US 20130219372A1

(19) United States

(12) Patent Application Publication

(10) Pub. No.: US 2013/0219372 A1

Li et al.

(43) Pub. Date: Aug. 22, 2013

(54) RUNTIME SETTINGS DERIVED FROM RELATIONSHIPS IDENTIFIED IN TRACER DATA

(71) Applicant: **Concurix Corporation**, (US)

(72) Inventors: **Ying Li**, Bellevue, WA (US); **Alexander G. Gounares**, Kirkland, WA (US); **Charles D. Garrett**, Woodinville, WA (US); **Russell S. Krajec**, Loveland, CO (US)

(73) Assignee: **Concurix Corporation**, Kirkland, WA (US)

(21) Appl. No.: **13/853,809**

(22) Filed: **Mar. 29, 2013**

Related U.S. Application Data

(60) Provisional application No. 61/801,298, filed on Mar. 15, 2013.

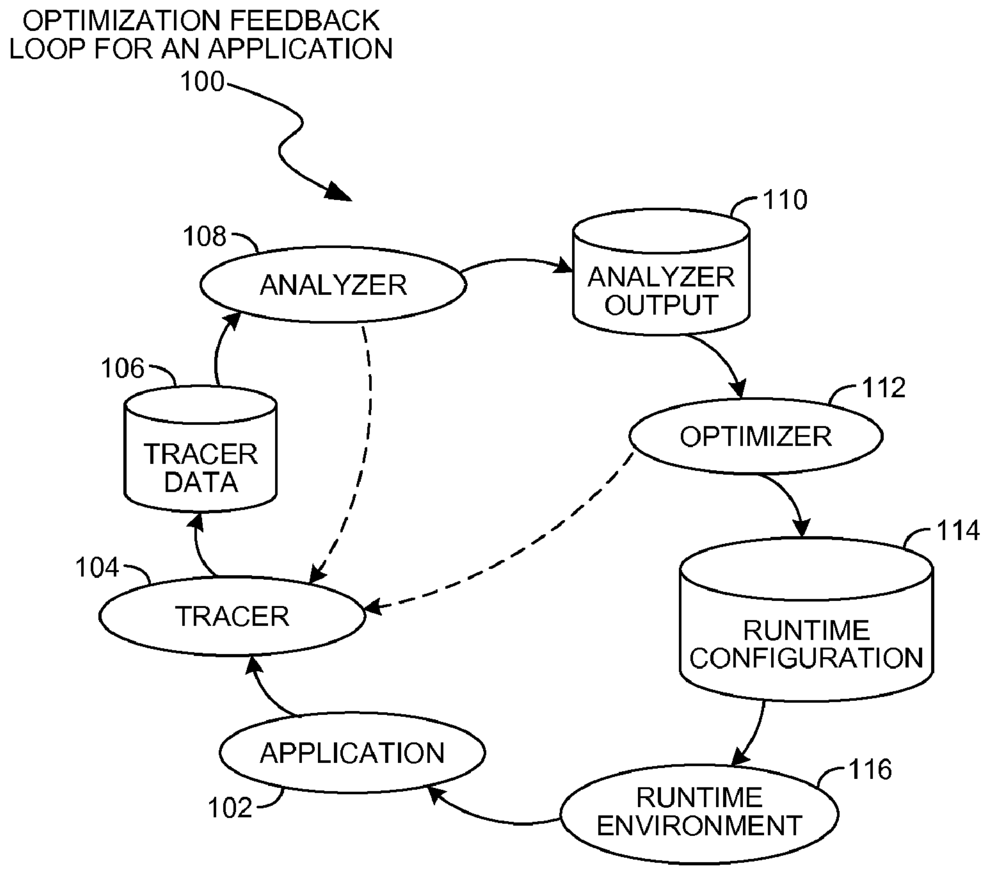
Publication Classification

(51) **Int. Cl.**
G06F 11/36 (2006.01)

(52) **U.S. Cl.**
CPC *G06F 11/3612* (2013.01)
USPC **717/128**

(57) **ABSTRACT**

An analysis system may perform network analysis on data gathered from an executing application. The analysis system may identify relationships between code elements and use tracer data to quantify and classify various code elements. In some cases, the analysis system may operate with only data gathered while tracing an application, while other cases may combine static analysis data with tracing data. The network analysis may identify groups of related code elements through cluster analysis, as well as identify bottlenecks from one to many and many to one relationships. The analysis system may generate visualizations showing the interconnections or relationships within the executing code, along with highlighted elements that may be limiting performance.



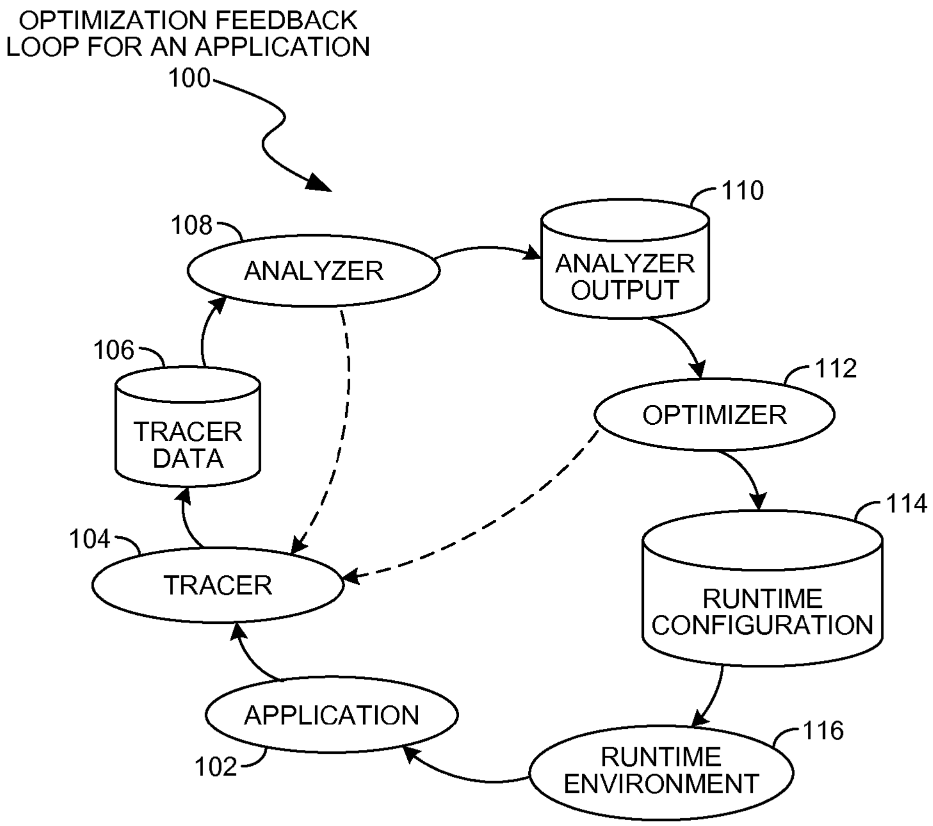

OPTIMIZATION FEEDBACK LOOP FOR AN APPLICATION
100
108
ANALYZER
110
ANALYZER OUTPUT
106
TRACER DATA
112
OPTIMIZER
104
TRACER
114
RUNTIME CONFIGURATION
APPLICATION
102
116
RUNTIME ENVIRONMENT

FIG. 1

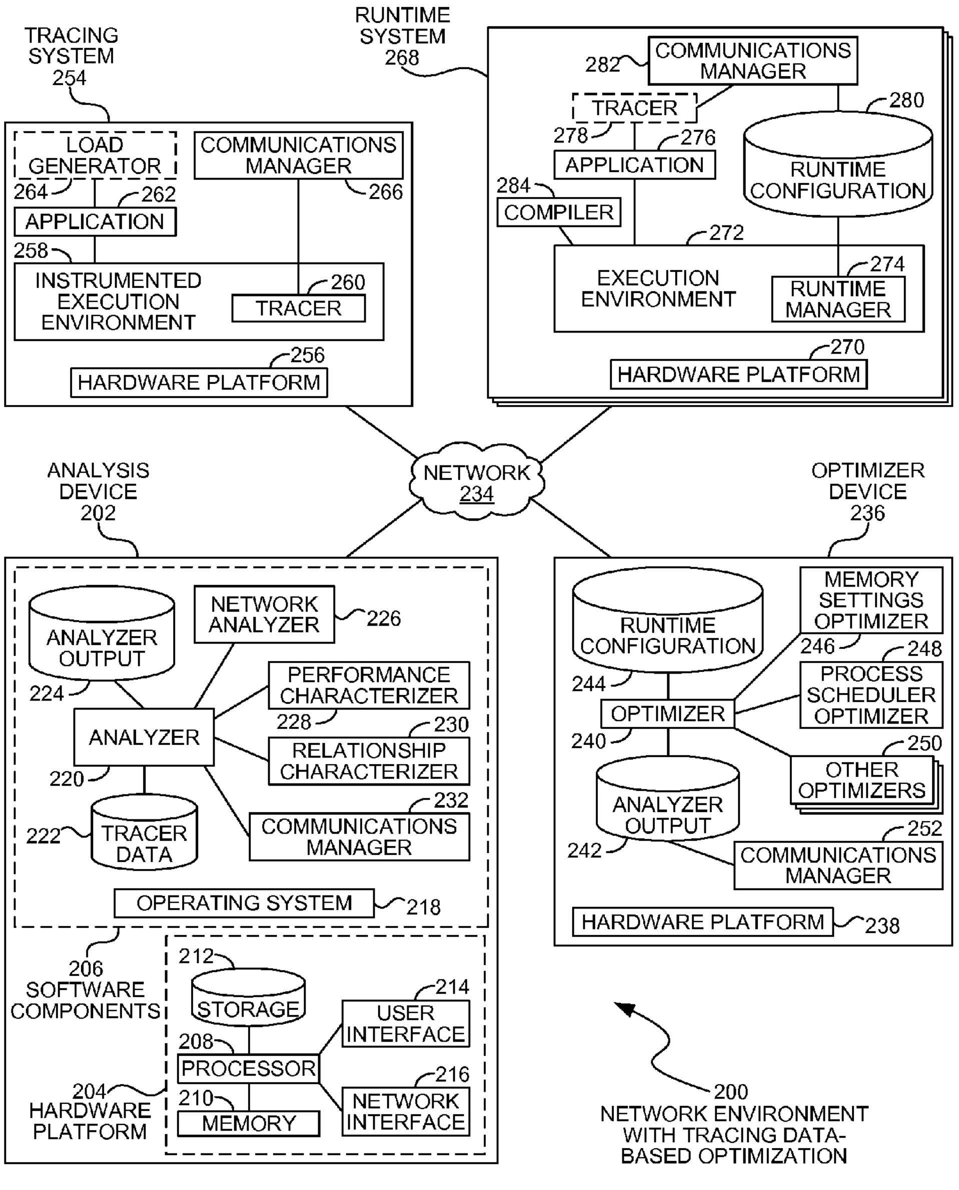

TRACING SYSTEM 254
LOAD GENERATOR
COMMUNICATIONS MANAGER
264
262
266
APPLICATION
258
INSTRUMENTED EXECUTION ENVIRONMENT
260
TRACER
256
HARDWARE PLATFORM
RUNTIME SYSTEM 268
282
COMMUNICATIONS MANAGER
TRACER
280
278
276
APPLICATION
RUNTIME CONFIGURATION
284
COMPILER
272
EXECUTION ENVIRONMENT
274
RUNTIME MANAGER
270
HARDWARE PLATFORM
NETWORK 234
ANALYSIS DEVICE 202
ANALYZER OUTPUT
NETWORK ANALYZER
226
PERFORMANCE CHARACTERIZER
224
228
ANALYZER
230
RELATIONSHIP CHARACTERIZER
220
232
222
TRACER DATA
COMMUNICATIONS MANAGER
OPERATING SYSTEM
218
206 SOFTWARE COMPONENTS
212
STORAGE
214
USER INTERFACE
208
PROCESSOR
216
204 HARDWARE PLATFORM
210
MEMORY
NETWORK INTERFACE
OPTIMIZER DEVICE 236
RUNTIME CONFIGURATION
MEMORY SETTINGS OPTIMIZER
246
248
244
PROCESS SCHEDULER OPTIMIZER
OPTIMIZER
240
250
OTHER OPTIMIZERS
ANALYZER OUTPUT
252
242
COMMUNICATIONS MANAGER
HARDWARE PLATFORM
238
200 NETWORK ENVIRONMENT WITH TRACING DATA-BASED OPTIMIZATION

