechanism. For example, a load factor may be a network traffic metric gathered from a network interface, a processor use metric collected from a

[0055] The load factor may be implied in some embodiments. For example, a load factor may be inferred from the number of workload items being processed at a given time, or by the rate at which work items may be received by the system.

[0056] The analyzer 124 may organize the data by checkpoint and may create a time series representing the time lag between a baseline time and the checkpoint timestamp for each workload. Such a time series may be analyzed to determine when the values grow. In a non-bottleneck steady state, such a time series would be expected to be a flat, straight line. When a bottleneck occurs, the values in such a time series would be expected to grow.

[0057] The growth in the time series values may be linear or non-linear, depending on the application. Some embodiments may monitor the checkpoint data in real time and, using time series analyses, may evaluate the data stream to determine when the data stream has deviated from an expected constant value.

[0058] In many embodiments, such an analysis may take into consider the variance of the data. Some data sets may contain more noise than others, and the correlation coefficient or other metrics of noise may be different for each application. In general, the larger the variance in the data, the greater a deviation may be present before a bottleneck may be identified.

[0059] Figure 2 is a diagram of an embodiment 200 showing components that may perform bottleneck detection. Embodiment 200 contains a device 202 that may be a single device in which bottleneck detection may occur, as well as several devices that may perform bottleneck detection on a larger scale, including monitoring applications that execute over multiple devices.

[0060] A single device architecture may gather tracer data containing timestamps gathered at various checkpoints, analyze the data, and graphically display the data or perform bottleneck detection.

[0061] A multiple device architecture may divide different components of the data gathering, analysis, and management functions over different devices. The multiple device architecture may be one way to deliver an application programming interface that may detect bottlenecks from tracer data.

[0062] The diagram of Figure 2 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 execution environment 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.

[0063] Embodiment 200 illustrates a device 202 that may have a hardware platform 204 and various software components. The device 202 as illustrated represents a conventional computing device, although other embodiments may have different configurations, architectures, or components.

[0064] In many embodiments, the device 202 may be a server computer. In some embodiments, the device 202 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.

[0065] The hardware platform 204 may include a processor 208, random access memory 210, and nonvolatile storage 212. The hardware platform 204 may also include a user interface 214 and network interface 216.

[0066] The random access memory 210 may be storage that contains data objects and executable code that can be quickly accessed by the processors 208. In many embodiments, the random access memory 210 may have a high-speed bus connecting the memory 210 to the processors 208.

[0067] The nonvolatile storage 212 may be storage that persists after the device 202 is shut down. The nonvolatile storage 212 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 212 may be read only or read/write capable. In some embodiments, the nonvolatile storage 212 may be cloud based, network storage, or other storage that may be accessed over a network connection.

[0068] The user interface 214 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.

[0069] The network interface 216 may be any type of connection to another computer. In many embodiments, the network interface 216 may be a wired Ethernet connection. Other embodiments may include wired or wireless connections over various communication protocols.

[0070] The software components 206 may include an operating system 218 on which various software components and services may operate. An operating system may provide an abstraction layer between executing routines and the hardware components 204, and may include various routines and functions that communicate directly with various hardware components.

[0071] An execution environment 220 may execute an application 222 and a tracer 226 may collect timestamps and other information at checkpoints in the application 222. The tracer 226 may store its output as tracer data 228.

[0072] The execution environment 220 may be any mechanism that may cause the application 222 to be executed and include a tracer 226 that may gather data at each checkpoint. In some embodiments, the execution environment 220 may be a virtual machine such as a process virtual machine or system virtual machine that may be instrumented with a tracer 226. In other embodiments, the execution environment 220 may be an integrated development environment that may include a code editor, compiler, debugging tools, and other functionality.

[0073] The tracer 226 may be any mechanism that may collect data at each checkpoint. In some cases, the tracer 226 may include function calls or other code that may be inserted into the executable code as binary code, intermediate code, or source code. In other cases, the tracer 226 may operate without modifying the executable code of the application 222.

[0074] A checkpoint inserter 224 may create checkpoints and cause the tracer 226 to collect data at the various checkpoints. In some cases, the checkpoint inserter 224 may decorate the application 222 with function calls or other tracer-related code. In other cases, the checkpoint inserter 224 may create checkpoints that may be monitored by the tracer 226 to collect the tracer data 228.

[0075] In some embodiments, the checkpoint inserter 224 may be a fully automated application part that may select checkpoints and cause the tracer 226 to execute at each checkpoint. In other embodiments, the checkpoint inserter 224 may have some user interface through which a human programmer may select locations for checkpoints, which may be automatically or manually inserted into an application.

[0076] A data analyzer 230 may receive the tracer data 228 to detect various bottlenecks. In some embodiments, the output of the data analyzer 230 may be transmitted to a rendering engine 232 to display graphical results.

[0077] In a network 234 environment, some embodiments may be deployed over multiple devices.

[0078] A tracer manager device 236 may operate on a hardware platform 238, which may be similar to the hardware platform 204. In some cases, the various hardware platforms may include cloud based execution environments which may or may not have a notion of a computing 'device'.

[0079] A tracer manager 240 may manage the operations of a tracing system over multiple devices, such as applications that may be deployed on a clustered computer configuration or other multiple-device architecture. In such embodiments, the tracer manager 240 may coordinate execution of an application, tracers on each device, as well as load generators and other components. The tracer manager 240 may also control a checkpoint inserter 242 and data analyzer 244.

[0080] In some embodiments, the tracer manager 240 may operate a bottleneck detection operation as a paid service. In such an embodiment, customers may pay for bottleneck detection analysis using a payment manager 245, which may charge on a subscription basis, a pay-per-use basis, or other mechanisms.

[0081] One or more execution platforms 246 may execute the application and may collect checkpoint data. The execution platforms 246 may each have a hardware platform 248 on which an execution environment 250 may run. Each application 252 may be identical instances of the same application or may be different components of a larger application. The tracers 254 may gather trace data when a checkpoint is reached.

[0082] A data collection device 264 may operate on a hardware platform 266 and may contain an application programming interface 268 that may receive data from the tracers 254 and store the tracer data 270 for analysis. The application programming interface 268 may receive data taken at each checkpoint occurrence, then store the data.

[0083] In some cases, the application programming interface 268 may perform some processing of the incoming data. For example, some embodiments may create a timestamp when a data element is received from a tracer 254. In another example, some embodiments may preprocess the incoming data into a format that may be further processed by an analysis engine.

[0084] The application programming interface 268 may be used by the tracer 226 that may operate on an embodiment with a single execution environment, as well as gathering data from multiple tracers 254 on multiple execution environments.

[0085] A load generator device 258 may operate on a hardware platform 260 and may have a load generator 262 application. The load generator 262 may create workloads that may be processed by an application. In some cases, such workloads may be artificial or fictitious workloads that may exercise the application so that bottlenecks may appear in the tracer data.

[0086] Figure 3A is a diagram illustration of an example embodiment 300 showing a graph identifying inflection points as bottlenecks from trace data. Embodiment 300 illustrates a graph showing load on the X axis verses backlog on the Y axis.

[0087] The backlog may indicate the amount of time that a workload took to reach a given checkpoint. Five lines 304, 306, 308, 310, and 312 are shown in the graph, and each line may represent the backlog for a given checkpoint as a function of the load experienced by the system.

[0088] The graph of embodiment 300 may or may not reflect the sequence of workloads processed by an application, but instead may reflect measurements taken at different levels of loading. If the workload was applied in ever increasing amounts, the graph 300 may represent the backlog received over time, but in many cases, data used to generate a graph such as embodiment 300 may be gathered over many cycles of high, medium, and low loads.

[0089] In a normal situation where a checkpoint does not experience a bottleneck, the checkpoint line may be horizontal lines, such as for checkpoints 304 and 306.