FIG. 2

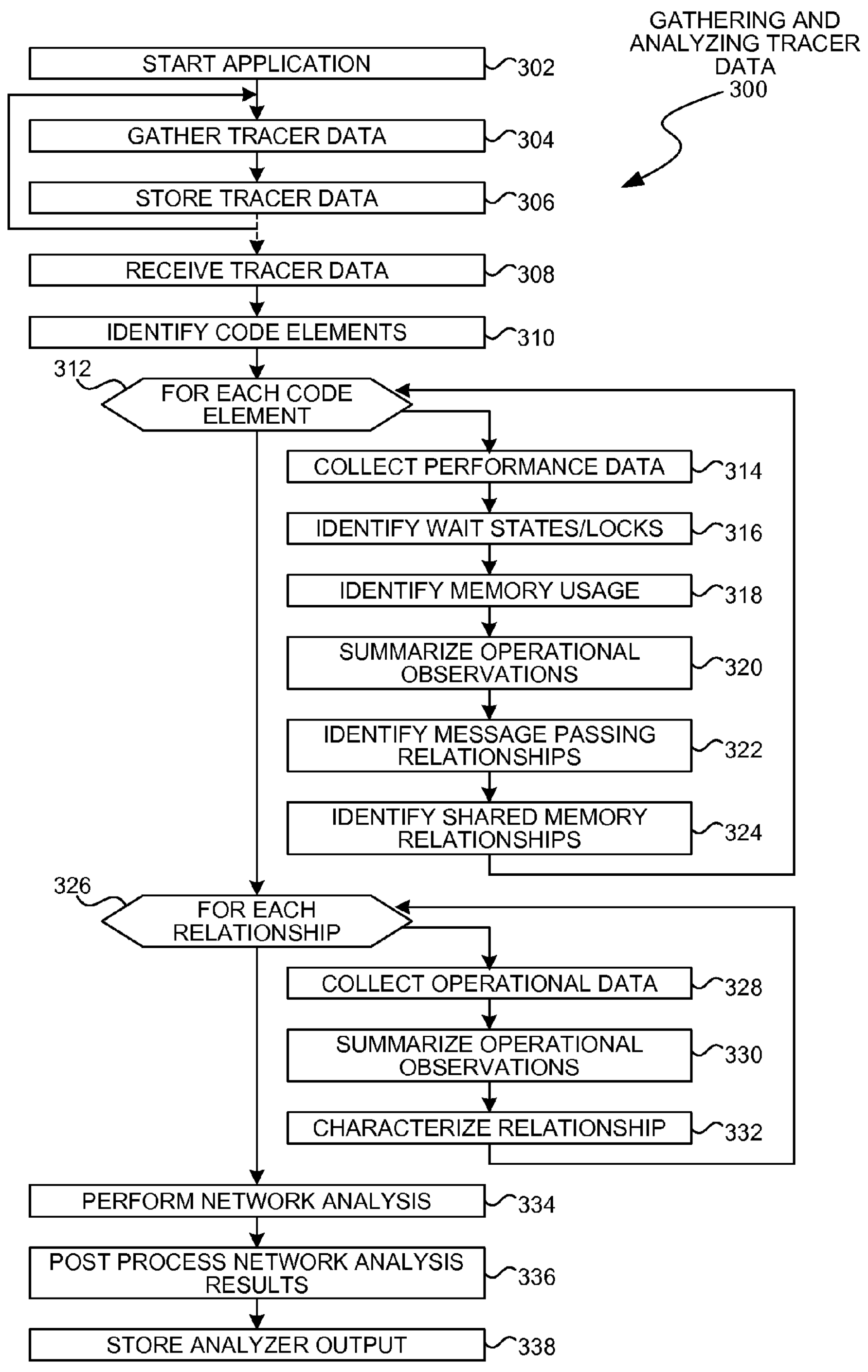

GATHERING AND ANALYZING TRACER DATA
300
START APPLICATION
302
GATHER TRACER DATA
304
STORE TRACER DATA
306
RECEIVE TRACER DATA
308
IDENTIFY CODE ELEMENTS
310
312
FOR EACH CODE ELEMENT
COLLECT PERFORMANCE DATA
314
IDENTIFY WAIT STATES/LOCKS
316
IDENTIFY MEMORY USAGE
318
SUMMARIZE OPERATIONAL OBSERVATIONS
320
IDENTIFY MESSAGE PASSING RELATIONSHIPS
322
IDENTIFY SHARED MEMORY RELATIONSHIPS
324
326
FOR EACH RELATIONSHIP
COLLECT OPERATIONAL DATA
328
SUMMARIZE OPERATIONAL OBSERVATIONS
330
CHARACTERIZE RELATIONSHIP
332
PERFORM NETWORK ANALYSIS
334
POST PROCESS NETWORK ANALYSIS RESULTS
336
STORE ANALYZER OUTPUT
338

FIG. 3

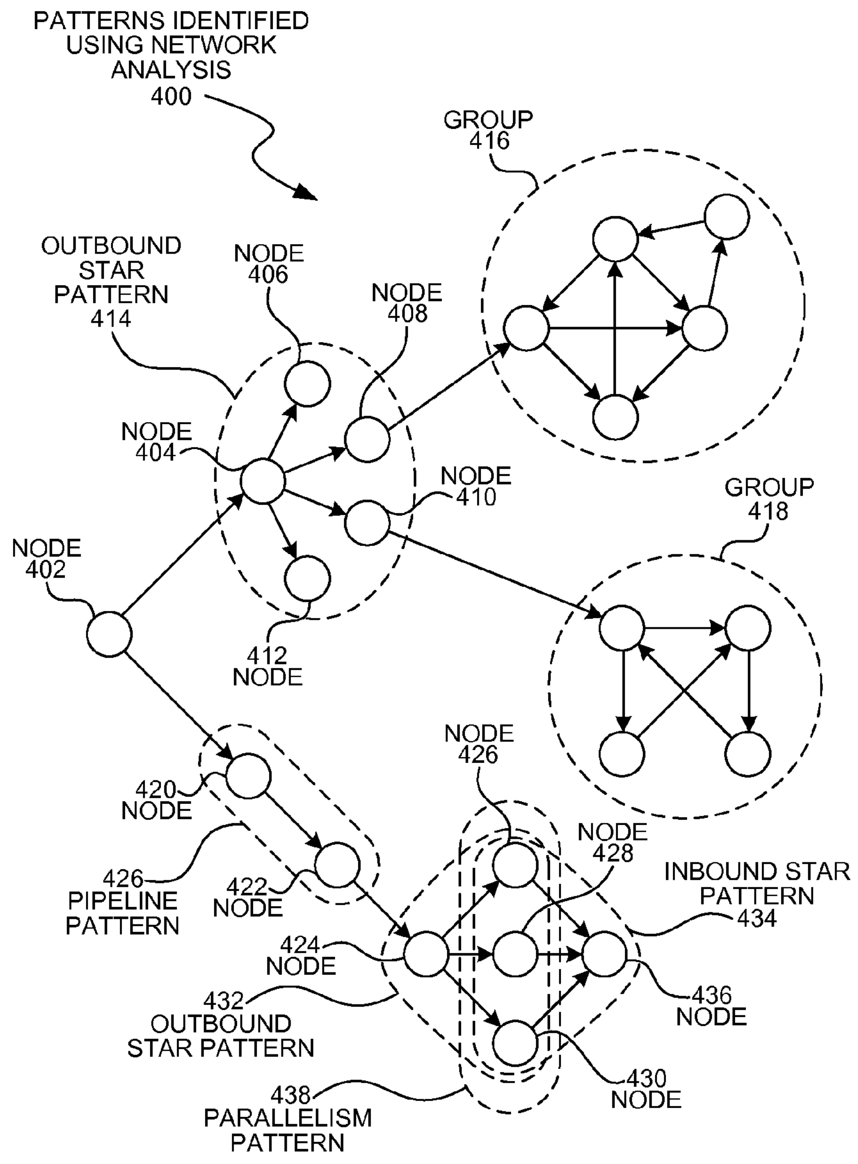

PATTERNS IDENTIFIED USING NETWORK ANALYSIS 400
GROUP 416
OUTBOUND STAR PATTERN 414
NODE 406
NODE 408
NODE 404
NODE 410
GROUP 418
NODE 402
412 NODE
NODE 426
420 NODE
NODE 428
INBOUND STAR PATTERN 434
426 PIPELINE PATTERN
422 NODE
424 NODE
436 NODE
432 OUTBOUND STAR PATTERN
430 NODE
438 PARALLELISM PATTERN

FIG. 4

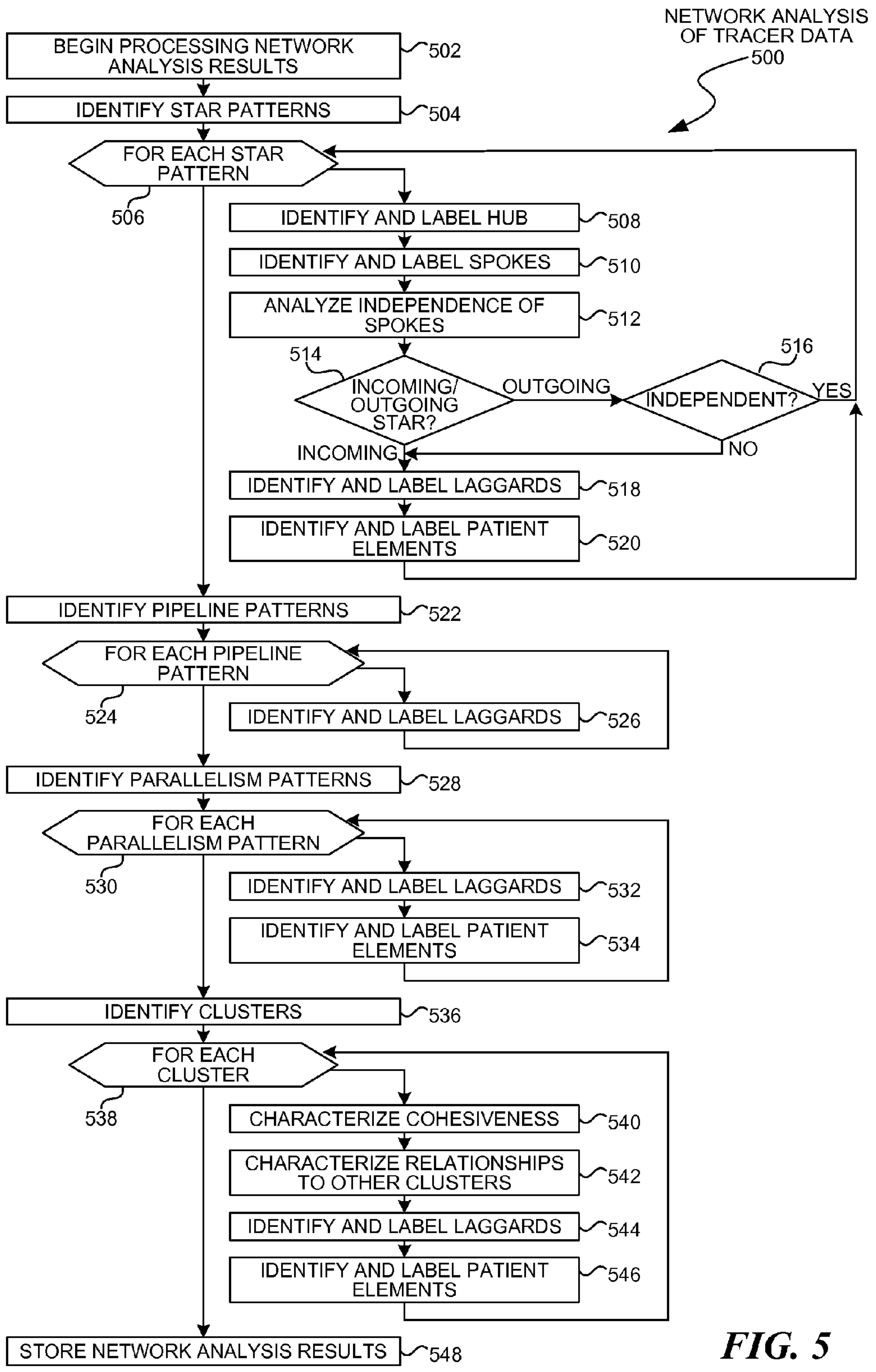

NETWORK ANALYSIS OF TRACER DATA
500
BEGIN PROCESSING NETWORK ANALYSIS RESULTS
502
IDENTIFY STAR PATTERNS
504
FOR EACH STAR PATTERN
506
IDENTIFY AND LABEL HUB
508
IDENTIFY AND LABEL SPOKES
510
ANALYZE INDEPENDENCE OF SPOKES
512
514
INCOMING/ OUTGOING STAR?
OUTGOING
516
INDEPENDENT?
YES
NO
INCOMING
IDENTIFY AND LABEL LAGGARDS
518
IDENTIFY AND LABEL PATIENT ELEMENTS
520
IDENTIFY PIPELINE PATTERNS
522
FOR EACH PIPELINE PATTERN
524
IDENTIFY AND LABEL LAGGARDS
526
IDENTIFY PARALLELISM PATTERNS
528
FOR EACH PARALLELISM PATTERN
530
IDENTIFY AND LABEL LAGGARDS
532
IDENTIFY AND LABEL PATIENT ELEMENTS
534
IDENTIFY CLUSTERS
536
FOR EACH CLUSTER
538
CHARACTERIZE COHESIVENESS
540
CHARACTERIZE RELATIONSHIPS TO OTHER CLUSTERS
542
IDENTIFY AND LABEL LAGGARDS
544
IDENTIFY AND LABEL PATIENT ELEMENTS
546
STORE NETWORK ANALYSIS RESULTS
548

FIG. 5

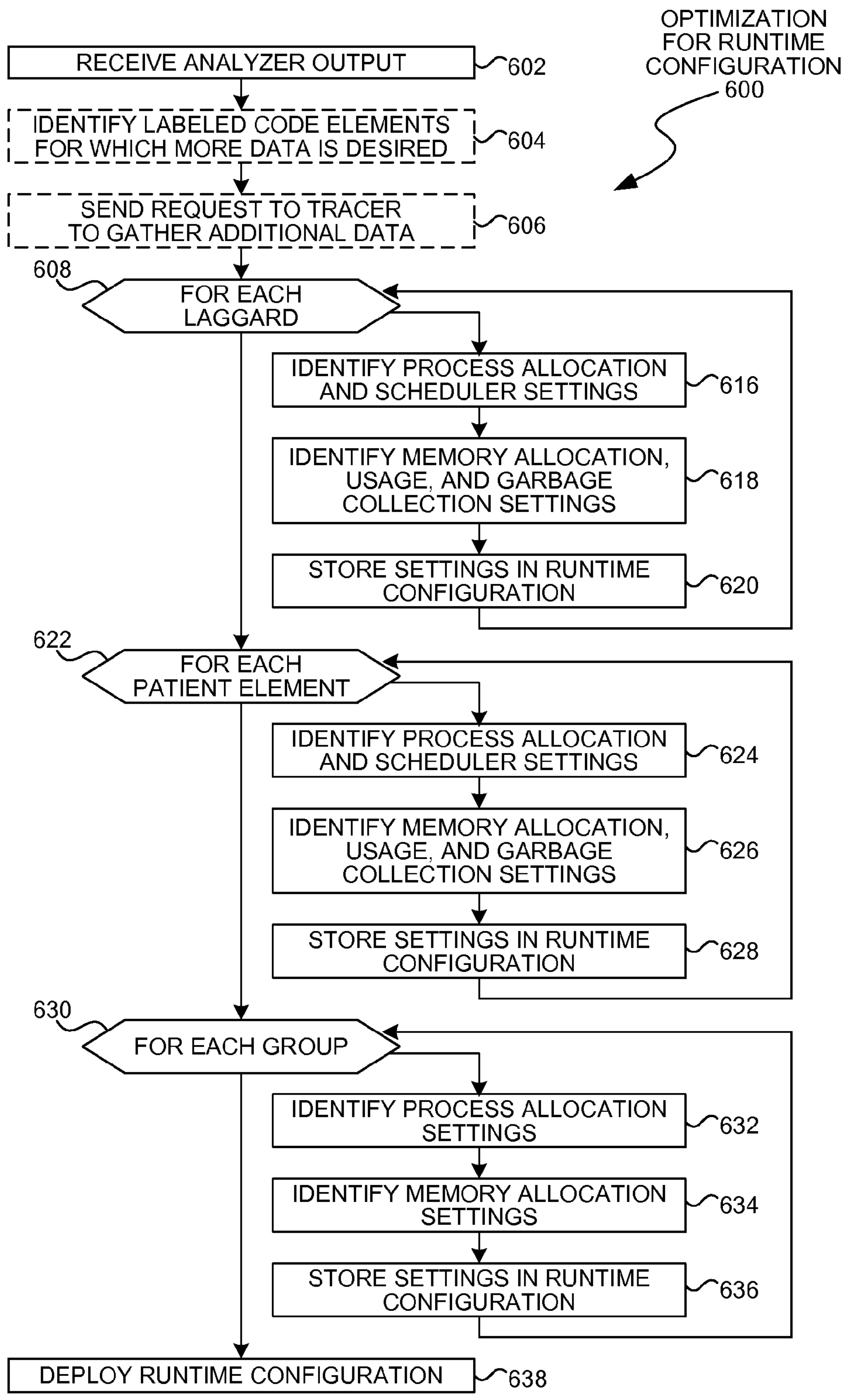

OPTIMIZATION FOR RUNTIME CONFIGURATION
600
RECEIVE ANALYZER OUTPUT
602
IDENTIFY LABELED CODE ELEMENTS FOR WHICH MORE DATA IS DESIRED
604
SEND REQUEST TO TRACER TO GATHER ADDITIONAL DATA
606
608
FOR EACH LAGGARD
IDENTIFY PROCESS ALLOCATION AND SCHEDULER SETTINGS
616
IDENTIFY MEMORY ALLOCATION, USAGE, AND GARBAGE COLLECTION SETTINGS
618
STORE SETTINGS IN RUNTIME CONFIGURATION
620
622
FOR EACH PATIENT ELEMENT
IDENTIFY PROCESS ALLOCATION AND SCHEDULER SETTINGS
624
IDENTIFY MEMORY ALLOCATION, USAGE, AND GARBAGE COLLECTION SETTINGS
626
STORE SETTINGS IN RUNTIME CONFIGURATION
628
630
FOR EACH GROUP
IDENTIFY PROCESS ALLOCATION SETTINGS
632
IDENTIFY MEMORY ALLOCATION SETTINGS
634
STORE SETTINGS IN RUNTIME CONFIGURATION
636
DEPLOY RUNTIME CONFIGURATION
638

FIG. 6

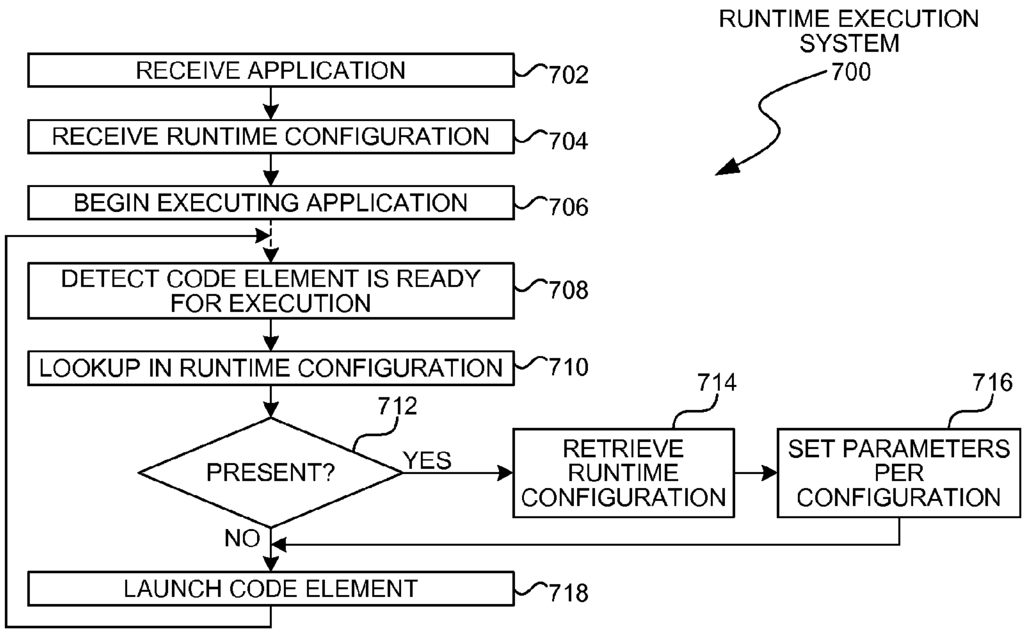

RUNTIME EXECUTION SYSTEM
700
RECEIVE APPLICATION
702
RECEIVE RUNTIME CONFIGURATION
704
BEGIN EXECUTING APPLICATION
706
DETECT CODE ELEMENT IS READY FOR EXECUTION
708
LOOKUP IN RUNTIME CONFIGURATION
710
PRESENT?
712
YES
NO
RETRIEVE RUNTIME CONFIGURATION
714
SET PARAMETERS PER CONFIGURATION
716
LAUNCH CODE ELEMENT
718