[0090] At a certain amount of load, checkpoint line 308 may diverge at the inflection point 314. The inflection point 314 may identify the load at which a bottleneck occurred, as well as identified that the bottleneck occurred between checkpoints 306 and 308. A programmer may be able to spot the inflection point 314 visually and investigate the bottleneck in the code between checkpoints 306 and 308.

[0091] As the load increases, a second inflection point 316 may indicate a second bottleneck that may occur between checkpoints 310 and 312. Again, a programmer may be able to investigate and attempt to address the bottleneck.

[0092] The graph of embodiment 300 may be created by mapping the elapsed time for each workload measured from an initial starting point. When such a measurement or calculation is performed, each checkpoint may be illustrated as a stacked line configuration, where the sequence of workflow may be from the bottom of the graph to the top.

[0093] The result of such a measurement may also yield lines that are parallel to each other. For example, checkpoint 310 remains parallel to checkpoint 308 after the inflection point 314. This indicates that the time between checkpoints 308 and 310 may not have changed even after the bottleneck was incurred. The rise of the checkpoint 310 may reflect the downstream effects of the bottleneck in checkpoint 308.

[0094] In some embodiments, the infection points 314 and 316 may be identified through numerical analysis. Such numerical analysis may attempt to fit a curve to the data points, beginning with a straight line curve, and progressing to more complex curves. When the data may not fit a straight line curve, an analysis may attempt to find an inflection point by fitting two line segments. The correlation coefficient for each curve fitting step may be used as a measure of variance in the data as well as a metric for determining when a fitted curve is a sufficient match.

[0095] The analysis of checkpoint lines may involve comparing the slope of a linear curve fitting analysis, such as linear regression. In such analysis, a positive change in slope from one checkpoint line to a subsequent checkpoint line may indicate a bottleneck.

[0096] Figure 3B is a diagram illustration of an example embodiment 302 showing a graph identifying inflection points as bottlenecks from trace data. Embodiment 302 illustrates a graph showing time on the X axis verses backlog on the

Y axis. Embodiment 300 as described above represents load on the X axis, embodiment 302 illustrates a different graph with time on the X axis.

[0097] The backlog may indicate the amount of time that a workload took to reach a given checkpoint. Six lines 318, 320, 322, 324, 326, and 328 are shown in the graph, and each line may represent the backlog for a given checkpoint as a function of the time.

[0098] The graph of embodiment 302 may illustrate a system's response to increasing and decreasing loads. The amount of load is not shown in the graph, but the effects of load may be illustrated.

[0099] As with graph 300, checkpoints 318 and 320 illustrate checkpoints where no bottlenecks have been experienced. At inflection point 330, checkpoint 322 experienced a bottleneck that continues to build until inflection point 332, where the bottleneck recedes until the bottleneck dissipates. While that is occurring, checkpoint 326 experiences an inflection point 334, which indicates a second bottleneck. In the graph 302, the bottleneck of checkpoint 326 appears to build while the bottleneck of checkpoint 332 recedes.

[00100] Figure 4 is a flowchart illustration of an embodiment 400 showing a method for gathering tracer data. The operations of embodiment 400 may illustrate one method that may be performed by the tracers 226 or 254 of embodiment 200.

[00101] 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.

[00102] A tracer may operate in an execution environment to gather timestamp and other data at each checkpoint. In some cases, the checkpoint may be identified by function calls or other markers in the executable code. In other cases, a checkpoint may be an event, executable code component, or other identifiable element of an executing code.

[00103] An execution environment may receive workloads for an application in block 402. For each workload in block 404, the workload may be executed to a first checkpoint in block 406 and timestamp and other data may be taken in block

408. The workload may be executed to a second checkpoint in block 410 and a second timestamp and other data may be taken in block 412. Similarly, the workload may be executed to a third checkpoint in block 414 and a third timestamp and other data may be taken in block 416. The sequence of execution to a checkpoint and collecting data may continue until the workload has finished being processed.

[00104] The execution environment may process many workloads in the manner of blocks 406 through 416. In many cases, multiple workloads may be progressing through an application at once.

[00105] After collecting all the timestamps and other data, the data may be stored in block 418 and analysis may be performed on the data in block 420.

[00106] Figure 5 is a flowchart illustration of an embodiment 500 showing a method for analyzing tracer data in real time. The operations of embodiment 500 may illustrate one method that may be performed by the data analyzers 230 or 244 of embodiment 200.

[00107] 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.

[00108] Embodiment 500 illustrates one method for analyzing tracer data in real time or near-real time. A timestamp may be received in block 502 and a workload level or other data may be determined in block 504. The timestamp may include a workload identifier and checkpoint identifier. In some embodiments, only the workload identifier and checkpoint identifier may be received, and the timestamp may be determined after receiving the data.

[00109] In the example of embodiment 500, the timestamp may represent a value in the Y axis and the workload level may represent a value in the X axis, which may be used to generate a graph such as may be shown in embodiment 300.

[00110] In the example of embodiment 302 where the X axis represents time, the workload level may not be collected in block 504.

[00111] A set of time series statistics may be updated in block 506. The time series statistics may be any type of statistics from which further analyses may be performed. In a simple example of such statistics, the time series statistics may include the sum of all X values, sum of all Y values, the sum of the square of X values, the sum of the square of Y values, the sum of the product of XY, and the number of samples. From these time series data, linear regression may be performed on the dataset to generate a slope and intercept as well as a correlation coefficient.

[00112] In such an embodiment, an analysis may be performed that compares the slope of adjacent checkpoint datasets. When the slope of a later checkpoint diverges or increases from a previous dataset, a bottleneck may be identified.

[00113] After updating the time series statistics in block 506, the process may loop back to block 502 to process another incoming dataset. Such a loop may be performed relatively quickly, and the remaining blocks 508 and 510 may be performed either offline or in a different thread or process so that the data collection of blocks 502 through 506 may proceed without delay.

[00114] In block 508, new values for a visualization graph may be determined and the visualization may be rendered in block 510. In many cases, the calculation and rendering operations of blocks 508 and 510 may consume a relatively large amount of resources than blocks 502 through 506, thus blocks 508 and 510 may be separated.

[00115] Figure 6 is a flowchart illustration of an embodiment 600 showing a method for detecting bottlenecks from tracer data. The operations of embodiment 600 may illustrate one method that may be performed by the data analyzers 230 or 244 of embodiment 200.

[00116] 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.

[00117] Embodiment 600 illustrates a method that analyzes data gathered in a process such as that of embodiment 400 and detects bottlenecks.

[00118] Historical data may be received in block 602. The data may be analyzed for each checkpoint in block 604. For a given checkpoint, the data for each workload may be analyzed in block 606.

[00119] For each workload, a time difference may be calculated from a previous checkpoint in block 608. The time difference may be calculated from the immediately preceding checkpoint in some embodiments, while other embodiments may calculate the time difference from a start time for the workload.

[00120] In block 610, a workload factor may be identified for the time corresponding to the timestamp of the current workload at the current checkpoint. The workload factor may be used in embodiments where the analysis may be performed on historical data. In an embodiment such as embodiment 302 where the X axis of a graph may be time, a workload factor may not be used.

[00121] After preparing the data in blocks 606 through 610, a curve fitting analysis may be performed. In some cases, the curve fitting may be performed against a load factor, while in other cases, the curve fitting may be performed against time.

[00122] An analysis of the fitted curve may be performed in block 614 for any anomalies. An anomaly may be a very high correlation coefficient in a linear curve fitting attempt, an inflection point in a more complicated curve fitting method, or some other indicator that that data may not be adequately represented by a line. When an anomaly is not detected in block 616, the checkpoint curve may not reveal a bottleneck. When an anomaly is detected in block 616, the location may be labeled as a bottleneck.

[00123] The analysis of blocks 604 through 618 may analyze the data at each checkpoint to attempt to identify a bottleneck. The analysis from blocks 620 through 630 may attempt to identify bottlenecks by comparing two checkpoint data streams to each other.