FIG. 7

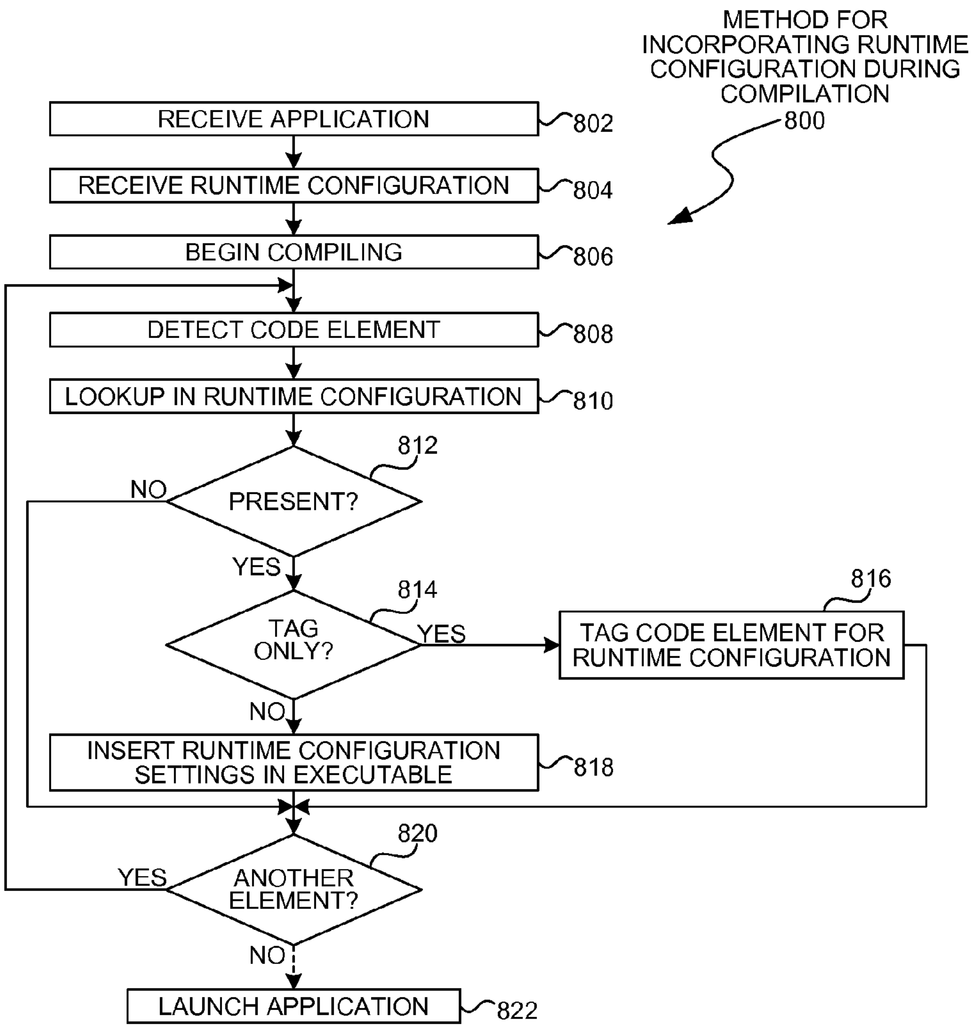

METHOD FOR INCORPORATING RUNTIME CONFIGURATION DURING COMPILATION
800
RECEIVE APPLICATION
802
RECEIVE RUNTIME CONFIGURATION
804
BEGIN COMPILING
806
DETECT CODE ELEMENT
808
LOOKUP IN RUNTIME CONFIGURATION
810
812
PRESENT?
NO
YES
814
TAG ONLY?
YES
NO
816
TAG CODE ELEMENT FOR RUNTIME CONFIGURATION
INSERT RUNTIME CONFIGURATION SETTINGS IN EXECUTABLE
818
820
ANOTHER ELEMENT?
YES
NO
LAUNCH APPLICATION
822

FIG. 8

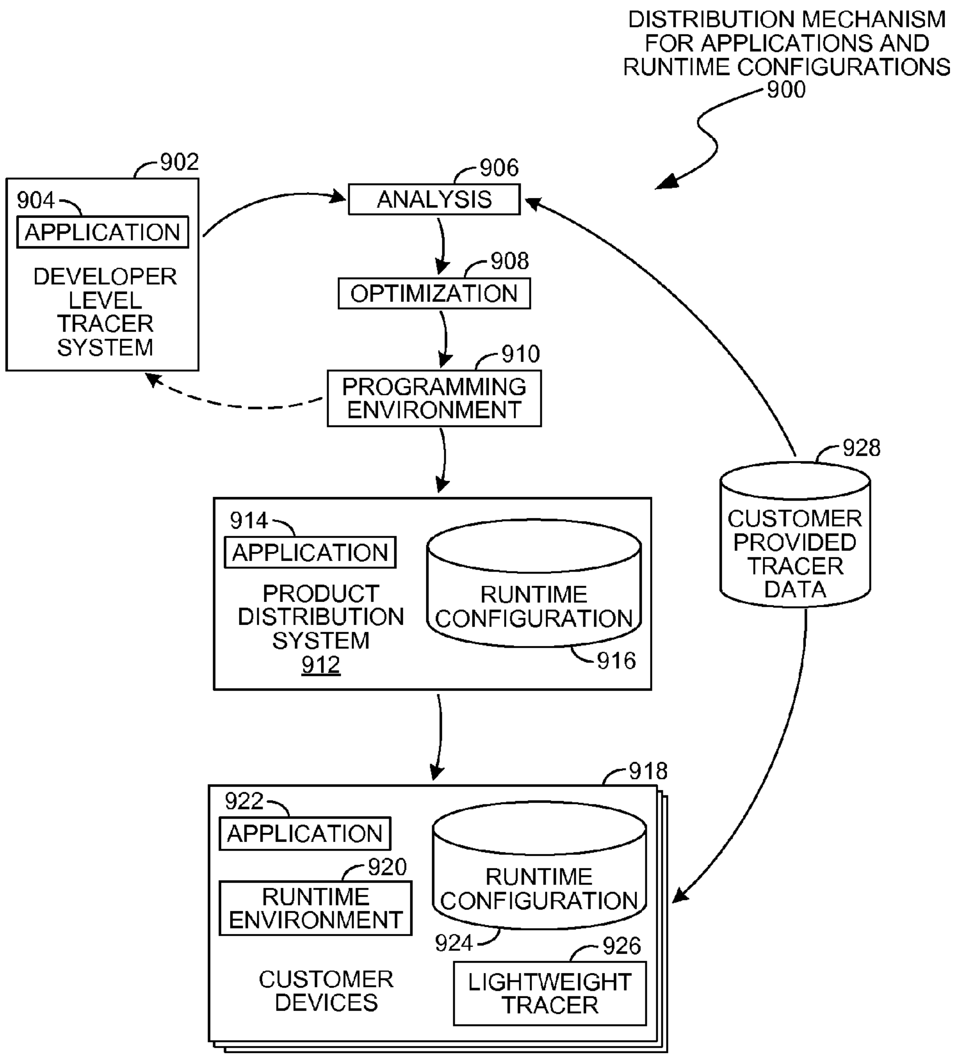

DISTRIBUTION MECHANISM FOR APPLICATIONS AND RUNTIME CONFIGURATIONS
900
902
904
APPLICATION
DEVELOPER LEVEL TRACER SYSTEM
906
ANALYSIS
908
OPTIMIZATION
910
PROGRAMMING ENVIRONMENT
914
APPLICATION
PRODUCT DISTRIBUTION SYSTEM
912
RUNTIME CONFIGURATION
916
928
CUSTOMER PROVIDED TRACER DATA
918
922
APPLICATION
920
RUNTIME ENVIRONMENT
RUNTIME CONFIGURATION
924
926
LIGHTWEIGHT TRACER
CUSTOMER DEVICES

FIG. 9

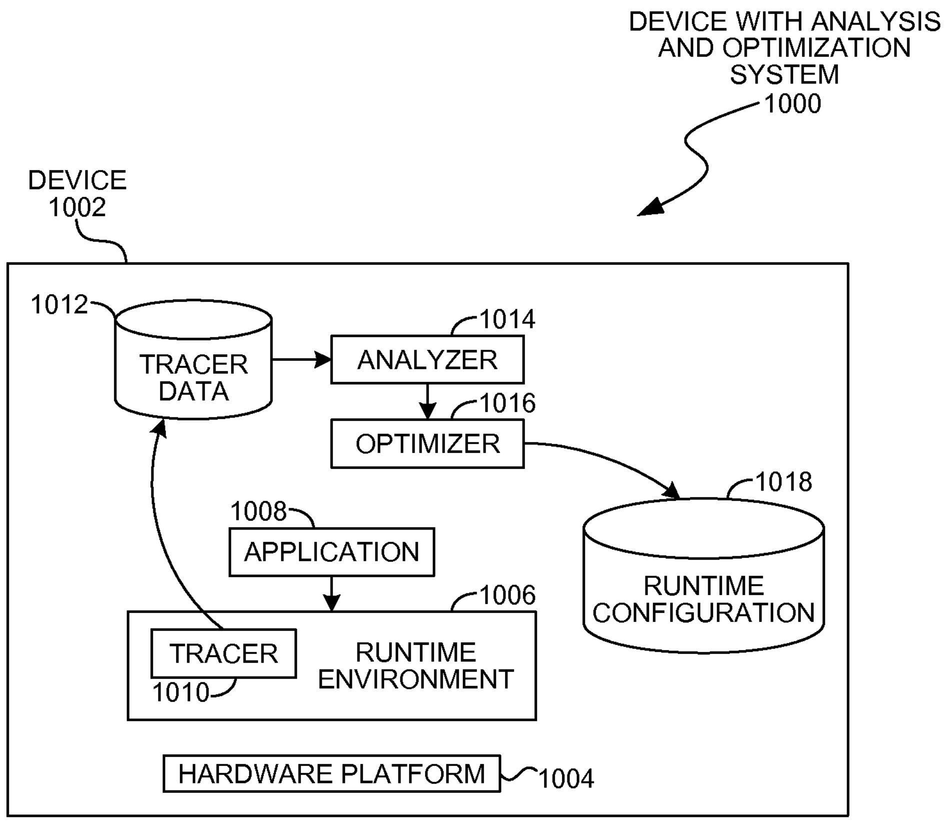

DEVICE WITH ANALYSIS AND OPTIMIZATION SYSTEM
1000
DEVICE
1002
1012
TRACER DATA
1014
ANALYZER
1016
OPTIMIZER
1018
RUNTIME CONFIGURATION
1008
APPLICATION
1006
TRACER
RUNTIME ENVIRONMENT
1010
HARDWARE PLATFORM
1004

FIG. 10

RUNTIME SETTINGS DERIVED FROM RELATIONSHIPS IDENTIFIED IN TRACER DATA

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This patent application claims the benefit of and priority to U.S. Provisional Patent Application Ser. No. 61/801,298 filed 15 Mar. 2013 by Ying Li, the entire contents of which are expressly incorporated by reference.

BACKGROUND

[0002] Increasing performance of a computer software application may have benefits in different scenarios. On one end of the scale, large applications that may execute in parallel on many server computers may benefit from decreased hardware costs when an application executes faster, as fewer instances of the application may be deployed to meet demand. On the other end of the scale, applications that may execute on battery-powered devices, such as mobile telephones or portable computers may consume less energy and give a better user experience when an application executes faster or otherwise has increased performance.

SUMMARY

[0003] An analysis system may perform network analysis on data gathered from an executing application. The analysis system may identify relationships between code elements and use tracer data to quantify and classify various code elements. In some cases, the analysis system may operate with only data gathered while tracing an application, while other cases may combine static analysis data with tracing data. The network analysis may identify groups of related code elements through cluster analysis, as well as identify bottlenecks from one to many and many to one relationships. The analysis system may generate visualizations showing the interconnections or relationships within the executing code, along with highlighted elements that may be limiting performance.

[0004] A settings optimizer may use data gathered from a tracer to generate optimized settings for executing an application. The optimizer may determine settings that may be applied to the application as a whole, as well as settings for individual code elements, such as functions, methods, and other sections of code. In some cases, the settings may be applied to specific instances of code elements. The settings may include processor related settings, memory related settings, and peripheral related settings such as network settings. The optimized settings may be distributed in the form of a model or function that may be evaluated at runtime by a runtime manager. In some embodiments, the optimized settings may be added to source code either automatically or manually.

[0005] A runtime system may use a set of optimized settings to execute an application. The optimized settings may have specific settings for the application, groups of code elements, individual code elements, and, in some cases, specific instances of code elements. The runtime system may detect that a code element is about to be executed, then apply the optimized settings for that code element. In some embodiments, the optimized settings may be determined by some calculation or algorithm that may be evaluated. Some optimized settings may be determined using parameters consumed by a code element as well as other parameters not consumed by the code element. The runtime system may apply settings to a process scheduler, memory manager, or other operating system component.

[0006] 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

[0007] In the drawings,

[0008] FIG. **1** is a diagram illustration of an embodiment showing a system for optimizing an application from tracing data.

[0009] FIG. **2** is a diagram illustration of an embodiment showing a network environment for optimization from tracing data.

[0010] FIG. **3** is a flowchart illustration of an embodiment showing a method for gathering and analyzing tracing data.

[0011] FIG. **4** is a diagram illustration of an example embodiment showing patterns that may be recognized from tracer data.

[0012] FIG. **5** is a flowchart illustration of an embodiment showing a method for network analysis of tracer data.

[0013] FIG. **6** is a flowchart illustration of an embodiment showing a method for optimization for runtime configuration.

[0014] FIG. **7** is a flowchart illustration of an embodiment showing a method for a runtime execution system.

[0015] FIG. **8** is a flowchart illustration of an embodiment showing a method for incorporating runtime configuration during compilation.

[0016] FIG. **9** is a diagram illustration of an embodiment showing a process for developing and distributing applications.

[0017] FIG. **10** is a diagram illustration of an embodiment showing a device with self-learning and self-optimizing execution of an application.

DETAILED DESCRIPTION

Analyzing Tracer Data Using Network Analysis

[0018] An analyzer may use network analysis techniques on tracer data from an application. A tracing system may gather performance and other data while an application executes, from which relationships between code elements may be identified. The relationships may be derived from message passing relationships, shared memory objects, dependencies, spawn events, function calls, and other data that may be available. The tracing data may include performance and resource usage measurements.

[0019] The network analysis may examine the relationships between code elements and the characteristics of those relationships. In many embodiments, the relationships may include data passed over a relationship, frequency and directionality of communications, and other characteristics.

[0020] In many cases, the tracer data may include performance or dynamic data for the code elements and the relationships. For example, the tracer data may include resource consumption data for a code element, such as processor utilization, memory consumed, network or other peripheral accesses, and other data. The resource consumption may include idle time waiting for resources to become available or waiting for other code elements to complete execution. Some embodiments may include memory consumption information such as heap allocation, heap changes, garbage collection information, and other memory related data. Relationship data may include frequency and size of data passed across a relationship, speed of communication, latency, idle time, and other dynamic factors.

[0021] The network analysis may identify individual code elements or groups of code elements that may affect the performance of an application. In some cases, the analysis may identify bottlenecks, choke points, or other code elements that may have significant contributions to the performance of an application.

[0022] In many cases, the static analysis of an application may not reveal actual bottlenecks in an application. For example, a programmer may not be able to fully understand an application created with several disjointed components. However, analysis of tracer data may reveal one or more unexpected bottlenecks.

[0023] Such a situation may occur when using object oriented programming models, where different executable components may be joined together in manners that may never have been contemplated during design of the components. As such, the performance of the components themselves, as well as the overall application may be difficult to predict.

[0024] Such a situation may also occur when using functional programming models, where messages may be passed between independent sets of code and many processes spawned. The effects of spawning many processes may not be fully comprehended when an application may be under load and the bottlenecks may not be predictable when programming the application.

[0025] The network analysis may identify several categories of code elements, such as elements that are depended upon by other elements, branch points that spawn many other elements, collection points that depend on many other elements, groups of elements that frequently interact, pipelines of related code elements that may process things in series, code elements that operate largely independent of other elements, and other categories.

[0026] A programmer may use the network analysis to understand the interacting parts of an application to identify areas to improve performance, reliability, or for some other use. In some embodiments, the network analysis output may identify specific code elements that may fall in the various categories, and the programmer may refactor, improve, or otherwise improve the application in different ways.

[0027] In some embodiments, a network analyzer may perform a two-stage analysis. The first stage may collect high level performance data that may, for example, take a snapshot of various performance counters at a regular interval. A first analysis may be performed to identify candidates for deeper tracing. The second stage may gather more detailed data, such as capturing every message passed by a specific code element. The detailed data may be used for further analysis, optimization, or other uses.

Optimized Settings for Code Elements

[0028] A settings optimizer may generate optimized settings for code components of an application from tracer data. The settings optimizer may operate with an analyzer to highlight certain code elements, then determine a set of execution parameters to improve overall execution of an application. The execution parameters may be applied at runtime with an automated system, or may be manual settings that a programmer may add to executable code.

[0029] The settings optimizer may vary parameters relating to operating system-level functions, such as memory settings, processor settings, peripheral settings, and other lower level functions. In some embodiments, such functions may be made available to an application through a virtual machine, which may be a system virtual machine or process virtual machine. By varying operating system-level parameters, the settings optimizer may determine how to run an application faster or with less resources without changing the application itself.

[0030] In some cases, the settings optimizer may generate optimized parameters that may be implemented by a programmer or other human. The optimized parameters may highlight areas of an application that may be refactored, redesigned, or otherwise improved to address various issues such as performance, reliability, and other issues. In one use scenario, the settings optimizer may identify hardware and software settings that may an administrator may use to determine when to deploy an application and on which hardware platform to deploy an application.

[0031] The settings optimizer may determine optimized settings using tracer data gathered while monitoring a running application. In some cases, the tracer data may be raw, tabulated data that may be processed into a mathematical model that may be used to predict the behavior of an application when a specific parameter may be changed. In other cases, the tracer data may be analyzed to find maximums, minimums, standard deviation, median, mean, and other descriptive statistics that may be used with an algorithm or formula to determine optimized settings.

[0032] The term "optimized" as used in this specification and claims means only "changed". An "optimized" setting may not be the absolute best setting, but one that may possibly improve an outcome. In many cases, a settings optimizer may have algorithms or mechanisms that may improve or change application behavior, but may not be "optimized" in a narrow definition of the word, as further improvements may still be possible. In some cases, an "optimized" setting create by a settings optimizer may actually cause performance or resource utilization to degrade. Throughout this specification and claims, the term "optimized" shall include any changes that may be made, whether or not those changes improve, degrade, or have no measurable effect.

[0033] The settings optimizer may classify a code element based in the behavior observed during tracing, then apply an optimization mechanism based on the classification. For example, the memory consumption and utilization patterns of a code element may cause a code element to be classified as one which may use a steady amount of memory but may create and delete memory objects quickly. Such a code element may be identified based on the memory usage and a set of optimized settings may be constructed that identify a garbage collection algorithm that optimizes memory consumption.

[0034] In many cases, the behavior of a code element or groups of code elements may not be known prior to running an application. In addition, the behavior may be effected by load. For example, a first code element may spawn a second code element, where the second code element may independently process a data item. While designing the software, a programmer may intend for the second code element to be lightweight enough that it may not adversely affect performance on a single processor. Under load, the first code element may spawn large numbers of the second code element, causing the behavior of the system under load to be different than may be contemplated by a programmer. An example optimized setting may be to launch the independent second code elements on separate processors as a large number of the second code element's processes may consume too many resources on the same processor as the first code element.

[0035] The settings optimizer may operate with a feedback loop to test new settings and determine whether or not the new settings had a desired effect. In such embodiments, the settings optimizer may operate with a tracer or other monitoring system to determine an effect based on a settings change. Such embodiments may use multiple feedback cycles to refine the settings.

[0036] A set of optimized settings may be stored in several different forms. In some cases, the optimized settings may be a human readable report that may include suggested settings that may be read, evaluated, and implemented by a human. In such cases, the report may include references to source code, and one embodiment may display the source code with the optimized settings as annotations to the source code.

[0037] The optimized settings may be stored in a computer readable form that may be evaluated at runtime. In such a form, the settings may be retrieved at runtime from a file or other database. Such a form may or may not include references to source code and may or may not be human readable.

Runtime Use of Analyzed Tracing Data

[0038] A runtime system may apply metadata generated from earlier tracing data to enhance the performance of a computer application. Code elements may be identified for non-standard settings during execution, then the settings applied when the code element is executed.

[0039] The code element may be a function, routine, method, or other individual code blocks to which a setting may be applied. In some cases, a library or other group of code elements may have a set of settings applied to the entire group of code elements. The code element may be a process, thread, or other independently executed code element in some embodiments.

[0040] In some embodiments, individual instances of a code element may be given different treatment than other instances. For example, an instance of a process that may be created with a specific input parameter may be given one set of execution settings while another instance of the same process with a different input parameter value may be given a different set of execution settings.

[0041] The execution settings may be generated from analyzing tracing data. The tracing data may include any type of data gathered during execution of the application, and may typically include identification of code elements and performance parameters relating to those elements.

[0042] The analysis may include identifying specific code elements that may be bottlenecks, then identifying execution settings that may cause those code elements to be executed faster. In an environment with multiple processors, some code elements may serve as dispersion points or collection points for other code elements.

[0043] In a simple example, a central or main process may spawn many independent processes that may be performed by on different, independent processors. In the example, the main process may execute on a single processor and while the main process is executing, the remaining processors may be idle, awaiting the spawned processes. In such an example, the overall performance of the application may be largely affected by the performance of the main process, so any performance increases that may be realized on just the main process may dramatically increase the performance of the overall application.

[0044] Such an example may highlight that some code elements may have a greater effect on overall performance than others. Such code elements may be identified by analyzing an application as a network of code elements, where the network contains code elements that communicate with each other. Network analysis techniques may analyze dependencies between code elements to identify bottlenecks, which may include bottlenecks that spawn other processes or that may collect data or otherwise depend on multiple other processes.