[00124] For each checkpoint in block 620, the curve of the current checkpoint is compared to the curve of the previous, upstream checkpoint in block 622. The comparison in block 622 may compare the slope of one checkpoint dataset to the slope of a second checkpoint dataset. In such embodiments, a diverging slope may indicate that a later checkpoint contains a bottleneck with respect to the previous checkpoint.

[00125] In other embodiments where the curve fitting is a more complex expression, the comparison may detect whether both checkpoint curves are offset or parallel to each other. Diverging data sets may indicate that the later checkpoint may contain a bottleneck with respect to the earlier checkpoint.

[00126] When the difference between the two curves is not significant in block 624, the current checkpoint may not be considered as a bottleneck in block 626.

[00127] When the difference between the two curves is significant in block 624, the current checkpoint may be considered to have a bottleneck in block 628, and the checkpoint may be labeled as a bottleneck in block 630.

[00128] Figure 7 is a flowchart illustration of an embodiment 700 showing an iterative method for detecting bottlenecks from a running application. The operations of embodiment 700 may illustrate one method that may be performed by the tracer manager 240 of embodiment 200.

[00129] 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.

[00130] Embodiment 700 is an example of an iterative method to identify a bottleneck with a high degree of specificity. Embodiment 700 is an example of a method by which a relatively small number of checkpoints may be spread through an application, and when a bottleneck is detected between two of the checkpoints, another set of checkpoints may be placed in the area of the application between the checkpoints and the process may be repeated.

[00131] The method of embodiment 700 may iterate repeatedly to find a bottleneck with a high degree of specificity. Such specificity may be at the level of a single function call or even a specific line of an application, depending on the embodiment.

[00132] In block 702, an application may be received. The application may be analyzed in block 704 to identify checkpoints, and the checkpoints may be added to the application in block 706.

[00133] In some embodiments, the checkpoints in block 704 may be ‘natural’ locations in an application where a checkpoint may be relevant. Examples of such locations may be at function calls or other points within the application. In other embodiments, the checkpoints may be identified by merely spacing the checkpoints within the application code by a predefined number of instruction lines or some other method.

[00134] The application may begin execution in block 708 and data may start to be collected. In block 710, a load may be applied, which may be an artificial load or a natural load in a production system.

[00135] The checkpoint data may be analyzed to identify a bottleneck in block 712. In some cases, the application may be driven with ever increasing loads until a bottleneck becomes apparent.

[00136] If the bottleneck is identified in block 712 but the bottleneck is not identified with enough specificity in block 714, an additional set of checkpoints may be determined in block 718 and added to the application in block 720. The older checkpoints may be removed or turned off in block 722, and the process may return to block 708 to iterate again.

[00137] The iterations may continue with smaller and smaller spacing between checkpoints until the bottleneck is defined with sufficient specificity in block 714, at which point the iterations may stop and the bottleneck may be identified for the developer in block 716.

[00138] 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 performed on a computer processor, said method comprising:

    identifying a workload processed by said application, said workload having multiple instances;

    capturing timestamps each time a workload instance encounters each of a plurality of checkpoints in said application; and

    for a first checkpoint, curve fitting said timestamps across a plurality of workload instances to generate a first historical curve.

2. The method of claim 1 further comprising:

    for a second checkpoint, curve fitting said timestamps across a plurality of workload instances to generate a second historical curve; and

    comparing said first historical curve to said second historical curve to identify a difference between said first historical curve and said second historical curve, and determining that said difference indicates a bottleneck.

3. The method of claim 2, said first historical curve having a first correlation coefficient, said second historical curve having a second correlation coefficient, and said difference comprising a difference between said first correlation coefficient and said second correlation coefficient.

4. The method of claim 3, said first historical curve and said second historical curve both being linear curve fitting.

5. The method of claim 2, said first historical curve having a first slope and said second historical curve having a second slope, said difference comprising a difference between said first slope and said second slope.

6. The method of claim 2, said first historical curve and said second historical curve being a function of load.

7. The method of claim 1, said workload being a process.

8. The method of claim 1, said workload being a thread.

9. The method of claim 1, said timestamps being relative time difference from a baseline timestamp.

10. The method of claim 9, said baseline timestamp being a starting timestamp for said first workload.

11. The method of claim 9, said baseline timestamp being a previous timestamp of a previous checkpoint.

12. A system comprising:

a processor;

a bottleneck analyzer that:

identifies a workload processed by said application, said workload having multiple instances;

captures timestamps each time a workload instance encounters each of a plurality of checkpoints in said application; and

for a first checkpoint, curve fits said timestamps across a plurality of workload instances to generate a first historical curve.

13. The system of claim 12, said bottleneck analyzer that further:

for a second checkpoint, curve fits said timestamps across a plurality of workload instances to generate a second historical curve; and

compares said first historical curve to said second historical curve to identify a difference between said first historical curve and said second historical curve, and determining that said difference indicates a bottleneck.

14. The system of claim 13, said first historical curve having a first correlation coefficient, said second historical curve having a second correlation coefficient, and said difference comprising a difference between said first correlation coefficient and said second correlation coefficient.

15. The system of claim 14, said first historical curve and said second historical curve both being linear curve fitting.

16. The system of claim 13, said first historical curve having a first slope and said second historical curve having a second slope, said difference comprising a difference between said first slope and said second slope.

17. The system of claim 13, said first historical curve and said second historical curve being a function of load.

18. The system of claim 12, said workload being a process.

19. The system of claim 12, said workload being a thread.

20. The system of claim 12, said timestamps being relative time difference from a baseline timestamp.

21. The system of claim 20, said baseline timestamp being a starting timestamp for said first workload.

22. The system of claim 20, said baseline timestamp being a previous timestamp of a previous checkpoint.

Bottleneck Detector Application Programming Interface

23. A system comprising:

a processor;

a tracer operating on said processor, said tracer that:

listens for a start workload call on an interface and logs said start workload call;

listens for a checkpoint call, determines a workload for said checkpoint call, and logs said checkpoint call with a timestamp and a checkpoint identifier;

a data analyzer that:

creates a first curve representing said checkpoint calls for a first checkpoint; and

identifies an abnormality from said first curve, said abnormality being identified from at least one of said checkpoint calls that deviates from said first curve.

24. The system of claim 23, said tracer that further:

returns a workload identifier in response to said start workload call.

25. The system of claim 24, said tracer that further:

receives a workload identifier with said checkpoint call.

26. The system of claim 25, said timestamp being transmitted with said checkpoint call.

27. The system of claim 25, said timestamp being determined when said checkpoint call is received.

28. The system of claim 24, said tracer that further:

creates a second curve representing said checkpoint calls for a second checkpoint; and

identifies said abnormality by comparing said first curve to said second curve, said at least one of said checkpoint calls being contained in said second curve.

29. The system of claim 28, said first curve having a different slope from said second curve.

30. The system of claim 28, said first curve having a different correlation coefficient than said second curve.

31. The system of claim 23, said first curve being a linear curve.

32. The system of claim 23, said first curve being a polynomial curve.

33. The system of claim 23, said timestamp being an incremental time from a beginning time.

34. The system of claim 33, said beginning time being defined at said start workload call.

35. The system of claim 34, said beginning time being a timestamp transmitted by a calling routine.

36. The system of claim 35, said beginning time being determined by said data listener.

37. The system of claim 33, said beginning time being a time defined by a previous checkpoint call.

38. The system of claim 23, said first curve correlating said timestamp and a load factor.

39. The system of claim 38, said load factor being received in said workload call.

40. The system of claim 38, said load factor being received in said checkpoint call.

41. The system of claim 38, said load factor being received from an external source.

42. The system of claim 41, said load factor being received as a data stream comprising timestamps.

Iterative Bottleneck Detector for Executing Applications

43. A method performed on a computer processor, said method comprising:

identifying a first set of checkpoints in an application;

for each of said first set of checkpoints, establishing a checkpoint function call that causes a checkpoint identifier and a timestamp to be captured;

executing said application and capturing said checkpoint identifiers, said timestamps, and workload identifiers for each of said first set of checkpoints;

identifying a first bottleneck between a first checkpoint and a second checkpoint;

identifying a second set of checkpoints between said first checkpoint and said second checkpoint;

for each of said second set of checkpoints, establishing one of said checkpoint calls;

executing said application and capturing said checkpoint identifiers, said timestamps, and said workload identifiers for each of said second set of checkpoints; and

identifying a second bottleneck between a third checkpoint and a fourth checkpoint, said third checkpoint and said fourth checkpoint being from said second set of checkpoints.