[0045] The network analysis techniques may identify communication paths or dependencies between code elements, then attempt to identify execution settings that may speed up the dependencies. In some embodiments, the analysis may identify parallel dependencies where several independent processes may feed a collection point, then identify the slowest of the processes to increase performance of that process. In some such embodiments, the fastest process of the parallel processes may be identified to lower the performance of the process so that resources may be allocated to one or more of the slower processes.

[0046] 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.

[0047] 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.

[0048] 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.

[0049] Throughout this specification, like reference numbers signify the same elements throughout the description of the figures.

[0050] 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.

[0051] 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.

[0052] 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.

[0053] 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.

[0054] 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.

[0055] FIG. **1** is a diagram of an embodiment **100** showing a sequence for optimizing the execution of an application based on tracer data. Embodiment **100** is an example of a sequence through which an application may be automatically sped up with or without any human intervention.

[0056] Embodiment **100** illustrates a high level sequence where data observed from executing an application may be used to generate optimized settings that may be applied to future runs of the application. The analysis and optimization steps may find specific types of patterns within the observed data, then apply optimizations based on those patterns. A runtime configuration may contain optimized parameter settings that may be applied to specific code elements, such as functions, methods, libraries. The optimized parameter settings may increase, decrease, or otherwise change various resources consumed by the code elements to maximize or minimize an optimization goal.

[0057] A simple example of a use scenario, an analyzer may identify a small set of functions within an application that cause the application to run slowly. These functions may be bottlenecks that, if sped up, may result in a large performance improvement.

[0058] The bottleneck functions may be found by analyzing an application as if the application were a network of interconnected code elements. The network may have communications that pass between code elements, and such communications may be explicit or implicit. An example of explicit communications may be messages passed within a message passing framework. An example of implicit communications may be shared memory objects that may be used to pass data from one code element to another.

[0059] In many programming technologies, the interaction between code elements may not be easily predictable. For example, many object oriented languages allow libraries of code elements to be joined together in different manners. In another example, many programming technologies allow for processes or threads to be spawned as a result of new input. The resource usage when spawning many independent processes or threads may not be predictable when writing the application.

[0060] The tracer data consumed by an analyzer may reflect observed behavior of an application. The observed behavior may show how an application actually behaves, as opposed to a static code analysis that may not show responses to loads placed on the application. In some embodiments, a system may use a combination of tracer data and static code analysis to identify various patterns and generate optimized settings for the application.

[0061] The analysis may include clustering analysis. Clustering analysis may identify groups of code elements that may be somehow interrelated, as well as identify different groups that may not be related. An optimization strategy for groups of code elements may be to place all the members of a group such that they share resources, such as processor or memory resources. The strategy may further identify groups that may be separated from each other. An example result of such analysis may place one group of code elements on one processor in one memory domain and a second group of code elements on another processor in another memory domain.

[0062] The analysis may also include identify pipeline patterns. A pipeline pattern may be made up of multiple code elements that may operate in series. Once identified, a pipeline may be analyzed to determine whether one or more of the code elements may be sped up.

[0063] Parallel patterns may also be identified through network analysis. A parallel pattern may identify two or more code elements that execute in parallel, and an optimization strategy may be to attempt to speed up a laggard. In some cases, a patient code element may be identified, which may be a code element that finishes executing but waits patiently for a laggard to finish. An optimization strategy for a patient code element may be to decrease the resources assigned to it, which may free up resources to be allocated to a laggard code element.

[0064] An application **102** may be traced using a tracer **104**, which may generate tracer data **106**. The application **102** may be any set of executable code for which a set of optimized settings may be generated. In some cases, the application **102** may contain a main application as well as various libraries, routines, or other code that may be called from the main application. In some cases, the application **102** may be merely one library, function, or other software component that may be analyzed.

[0065] The tracer **104** may collect observations that reflect the actual behavior of the application **102**. In some embodiments, a load generator may create an artificial load on the application **102**. In other embodiments, the application **102** may be observed in real-world conditions under actual load.

[0066] The tracer data **106** may identify code elements that may be independently monitored. The code elements may be functions, methods, routines, libraries, or any other section of executable code. For each of the code elements, the tracer data **106** may include resource usage data, such as the amount of memory consumed, processor usage, peripheral device accesses, and other usage data. Many embodiments may include performance data, such as the amount of time to complete a function, number of memory heap accesses, and many other parameters.

[0067] The tracer data **106** may have different levels of detail in different embodiments. Some embodiments may collect data over a time interval and create summary statistics for actions over a time interval. For example, a tracer may report data every two seconds that summarize activities performed during the time period. Some embodiments may trace each memory access or even every processor-level command performed by an application. Other embodiments may have different monitoring mechanisms and data collection schemes.

[0068] A network representation of an application may be created by an analyzer **108**. In one embodiment, an application may be represented by nodes representing code elements and edges representing communications or other relationships between code elements. The network may change as the application executes, with nodes being added or removed while the application responds to different inputs, for example.

[0069] The analyzer **108** may identify certain code elements, memory objects, or other elements for which additional tracer data may be useful. In some embodiments, the analyzer **108** may send a request to the tracer **104** for such additional tracer data.

[0070] The analyzer output **110** may include those code elements, memory objects, or other application elements that may be of interest. An element of interest may be any element that, if changed, may have an effect on the overall application. In many embodiments, the elements of interest may be a small portion of the overall code elements, but where improving resource usage for these elements may improve the application as a while.

[0071] An optimizer **112** may apply various optimization algorithms to the elements of interest. In many cases, an analyzer **108** may assign specific classifications to code elements that may indicate which optimization algorithm to apply. For example, an analyzer **108** may identify an outgoing star pattern and the various components of the pattern. In turn, an optimizer **112** may apply optimization algorithms to each of the labeled code elements according to their classification.

[0072] A runtime configuration **114** may be the output of an optimizer **112** and an input to a runtime environment **116**. The runtime configuration **114** may identify various code elements and the runtime settings that may be applied to the code elements, which the runtime environment **116** may apply as it executes the application **102**.

[0073] The runtime configuration **114** may be a database that contains a code element and its corresponding settings that may be referenced during execution. In some embodiments, the runtime configuration **114** may be a simple tabulated file with predefined settings for certain code elements. In other embodiments, the runtime configuration **114** may be a mathematical model or algorithm from which an optimized setting may be computed at runtime. In such embodiments, an optimized setting may change from one condition to another.

[0074] Embodiment **100** illustrates an optimization cycle that may use observations of application behavior to identify potential code elements for optimization. Such a system may not necessarily optimize every code element, but may identify a subset of code elements that may be bottlenecks or have some other characteristic that may be addressed at runtime to improve the application.

[0075] FIG. **2** is a diagram of an embodiment **200** showing a network environment that may optimize an application using tracing data. Embodiment **200** illustrates hardware components that may implement the process of embodiment **100** in a distributed version.

[0076] The distributed nature of embodiment **200** may use separate devices as an analysis device **202**, an optimizer device **236**, a runtime system **268**, and a tracing system **254**. In such a system, tracer data may be captured by the tracing system **254** and the optimized runtime configuration may be deployed on a runtime system **268**.

[0077] One use scenario of such an architecture may be when tracing operations may use hardware or software components that may not be available on a runtime system. For example, some runtime systems may have a limited amount of computing resources, such as a mobile computing device like a mobile telephone or tablet computer. In such an example, the tracing system may be a high powered computer system that contains an emulator for the target runtime system, where the emulator may contain many more monitoring connections and other capabilities to effectively monitor and trace an application.

[0078] Such a use scenario may also test, optimize, and refine a set of runtime configuration settings during a development stage of an application, then deploy the application with an optimized runtime configuration to various users.

[0079] In some use scenarios, the tracing and analysis steps may be performed as part of application development. In such a scenario, the tracing and analysis steps may monitor an application then identify code elements that may be actual or potential performance bottlenecks, inefficient uses of resources, or some other problem. These code elements may be presented to a developer so that the developer may investigate the code elements and may improve the application.

[0080] In some embodiments, the tracing, analysis, and optimization steps may result in a set of optimized parameters that a programmer may manually incorporate into an application.

[0081] The diagram of FIG. **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.

[0082] Embodiment **200** illustrates an analysis device **202** that may have a hardware platform **204** and various software components. The analysis device **202** as illustrated represents a conventional computing device, although other embodiments may have different configurations, architectures, or components.

[0083] In many embodiments, the analysis device **202** may be a server computer. In some embodiments, the analysis 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.

[0084] 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**.

[0085] 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**.

[0086] 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.

[0087] 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.

[0088] 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.

[0089] 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.

[0090] An analyzer **220** may receive tracer data **222** and generate analyzer output **224**. The tracer data **222** may be raw or preprocessed observations from a tracer that monitors an application while the application executes. The tracer data **222** may be generated from real-world or artificial loads placed on the application, and may reflect the behavior of the application and the various code elements that make up the application.

[0091] The analyzer **220** may have several components that may perform different types of analysis. For example, a network analyzer **226** may analyze the application as if the application were a network of code elements with communications and other relationships between the code elements. The network analyzer **226** may attempt to identify patterns, clusters, groups, and other characteristics from a network topology, as well as identify specific code elements that may be causing performance issues.

[0092] A performance characterizer **228** may be an analysis component that evaluates and characterizes or classifies the performance of various code elements. The classification may assist a network analyzer **226** or other component in identifying problem areas or in determining an appropriate optimization technique.

[0093] One type of characterization performed by a performance characterizer **228** may compare the performance of a particular code element to the average performance of other code elements in a single application or to the average performance of code elements observed from multiple applications. The characterization may identify outliers where code elements have above average or below average performance.

[0094] Another type of characterization from a performance characterizer **228** may identify the type or types of performance issues observed for a code element. For example, a characterization may indicate that a code element had excessive garbage collection, consumed large amounts of memory, or contended for various locks.

[0095] A relationship characterizer **230** may be an analysis component that evaluates and characterizes the relationships between code elements. The classifications may be derived from the actual behavior of the application. Examples of relationships characterizations may include message passing relationships, shared memory relationships, blocking relationships, non-blocking relationships, and other types of characterizations.

[0096] Each type of relationship may assist in classifying a code element for further evaluation or optimization. For example, a blocking relationship where one code element stalls or waits for another code element to finish may have a different optimization algorithm than a message passing relationship that may use a mailbox metaphor to process incoming messages.

[0097] The types of characterizations may reflect the underlying programming logic used for an application. For example, the relationship characterizations that may be found in a functional programming paradigm may be much different than the relationship characterizations from an object oriented programming paradigm.

[0098] A communications manager **232** may be a component that may manage communications between the various devices in embodiment **200**. The communications manager **232** may, for example, retrieve tracer data **222** from the tracing system **254** and may transmit analyzer output **224** to an optimizer device **236**.

[0099] In some embodiments, the communications manager **232** may automatically collect the tracer data **222** as it may become available, then cause the analyzer **220** to begin analysis. The communications manager **232** may also transmit the analyzer output **224** to the optimizer device **236** for further processing.

[0100] The optimizer device **236** may operate on a hardware platform **238**, which may be similar to the hardware platform **204**.

[0101] An optimizer **240** may receive analyzer output **242** and create a runtime configuration **228**, which may be consumed by a runtime system **268**. The optimizer **240** may have a memory settings optimizer **246**, a process scheduler optimizer **248**, as well as other optimizers **250**.