44. The method of claim 43, said first checkpoint and said second checkpoint being consecutive checkpoints.

45. The method of claim 43 further comprising:

scanning said application to identify said first set of checkpoints.

46. The method of claim 45, said first set of checkpoints being selected based on proximity to function calls.

47. The method of claim 46, said function calls comprising library function calls.

48. The method of claim 46, said first set of checkpoints being identified by analyzing source code for said application.

49. The method of claim 48, said first set of checkpoints being identified using a label comprising at least a portion of a function name.

50. The method of claim 49, said second set of checkpoints being identified using a label comprising at least a portion of a function name.

51. The method of claim 46, said first set of checkpoints being identified by analyzing intermediate code for said application.

52. The method of claim 43 further comprising:

executing a load generator while executing said application.

53. The method of claim 52 further comprising:

determining a load factor for a plurality of said timestamps.

54. The method of claim 53 further comprising:

correlating said load factor to said timestamps for said checkpoint identifiers.

55. The method of claim 43, said first set of checkpoints being manually inserted in said application.

56. The method of claim 55, said second set of checkpoints being manually inserted in said application.

57. The method of claim 56, said first set of checkpoints being selected from a plurality of checkpoints manually inserted in said application.

58. A system comprising:

a processor;

a checkpoint inserter that:

analyzes an application to identify a plurality of locations for checkpoints; and

inserts a checkpoint function call that causes a checkpoint identifier and a timestamp to be captured;

an analyzer operating on said processor, said analyzer that:

identifies a first set of checkpoints;

executes said application and capturing said checkpoint identifiers, said timestamps, and workload identifiers for each of said first set of checkpoints;

identifies a first bottleneck between a first checkpoint and a second checkpoint;

identifies a second set of checkpoints between said first checkpoint and said second checkpoint;

executes said application and capturing said checkpoint identifiers, said timestamps, and said workload identifiers for each of said second set of checkpoints; and

identifies a second bottleneck between a third checkpoint and a fourth checkpoint, said third checkpoint and said fourth checkpoint being from said second set of checkpoints.

59. The system of claim 58, said checkpoint inserter that scans source code of said application to identify said locations for checkpoints.

60. The system of claim 58, said checkpoint inserter that scans intermediate code of said application to identify said locations for checkpoints.

61. The system of claim 58, said checkpoint function being a call to an application programming interface for said analyzer.

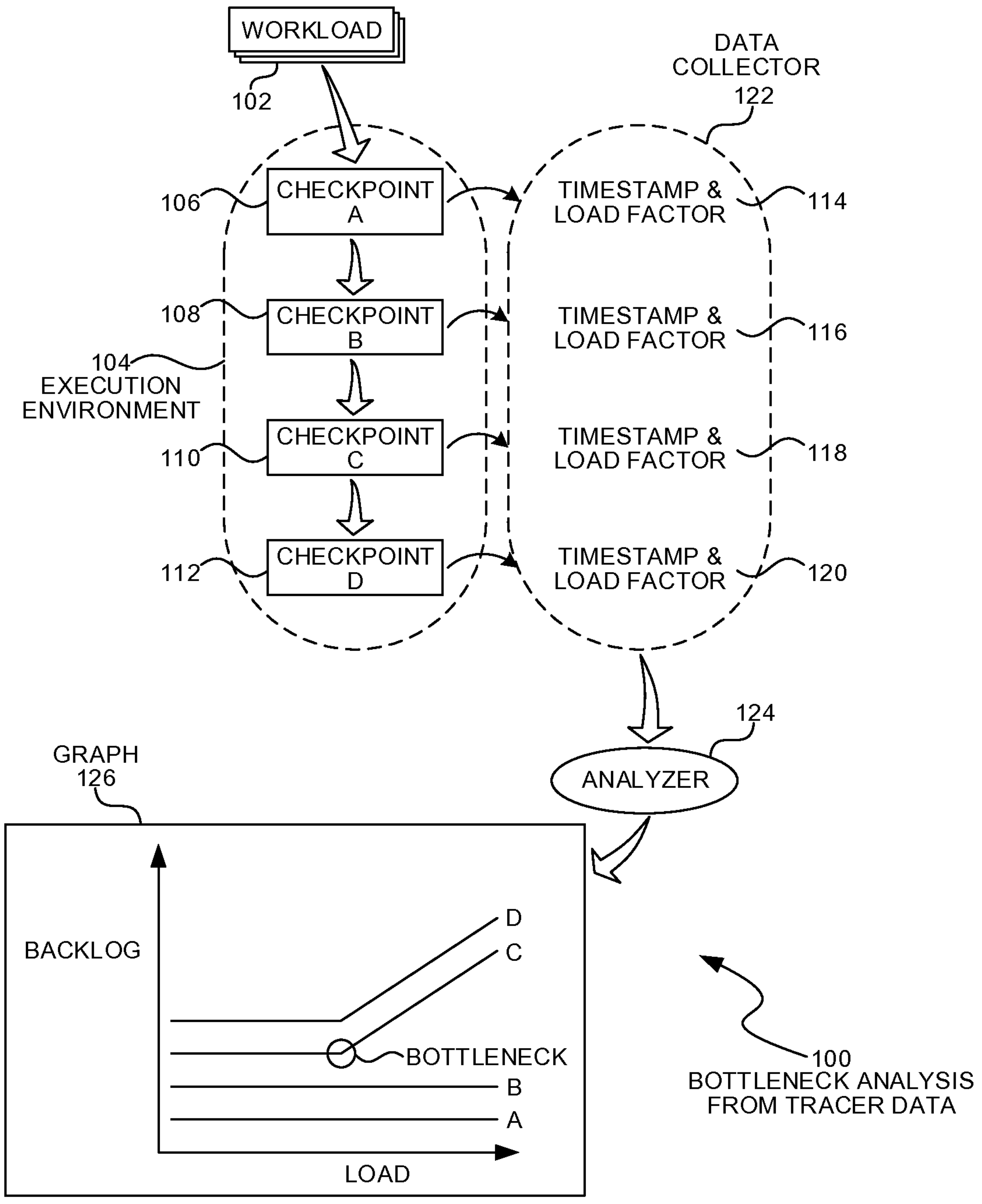
WORKLOAD
102
DATA COLLECTOR
122
106
CHECKPOINT A
TIMESTAMP & LOAD FACTOR
114
108
CHECKPOINT B
TIMESTAMP & LOAD FACTOR
116
104
EXECUTION ENVIRONMENT
110
CHECKPOINT C
TIMESTAMP & LOAD FACTOR
118
112
CHECKPOINT D
TIMESTAMP & LOAD FACTOR
120
124
ANALYZER
GRAPH
126
BACKLOG
D
C
BOTTLENECK
B
A
LOAD
100
BOTTLENECK ANALYSIS FROM TRACER DATA