[0102] The memory settings optimizer **246** may determine memory related settings that may be appropriate for a specific code element. The memory settings may include an initial heap size, garbage collection scheme, or other settings that may be memory related.

[0103] The process scheduler optimizer **248** may identify processor related settings for a code element. The processor related settings may include priority settings, processor affinity settings, and other settings. In some embodiments, the ordering or priority of multiple code elements may be defined. For example, a first process that has a dependency or lock on a second process may be scheduled to be executed after the second process.

[0104] The optimizer **240** may apply various optimizations based on the conditions and situations identified by the analyzer **220**. Each situation may have a different optimizer algorithm that may determine runtime settings for an application. In many embodiments, the optimizer **240** may have various other optimizers **250** that may be added over time.

[0105] The optimizer device **236** may have a communications manager **252** similar to the communications manager **232** on the analysis device **202**. The communications manager **252** may enable the various components in embodiment **200** to operate as a single system that may automatically trace, analyze, and optimize an application across the network **234**.

[0106] A tracing system **254** may have a hardware platform **256** that may be similar to the hardware platform **204** of the analysis device. The tracing system may have an instrumented execution environment **258** in which a tracer **260** may monitor an application **262**. Some embodiments may have a load generator **264**, which may exercise the application **262** so that the tracer **260** may observe the application behavior under different use scenarios.

[0107] The tracing system **254** may also have a communications manager **266**, which like its counterpart communication managers **232** and **252**, may serve to automatically implement a sequence of gathering and analyzing tracer data.

[0108] The runtime systems **268** may represent the delivery hardware for the application **262**. In some embodiments, the runtime systems **268** may have a different hardware platform **270** than the tracing system **254**. For example, the instrumented execution environment **258** may be a virtual machine that may execute an operating system emulator for a mobile device, where the mobile device may be the runtime systems **268**. In such an example, an application may be distributed with a runtime configuration that may allow the application to execute faster or using less resources.

[0109] The runtime system **268** may have a hardware platform **270** similar to the hardware platform **204**, on which an execution environment **272** may execute an application **276**. A runtime manager **274** may observe the application **276** as it executes, and may identify a code element prior to execution. The runtime manager **274** may look up the code element in the runtime configuration **280**, and cause the code element to be executed with the settings defined in the runtime configuration **280**.

[0110] In some embodiments, a runtime system **268** may include a tracer **278**, which may collect tracer data that may be transmitted to the analysis device **202**. A communications manager **282** may facilitate such a transmission, among other things.

[0111] In some embodiments, the runtime configuration **280** may be incorporated into an application **276** using a just in time compiler **284**. In such an embodiment, the runtime configuration **280** may be consumed by a compiler **284** to add runtime settings to the application **276**. When the application **276** may be executed, the runtime configuration settings may be embedded or otherwise incorporated into the compiled code. Such a compiler may be a just in time compiler, although in other embodiments, the compiler may be a conventional compiler that may compile code ahead of time.

[0112] FIG. **3** is a flowchart illustration of an embodiment **300** showing a method for gathering and analyzing tracer data. The operations of embodiment **300** may illustrate one method that may be performed by the tracer **104** and analyzer **108** of embodiment **100**.

[0113] 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.

[0114] Embodiment **300** may illustrate a generalized process for gathering and analyzing tracer data. In block **302**, an application may be started. During execution, tracer data may be gathered in block **304** and stored in block **306**. The process may loop continuously to gather observations about the application. In many cases, the application may be subjected to loads, which may be generated in the real world or simulated using load generators.

[0115] The analysis of the tracer data may begin in block **308** when the tracer data is received. Code elements may be identified in block **310** and then processed in block **312**.

[0116] For each code element in block **312**, performance data may be collected in block **314**. Wait states and locks encountered by the code element may be identified in block **316**, as well as memory usage in block **318**. The operational observations may be summarized in block **320**.

[0117] While analyzing a code element, any message passing relationships may be identified in block **322** and any shared memory relationships may be identified in block **324**. Message passing relationships and shared memory relationships may link two code elements together. In some cases, a relationship may be directional, such as message passing relationships, where the directionality may be determined from tracer data. In other cases, the directionality may not be detectable from tracer data.

[0118] In some embodiments, a shared memory object may indicate a relationship. A directionality of shared memory relationships may be indicated when one code element depends on another code element. In such a case, a lock or wait state of one of the code elements may indicate that it is dependent on another code element and may therefore be the receiver in a directional relationship. In some cases, the tracer data may not have sufficient granularity to determine directionality.

[0119] Each relationship may be analyzed in block **326**. For each relationship in block **326**, operational data may be collected in block **328**, which may be summarized in block **330** and used to characterize the relationship in block **332**.

[0120] In many embodiments, a relationship may be classified using various notions of strength. A strong relationship may be one in which many messages are passed or where a large amount of data may be shared. A weak relationship may have little shared data or few messages. Some such embodiments may use a numerical designator for strength, which may be a weighting applied during network analyses.

[0121] Once the code elements and relationships have been classified, network analysis may be performed in block **334**. A more detailed example of a method for network analysis may be found later in this application.

[0122] The network analysis results may be post processed in block **336** and stored in block **338**.

[0123] FIG. **4** is a diagram illustration of an embodiment **400** showing an example of a network analysis that may be performed using application trace data. An application may be analyzed by identifying code elements and relationships between code elements. In the example of embodiment **400**, the various nodes may represent code elements and the relationships between nodes may be illustrated as edges or connections between the nodes. In the example of embodiment **400**, the relationships may be illustrated as directional relationships.

[0124] Embodiment **400** is merely an example of some of the patterns that may be identified through network analysis. The various patterns may indicate different classifications of code elements, which may indicate the type of optimization that may be performed.

[0125] Node **402** may be connected to node **404**, which may spawn nodes **406**, **408**, **410**, and **412**. An outbound star pattern **414** may be recognized through automatic network analysis. The outbound star pattern **414** may have a hub node **404** and several spoke nodes. In many cases, an outbound star pattern may be a highly scalable arrangement, where multiple code elements may be launched or spawned from a single element. The spoke nodes may represent code elements that may operate independently. Such spoke nodes may be placed on different processors, when available, which may speed up an application by operating in parallel.

[0126] A network analysis may also recognize groups **416** and **418**. The groups may include code elements that interrelate, due to calling each other, sharing memory objects, or some other relationship. In general, groups may be identified as having tighter relationships within the group and weaker relationships outside the groups.

[0127] Groups may be optimized by combining the group members to the same hardware. For example, the members of a group may be assigned to the same memory domain or to the same processor in a multi-processor computer, while other groups may be assigned to different memory domains or other processors.

[0128] Node **402** may be connected to node **420**, which may be connected to node **422**, which may be connected to node **424**. The series of nodes **402**, **420**, **422**, and **424** may be identified as a pipeline pattern **426**.

[0129] A pipeline pattern may be a sequence of several code elements that feed each other in series. When analyzing a pipeline pattern, one or more of the code elements may act as a bottleneck. By speeding up a slow element in a pipeline, the overall performance may increase linearly with the increase.

[0130] A pipeline pattern may also be treated as a group, where the members may be placed on the same processor or share the same memory locations. In many cases, a pipeline pattern may be identified when the relationships between the code elements are strong and may pass large amounts of data. By placing all of the pipeline members one the same processor, each member may be processed in sequence with little lag time or delay.

[0131] Node **424** may be connected to nodes **426**, **428**, and **430**, each of which may be connected to node **436**. The network analysis may identify an outgoing star pattern **432**, an inbound star pattern **434**, and a parallelism pattern **438**.

[0132] The analysis of the outgoing star pattern **432** may be similar to the outbound star pattern **414**.

[0133] An inbound star pattern **434** may indicate a bottleneck at the hub node **436**, which may receive messages or share memory with several other nodes **426**, **428**, and **430**. The degree to which node **436** may act as a bottleneck may be affected by the type of relationships. In the case where node **436** receives and processes messages from multiple nodes, the node **436** may experience a much higher workload than other nodes. As such, the amount of processing performed by node **436** may drastically affect the overall performance of an application.

[0134] The hub node of an inbound star pattern may limit the scaling of an application in a multi-processor system, as only one processor may perform the actions of node **436**. As such, the hub node of an inbound star may be flagged for a programmer to consider refactoring or redesigning the code in this area.

[0135] A parallelism pattern **438** may have several processes that may operate in parallel. In applications where the receiving node **436** may depend on results from all three nodes **426**, **428**, and **430** before continuing execution, an optimization routine may identify the slowest node in the parallelism pattern for speed improvement.

[0136] In many cases, nodes that may be bottlenecks may be improved by applying more resources to the code element. The resources may be in the form of additional processor resources, which may be achieved by raising the priority of a code element, placing the code element on a processor that may be lightly loaded, or some other action. In some cases, the resources may be memory resources, which may be improved by increasing memory allocation, changing garbage collection schemes, or other changes.

[0137] When a parallelism pattern may be detected, some of the parallel code element may finish early and may be patiently waiting for other laggard code elements to finish. Those patent code elements may be adjusted to consume fewer resources during their operation. One such adjustment may lower the priority for a patient node or assign fewer memory resources.

[0138] FIG. **5** is a flowchart illustration of an embodiment **500** showing a method for performing network analysis on tracer data. The operations of embodiment **500** may illustrate one method that may be performed during the operations of block **334** of embodiment **300**.

[0139] 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.

[0140] Embodiment **500** illustrates one method for performing network analysis on a graph composed of code elements as nodes and relationships between code elements as edges. Embodiment **500** may be performed on graphs such as the example graph of embodiment **400**.

[0141] The network analysis may begin in block **502** using preprocessed data that may characterize nodes and relationships as described in embodiment **300**. A graph of the various components may be constructed.

[0142] From the graph, star patterns may be identified in block **504**. Each star pattern may be evaluated in block **506**. For each star pattern, the hub nodes may be labeled in block **508** and the spoke nodes may be labeled in block **510**.

[0143] The independence of the spokes may be analyzed in block **512**. When the star pattern is an outgoing star pattern in block **514** and the outbound spokes are independent in block **516**, the process may return to block **506** to process another star pattern.

[0144] When the star pattern is an outgoing star pattern in block **514** and there may be no independence between the outgoing spoke nodes in block **516**, the laggards of the spoke nodes may be identified and labeled in block **518**, and any patient elements may be identified and labeled in block **520**.

[0145] When the star pattern may be an incoming star pattern in block **514**, the laggards may be identified in block **518** and the patient elements may be identified in block **520**. In an incoming star pattern, the laggard and patient elements may be useful to know when the hub of the incoming star may be in a lock state waiting for all of the incoming spokes to complete their work. In an embodiment where the incoming star pattern is not dependent on all of the spoke elements, the laggard and patient elements may not be labeled, but the hub element may be labeled as an incoming hub.

[0146] In block **522**, pipeline patterns may be identified. For each pipeline pattern in block **524**, the laggard elements in the pipeline may be identified in block **526**. The laggards may be one or more elements in a pipeline pattern that may contribute to a performance bottleneck.

[0147] Parallelism patterns may be identified in block **528**. For each parallelism pattern in block **530**, the laggards may be identified in block **532** and the patient elements may be identified in block **534**.

[0148] Clusters may be identified in block **536**. For each cluster in block **538**, the cohesiveness of the cluster may be identified in block **540**.