FIG. 1

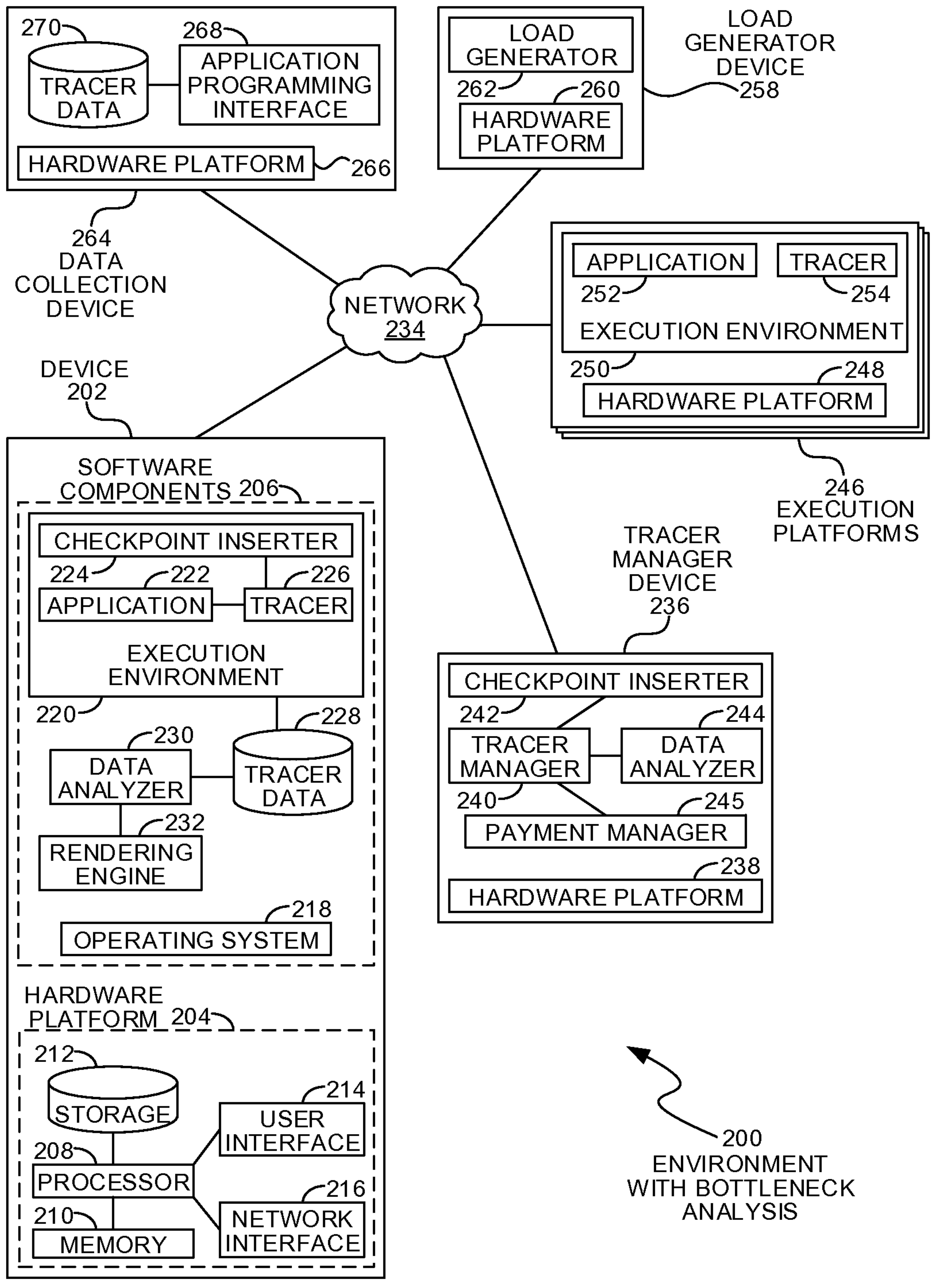
270
TRACER DATA
268
APPLICATION PROGRAMMING INTERFACE
HARDWARE PLATFORM
266
264
DATA COLLECTION DEVICE
LOAD GENERATOR
262
260
HARDWARE PLATFORM
LOAD GENERATOR DEVICE
258
NETWORK
234
APPLICATION
252
TRACER
254
EXECUTION ENVIRONMENT
250
248
HARDWARE PLATFORM
246
EXECUTION PLATFORMS
DEVICE
202
SOFTWARE COMPONENTS 206
CHECKPOINT INSERTER
224
222
226
APPLICATION
TRACER
EXECUTION ENVIRONMENT
220
228
230
DATA ANALYZER
TRACER DATA
232
RENDERING ENGINE
218
OPERATING SYSTEM
HARDWARE PLATFORM 204
212
STORAGE
214
USER INTERFACE
208
PROCESSOR
210
MEMORY
216
NETWORK INTERFACE
TRACER MANAGER DEVICE
236
CHECKPOINT INSERTER
242
244
TRACER MANAGER
DATA ANALYZER
240
245
PAYMENT MANAGER
238
HARDWARE PLATFORM
200
ENVIRONMENT WITH BOTTLENECK ANALYSIS

FIG. 2

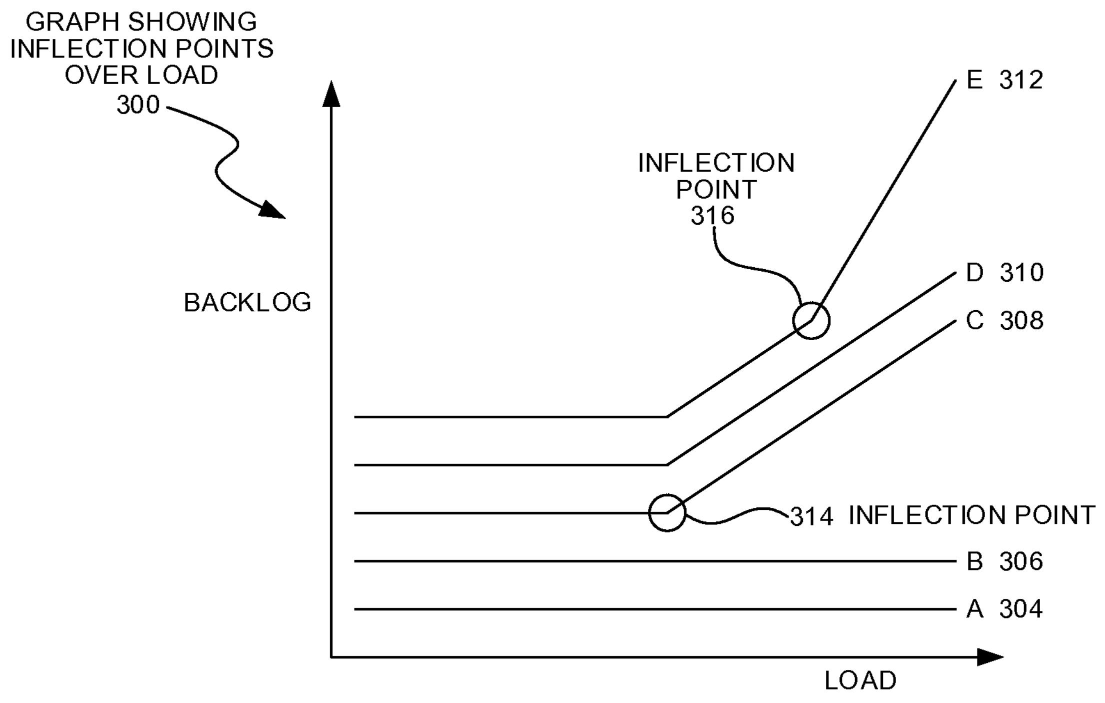
GRAPH SHOWING INFLECTION POINTS OVER LOAD 300
BACKLOG
LOAD
E 312
INFLECTION POINT 316
D 310
C 308
314 INFLECTION POINT
B 306
A 304