[0149] The cohesiveness may be a weighting or strength of the grouping. In some embodiments, groups with weak cohesiveness may be divided across different processors or memory domains, while groups with strong cohesiveness may be kept together on the same hardware components.

[0150] The relationships between a given cluster and other clusters may be characterized in block **542**. Clusters with strong inter-cluster relationships may have a higher likelihood for remaining together, while clusters with weak inter-cluster relationships may be more likely to be split.

[0151] Within the clusters, laggards may be labeled in block **544** and patient elements may be labeled in block **546**.

[0152] The network analysis results may be stored in block **548**.

[0153] The stored network analysis results may be automatically consumed by an optimizer routine to generate a runtime configuration. In some embodiments, the stored network analysis results may be used by a programmer to analyze code under development. Such an analysis may assist the programmer in finding bottlenecks and other areas that may adversely affect performance of the application.

[0154] FIG. **6** is a flowchart illustration of an embodiment **600** showing a method for optimizing tracer analysis results to create a runtime configuration. The operations of embodiment **600** may illustrate one method that may be performed by the optimizer **112** of embodiment **100**.

[0155] 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.

[0156] Embodiment **600** illustrates one method for determining runtime configurations that may be optimized derived from patterns observed during network analysis. During network analysis, some of the code elements may be identified as laggards, patient elements, as well as members of groups. Each classification of elements may be optimized by identifying parameters that may speed up certain code elements, make more efficient usage of code elements, or apply settings across groups of code elements.

[0157] The analyzer output may be received in block **602**. In some embodiments, some code elements may have been labeled as requesting more data in block **604**, and identifiers for those code elements may be transmitted to a tracer in block **606**.

[0158] In some embodiments, additional data may be requested in order to identify optimized settings. For example, an initial tracer may capture the various code elements, relationships, and high level performance metrics. After a network analysis identifies specific code elements as being a bottleneck, those code elements may be traced again, but at a more detailed level. The second tracing pass may gather information such as memory usage details, messages passed, processor utilization, or other details from which optimized runtime configuration may be derived.

[0159] For each laggard code element in block **608**, process allocation and scheduler settings may be identified in block **616**. Memory allocation, usage, and garbage collection settings may be identified in block **618**. The settings may be stored in a runtime configuration in block **620**.

[0160] The process allocation settings may assist in placing the laggard on a specific processor. In many embodiments, the process allocation settings may include an affinity or relationship to other code elements. At runtime, code elements that may have a strong positive affinity may be placed on the same processor, while code elements that may have a strong repulsive affinity may be placed on different processors.

[0161] The scheduler settings may assist a process scheduler in determining when and how to execute the laggard. In some embodiments, the scheduler settings may indicate that one code element may be executed before another code element, thereby hinting or expressly determining an order for processing. The scheduler settings may include prioritization of the laggard. In many cases, laggards may be given higher priority so that the laggard process may be executed faster than other processes.

[0162] The memory allocation settings may relate to the amount of memory allocated to a code element as well as various settings defining how memory may be managed while a code element executes. For example, a setting may define an initial heap allocation, while another setting may define the increment at which memory may be additionally allocated. The memory related settings may include garbage collection schemes, as well as any configurable parameters relating to garbage collection.

[0163] Each patient element may be analyzed in block **622**. For each patient element, process allocation and scheduler settings may be determined in block **624**, and memory allocation, usage, and garbage collection settings may be identified in block **626**. The settings may be stored in a runtime configuration in block **628**.

[0164] With a laggard code element, the optimized settings may attempt to cause the code element to speed up. Higher prioritization, more memory, or other settings may help the code element complete its work faster, thereby causing the entire application to execute faster.

[0165] With a patient code element, the optimized settings may attempt to limit the amount of resources. For example, lowering the priority of the process may cause a patient code element to be executed slower and may free up processor resources that may be allocated to laggard processes. Such an example may illustrate efficient deployment of resources that may improve an application's performance.

[0166] Each group or cluster may be evaluated in block **630**. For each group in block **630**, process allocation settings may be identified in block **632** and memory allocation settings may be identified in block **634**. The settings may be stored in the runtime configuration in block **636**.

[0167] For groups, the runtime settings may identify affinity between code elements such that group members may be processed on the same processor, have access to the same memory domain, or afforded some other similar treatment. In some cases, the group designation may permit elements to be separated at runtime when the group cohesiveness may be weak, but may otherwise attempt to keep the group together.

[0168] FIG. **7** is a flowchart illustration of an embodiment **700** showing a method using runtime configuration as an application executes. The operations of embodiment **700** may illustrate one method that may be performed by the runtime environment **116** of embodiment **100**.

[0169] 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.

[0170] Embodiment **700** illustrates one example of how optimized runtime configuration may be applied. Embodiment **700** may be an example of an interpreted or compiled runtime environment that may identify when a code element may be executed, then look up and apply a setting to the code element.

[0171] An application may be received in block **702**, and a runtime configuration may be received in block **704**. Execution of the application may begin in block **706**.

[0172] While the code executes, block **708** may detect that a code element is about to be executed. The code element may be looked up in the runtime configuration in block **710**. When the code element is present in the runtime configuration in block **712**, the settings may be retrieved in block **714** and any configuration changes made in block **716**. The code element may be launched in block **718**. If the code element is not found in the runtime configuration in block **712**, the code element may be launched in block **718** with default settings.

[0173] Some embodiments may apply the same runtime configuration settings to each instance of a code element. Other embodiments may apply one set of runtime configuration settings to one instance and another set of runtime configuration settings to another instance. Such embodiments may evaluate the input parameters to a code element to determine which set of settings to apply. Some such embodiments may evaluate other external parameters or settings to identify conditions for when to apply optimized configuration settings.

[0174] FIG. **8** is a flowchart illustration of an embodiment **800** showing a method for incorporating runtime configuration during compiling.

[0175] 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.

[0176] Embodiment **800** may illustrate one method by which runtime configuration may be incorporated into a compiled version of an application. In one mechanism, the compiled version of an application may be tagged. A tag may cause a runtime environment to identify that a code element has optimized settings available, which may cause the settings to be retrieved and implemented.

[0177] In another mechanism of incorporation using a compiler, the optimized runtime configuration settings may be placed into the runtime executable by the compiler. Such settings may be added to the compiled code such that the execution may be performed without a runtime configuration.

[0178] The compilation may be performed as a just in time compilation. In a typical embodiment, an application may be compiled into an intermediate language, which may be compiled at runtime. In other embodiments, the compilation may be performed prior to execution and the executable code may be stored and retrieved prior to execution.

[0179] An application may be received in block **802** and the runtime configuration received in block **804**. Compiling may begin in block **806**.

[0180] A code element may be detected in block **808** and a lookup may be performed in block **810** to determine whether or not the runtime configuration may contain settings for the code element. When the settings are not present in block **812**, the process may skip to block **820**. If more elements are present in block **820**, the process may return to block **808**.

[0181] When the settings are present in block **812**, the runtime configuration may be added to the executable in two manners. In a tagging manner in block **814**, the executable code may be tagged in block **816** to have a lookup performed during execution. In a non-tagging manner in block **814**, the runtime configuration settings may be inserted into the executable code in block **818**.

[0182] The process may revert to block **808** if more elements exist in block **820**. When all the elements have been compiled in block **820**, the application may be launched in block **822**.

[0183] FIG. **9** is a diagram illustration of an embodiment **900** showing one development and distribution mechanism for applications with runtime configurations. Embodiment **900** illustrates one system for using network analysis in an offline mode, then distributing an application with a runtime configuration to client devices.

[0184] A developer-level tracing system **902** may execute an application **904**, from which analysis **906** and optimization **908** may be performed on the tracer data. The results of the analysis **906** and optimization **908** may be displayed in a programming environment **910**. A programmer may view the results in the programming environment **910** and may update or change the application **904**, then re-run the analysis and optimization.

[0185] In some embodiments, the programming environment **910** may include an editor, compiler, and other components. In some cases, the developer level tracer **902** and the analysis and optimization components may be parts of the programming environment.

[0186] Once the changes to the application may be complete, a product distribution system **912** may distribute the application **914** and runtime configuration **916** to various customer devices **918**.

[0187] The customer devices **918** may have a runtime environment **920** that executes the application **922** with a runtime configuration **924**. In some embodiments, a lightweight tracer **926** may collect some data that may be transmitted as customer provided tracer data **928**, which may be incorporated back into the development process.

[0188] FIG. **10** is a diagram illustration of an embodiment **1000** showing a single device in which tracing, analysis, optimization, and execution of an application may occur. Embodiment **1000** may be a self-contained device that may learn or adapt to executing a particular application faster or with better resource utilization.

[0189] A device **1002** may contain a hardware platform **1004** on which a runtime environment **1006** may execute an application **1008**. While the application **1008** executes, a tracer **1010** may collect tracer data **1012**. An analyzer **1014** and optimizer **1016** may process the tracer data **1012** to generate a runtime configuration **1018**. The runtime configuration **1018** may then be used to execute the application **1008**. In some embodiments, a feedback loop may then again trace the application and continually refine the runtime configuration **1018**, thereby continually improving the application.

[0190] 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.

What is claimed is:

1. A method performed by a computer processor, said method comprising:

receiving relationship metadata derived from analysis of data collected while tracing an application, said relationship metadata comprising relationships between code elements related to said application;

during runtime for said application:

preparing a first code element for execution;

determining a set of settings for said first code element from said metadata; and

causing said first code element to be executed using said set of settings.

2. The method of claim **1**, said metadata further comprising relationship strength indicators derived from tracing analysis of said application.

3. The method of claim **2**, said relationship strength indicators comprising attractive strength indicators and repulsive strength indicators.

4. The method of claim **3**, a first attractive strength indicator being used at runtime to cause said first code element to be executed using a shared resource with a second code element.

5. The method of claim **4**, said shared resource being a memory resource.

6. The method of claim **4**, said shared resource being a processor resource.

7. The method of claim **4**, said shared resource being a peripheral resource.

8. The method of claim **7**, said peripheral resource comprising a network interface.

9. The method of claim **2**, said set of settings causing said first code element to be executed on a first processor, said first processor executing a second code element, said first code element and said second code element having an attractive strength indicator for a first relationship.

10. The method of claim **9**, said first relationship being derived from a dependency observed during tracing said application.

11. The method of claim **10**, said dependency comprising a commonly shared memory object.

12. The method of claim **10**, said dependency comprising a sequential processing relationship between said first code element and said second code element.

13. A system comprising:

a processor;

a runtime environment that:

receives relationship metadata derived from analysis of data collected while tracing an application, said relationship metadata comprising relationships between code elements related to said application;

during runtime for said application:

prepares a first code element for execution;

determines a set of settings for said first code element from said metadata; and

causes said first code element to be executed using said set of settings.

14. The system of claim **13**, said metadata further comprising relationship strength indicators derived from tracing analysis of said application.

15. The system of claim **14**, said relationship strength indicators comprising attractive strength indicators and repulsive strength indicators.

16. The system of claim **15**, a first attractive strength indicator being used at runtime to cause said first code element to be executed using a shared resource with a second code element.

17. The system of claim **16**, said shared resource being a memory resource.

18. The system of claim **16**, said shared resource being a processor resource.

19. The system of claim **16**, said shared resource being a peripheral resource.

20. The system of claim **19**, said peripheral resource comprising a network interface.

* * * * *