FIG. 3A

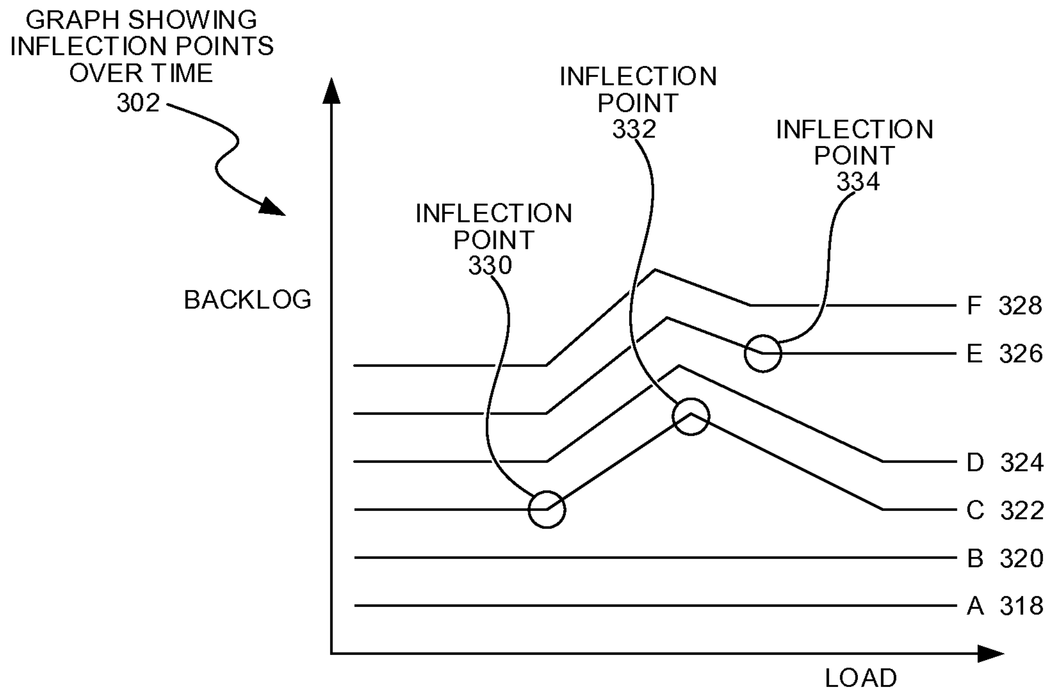
GRAPH SHOWING INFLECTION POINTS OVER TIME 302
BACKLOG
LOAD
INFLECTION POINT 330
INFLECTION POINT 332
INFLECTION POINT 334
F 328
E 326
D 324
C 322
B 320
A 318

FIG. 3B

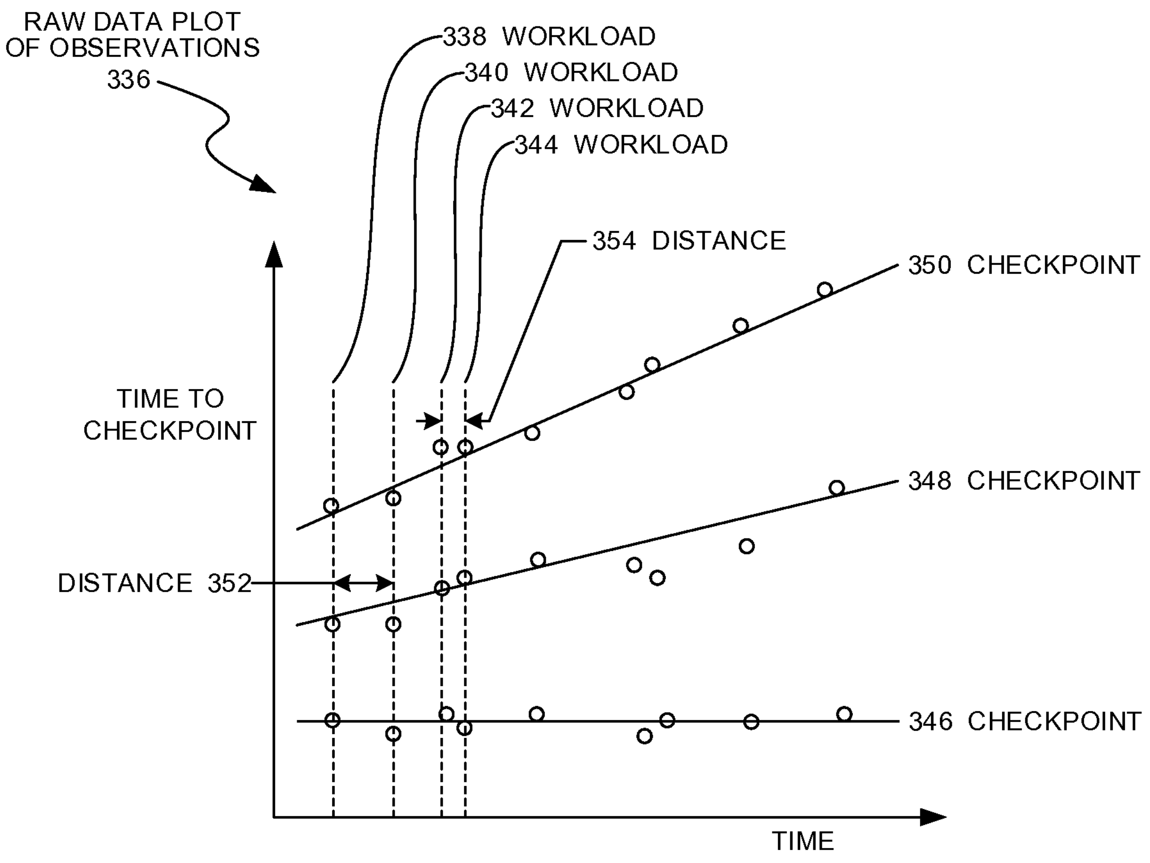
RAW DATA PLOT OF OBSERVATIONS 336
338 WORKLOAD
340 WORKLOAD
342 WORKLOAD
344 WORKLOAD
354 DISTANCE
350 CHECKPOINT
TIME TO CHECKPOINT
348 CHECKPOINT
DISTANCE 352
346 CHECKPOINT
TIME

FIG. 3C

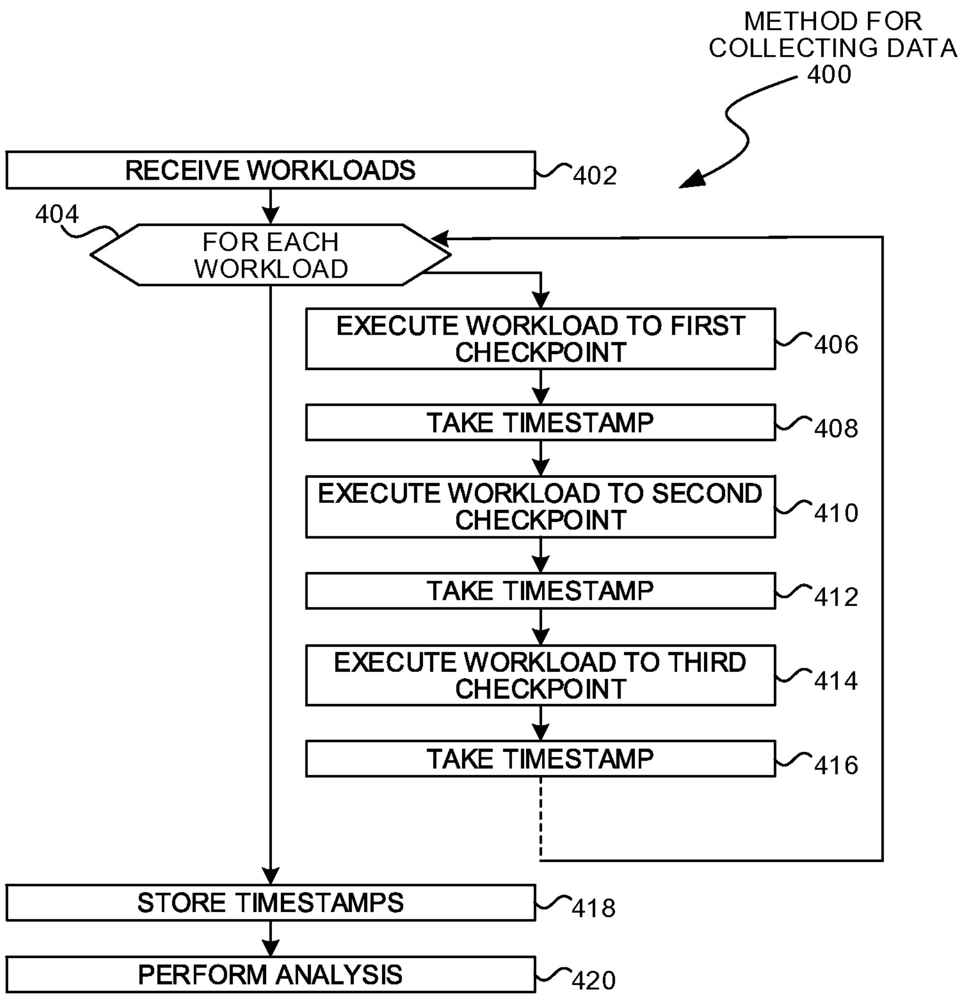
METHOD FOR COLLECTING DATA
400
RECEIVE WORKLOADS
402
404
FOR EACH WORKLOAD
EXECUTE WORKLOAD TO FIRST CHECKPOINT
406
TAKE TIMESTAMP
408
EXECUTE WORKLOAD TO SECOND CHECKPOINT
410
TAKE TIMESTAMP
412
EXECUTE WORKLOAD TO THIRD CHECKPOINT
414
TAKE TIMESTAMP
416
STORE TIMESTAMPS
418
PERFORM ANALYSIS
420

FIG. 4

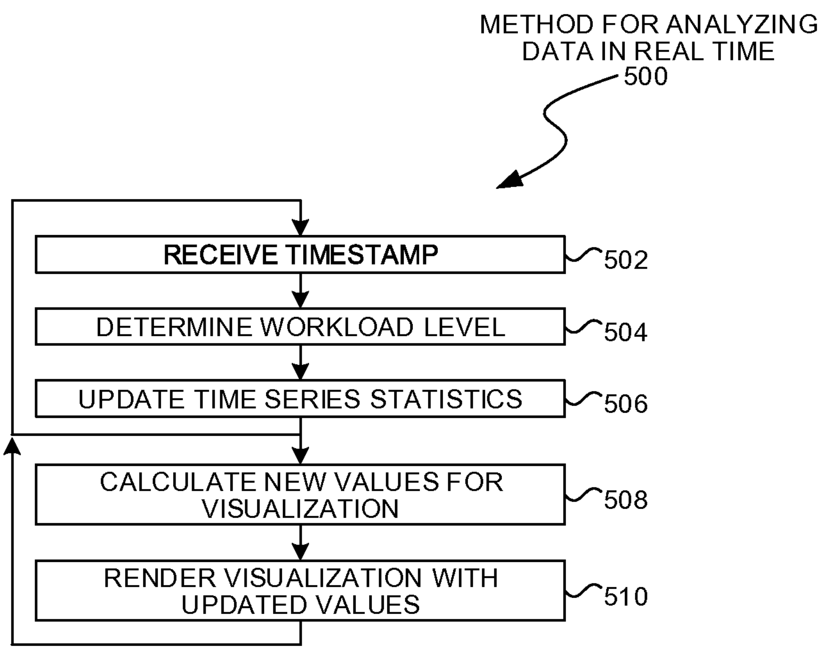
METHOD FOR ANALYZING DATA IN REAL TIME
500
RECEIVE TIMESTAMP
502
DETERMINE WORKLOAD LEVEL
504
UPDATE TIME SERIES STATISTICS
506
CALCULATE NEW VALUES FOR VISUALIZATION
508
RENDER VISUALIZATION WITH UPDATED VALUES
510

FIG. 5

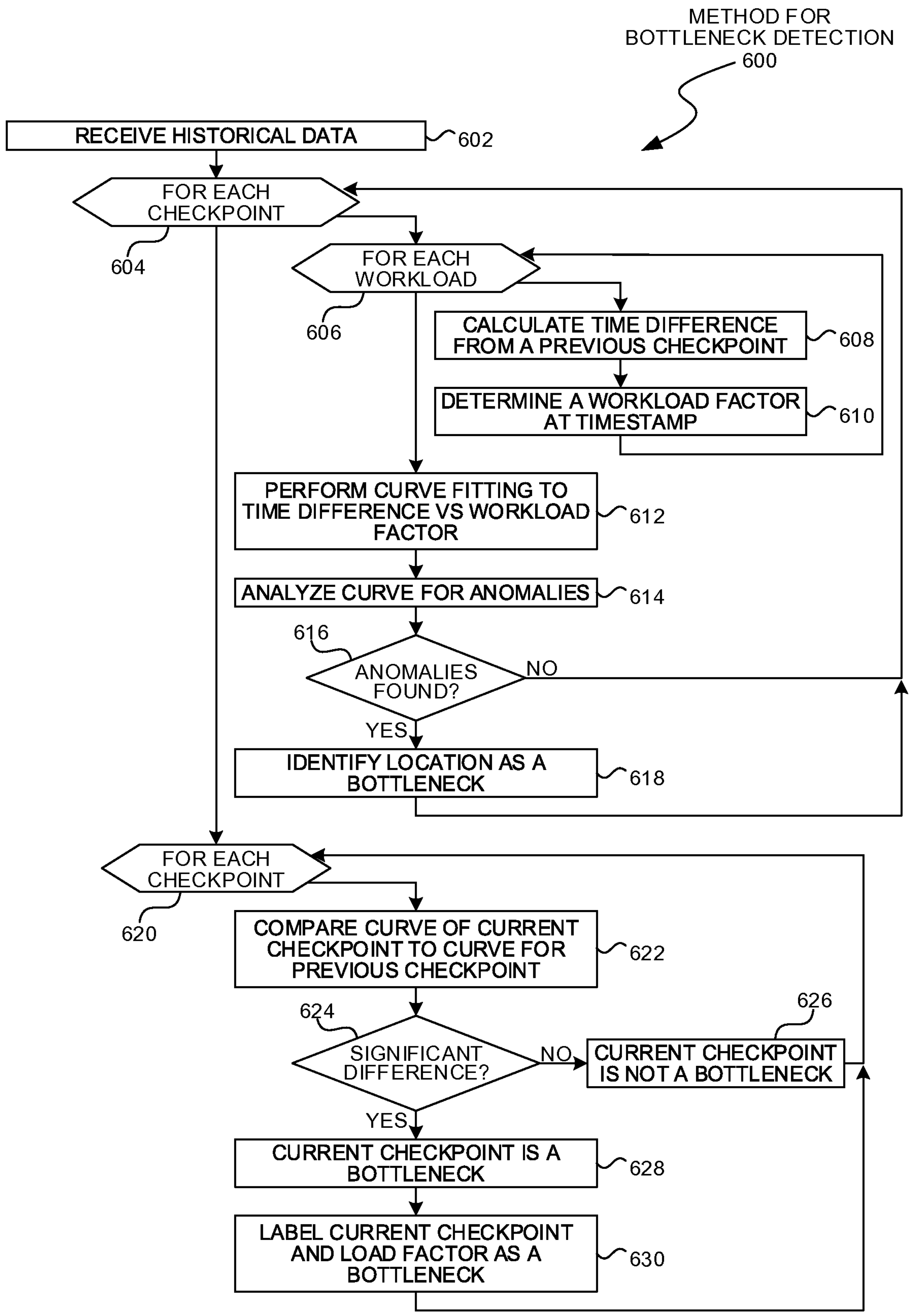
METHOD FOR BOTTLENECK DETECTION
600
RECEIVE HISTORICAL DATA
602
FOR EACH CHECKPOINT
604
FOR EACH WORKLOAD
606
CALCULATE TIME DIFFERENCE FROM A PREVIOUS CHECKPOINT
608
DETERMINE A WORKLOAD FACTOR AT TIMESTAMP
610
PERFORM CURVE FITTING TO TIME DIFFERENCE VS WORKLOAD FACTOR
612
ANALYZE CURVE FOR ANOMALIES
614
616
ANOMALIES FOUND?
NO
YES
IDENTIFY LOCATION AS A BOTTLENECK
618
FOR EACH CHECKPOINT
620
COMPARE CURVE OF CURRENT CHECKPOINT TO CURVE FOR PREVIOUS CHECKPOINT
622
624
SIGNIFICANT DIFFERENCE?
NO
CURRENT CHECKPOINT IS NOT A BOTTLENECK
626
YES
CURRENT CHECKPOINT IS A BOTTLENECK
628
LABEL CURRENT CHECKPOINT AND LOAD FACTOR AS A BOTTLENECK
630

FIG. 6

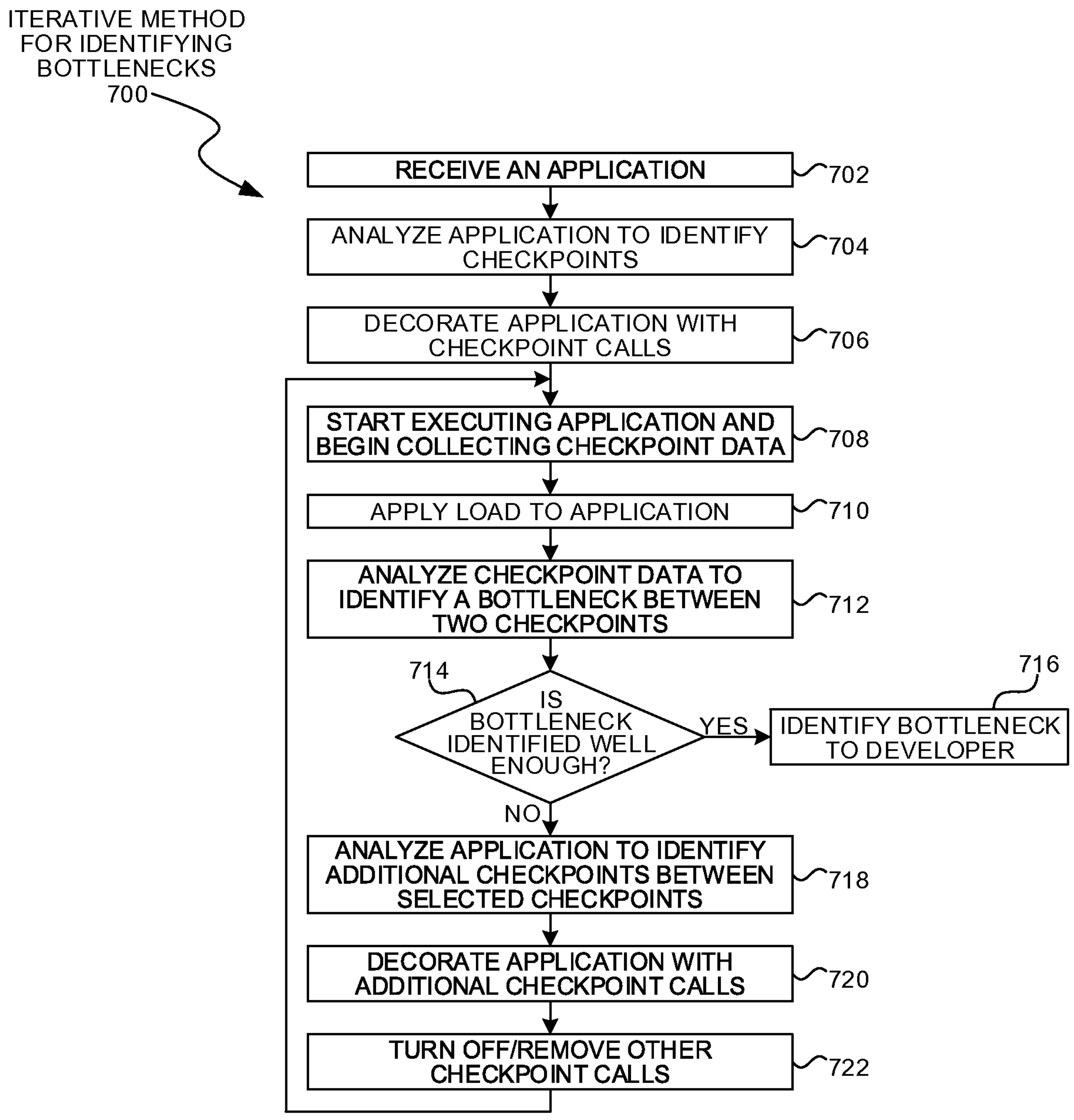
ITERATIVE METHOD FOR IDENTIFYING BOTTLENECKS
700
RECEIVE AN APPLICATION
702
ANALYZE APPLICATION TO IDENTIFY CHECKPOINTS
704
DECORATE APPLICATION WITH CHECKPOINT CALLS
706
START EXECUTING APPLICATION AND BEGIN COLLECTING CHECKPOINT DATA
708
APPLY LOAD TO APPLICATION
710
ANALYZE CHECKPOINT DATA TO IDENTIFY A BOTTLENECK BETWEEN TWO CHECKPOINTS
712
714
IS BOTTLENECK IDENTIFIED WELL ENOUGH?
YES
716
IDENTIFY BOTTLENECK TO DEVELOPER
NO
ANALYZE APPLICATION TO IDENTIFY ADDITIONAL CHECKPOINTS BETWEEN SELECTED CHECKPOINTS
718
DECORATE APPLICATION WITH ADDITIONAL CHECKPOINT CALLS
720
TURN OFF/REMOVE OTHER CHECKPOINT CALLS
722

FIG. 7

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2013/075876

A. CLASSIFICATION OF SUBJECT MATTER

G06F 11/34(2006.01)i, G06F 9/44(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 11/34; G06F 11/30; G06F 9/30; G06F 15/00; G06F 15/173; G06F 9/46; G06F 9/455; 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: bottleneck detector, workload, checkpoint, curve fitting, abnormality, analyzer, tracer, inserter

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2008-0022285 A1 (LUDMILA CHERKASOVA et al.) 24 January 2008 See paragraphs [0003]-[0008], [0018]-[0027], [0051], [0068]-[0095]; and figures 1-2, 5-8.	1-61
A	US 6970805 B1 (MICHAEL J. BIERMA et al.) 29 November 2005 See column 2, lines 3-61; column 3, line 13 - column 5, line 20; claim 1; and figures 1A-2C.	1-61
A	US 2012-0278594 A1 (PRATHIBA KUMAR et al.) 01 November 2012 See paragraphs [0001]-[0004], [0034]-[0040], [0044]; and figures 3-4.	1-61
A	US 2010-0268816 A1 (TOSHIAKI TARUI et al.) 21 October 2010 See paragraphs [0020]-[0030], [0113]-[0136], [0165]-[0176]; and figures 1, 8.	1-61
A	US 2012-0233310 A1 (SANDIP AGARWALA et al.) 13 September 2012 See paragraphs [0004]-[0008], [0019]-[0021], [0059]-[0068]; and figures 5-6.	1-61

☐ 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	"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
"E" earlier application or patent but published on or after the international filing date	"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
"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)	"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
"O" document referring to an oral disclosure, use, exhibition or other means	"&" document member of the same patent family
"P" document published prior to the international filing date but later than the priority date claimed	

Date of the actual completion of the international search 04 April 2014 (04.04.2014)	Date of mailing of the international search report **07 April 2014 (07.04.2014)**
Name and mailing address of the ISA/KR International Application Division Korean Intellectual Property Office 189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701, Republic of Korea Facsimile No. +82-42-472-7140	Authorized officer BYUN, Sung Cheal Telephone No. +82-42-481-8262

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/US2013/075876

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008-0022285 A1	24/01/2008	US 8112756 B2	07/02/2012
US 6970805 B1	29/11/2005	None	
US 2012-0278594 A1	01/11/2012	None	
US 2010-0268816 A1	21/10/2010	JP 2010-250689 A	04/11/2010
US 2012-0233310 A1	13/09/2012	None	