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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
22 June 2017 (22.06.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/106351 A1

(51) **International Patent Classification**:
G06F 7/00 (2006.01) *G06F 12/00* (2006.01)
G06F 9/445 (2006.01) *G06F 17/30* (2006.01)
G06F 9/45 (2006.01)

(21) **International Application Number**:
PCT/US2016/066665

(22) **International Filing Date**:
14 December 2016 (14.12.2016)

(25) **Filing Language**: English

(26) **Publication Language**: English

(30) **Priority Data**:
14/968,296 14 December 2015 (14.12.2015) US

(71) **Applicants**: **DATAWARE VENTURES, LLC** [US/US]; 9040 S. Rita Road, Suite 1270, Tucson, AZ 85747 (US). **THE ARIZONA BOARD OF REGENTS ON BEHALF OF THE UNIVERSITY OF ARIZONA** [US/US]; The University of Arizona, Tech Transfer Arizona, Services Annex, 4th Fl, PO Box 210300A, Tucson, AZ 85721 (US).

(72) **Inventors**: **SNODGRASS, Richard, T.**; PO Box 210077, Tucson, AZ 85721-0077 (US). **DEBRAY, Saumya, K.**; 1040 E. 4th Street, Tucson, AZ 85721 (US). **ZHANG, Rui**; 9040 S. Rita Road, Suite 1270, Tucson, AZ 85747 (US). **LIU, Yang**; 1040 E. 4th Street, Tucson, AZ 85721 (US).

(74) **Agents**: **SOLOWAY, Norman, P.** et al.; c/o Hayes Soloway P.C., 4640 E. Skyline Drive, Tucson, AZ 85718 (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, DJ, 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, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, 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, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

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

[Continued on next page]

(54) **Title**: BROADENING FIELD SPECIALIZATION

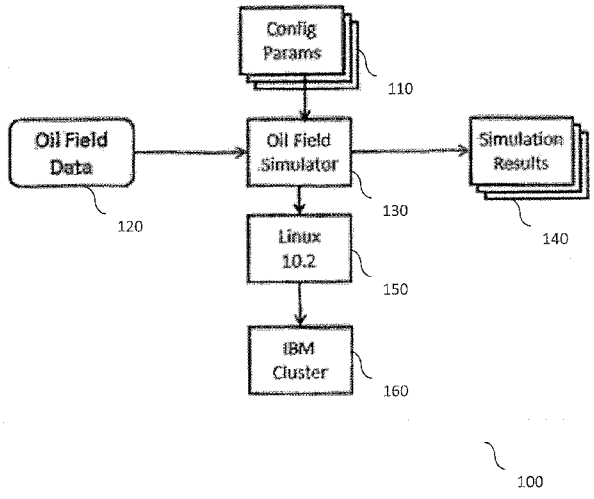


FIG. 1

(57) **Abstract**: Four extensions to the conventional field specialization process are disclosed. The first extension is cross-application value flows, where a value transfers out of one application and subsequently into another application. The second extension is an inter-application analysis. Static and dynamic analysis is performed by a Spiff Toolset not just on the source code of a single application, but also across the data read and written by that application. The third extension is invariant cross-application termination, verifying the possibility of an invariant originating in an application and terminating in a specialization opportunity in a separate application. The fourth extension relates to run-time code placement algorithms to mitigate an increase in I-cache pressure and L2-cache pressure. A MaxResidency algorithm uses information from dynamic analysis of provided workloads before DBMS compilation and specific structure of the query evaluation plan to place run-time code, thus retaining possible run time improvement of that code.

WO 2017/106351 A1

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

Broadening Field Specialization

[0001] This invention was made with Government support under Grant Nos. IIS0803229, CNS0938948, and 1318343 awarded by NSF. The Government has certain rights in the invention.

[0002] The present disclosure is generally related to field specialization broadening to multiple application domains, and more particularly is related to systems and methods for broadening field specialization to extend particular values of the invariants beyond the source code of the application to the data sources themselves.

[0003] A database management system (DBMS) is a collection of software programs that manage the storage and access of data. As larger volumes of data are being generated nowadays and thus must be stored and efficiently accessed, DBMSes have been adopted across a wide range of application domains. Driven by such ubiquitous deployments over the last four decades, DBMSes have been designed and engineered based on a few data models that are generally applicable to those domains. The relational data model is the one most prevalently adopted by commercial and open-source DBMSes. A significant amount of effort has been devoted to efficiently support this data model.

[0004] Due to the generality of the relational data model, relational database management systems are themselves general, in that they can handle whatever schema the user specifies and whatever query or modification is presented to them. Relational operators work on essentially any relation and must contend with predicates specified on any attribute of the underlying relations. Through such innovations as effective indexing structures, innovative concurrency control mechanisms, and sophisticated query optimization strategies, the relational DBMSes available today are very efficient. Such generality and efficiency has enabled their proliferation and use in many domains.

[0005] Nevertheless, such generality is realized via multiple layers of indirections and sophisticated code logic. Efficiency can be further enhanced for DBMSes by exploiting invariant values present during the execution of such systems. Field specialization technology has been developed to automatically identify invariants and effect code specialization based upon invariants. Field specialization is the process of inserting spiffs into DBMS code so that the DBMS can specialize itself at runtime to

exploit runtime invariants. A spiff, which stands for specializer in the field, is code that dynamically creates specialized code at DBMS runtime. The specialized code is both smaller and faster than the original unspecialized code. Field specialization gets its name from the fact that the code is specialized "in the field," i.e., after the DBMS has been deployed and is running at the end user's site. A spiff uses the actual value of a runtime invariant—which can only be known at runtime—to dynamically produce code that is specialized to that particular value of the runtime invariant.

[0006] It would be desirable to have the field specialization broadened to multiple application domains to elaborate a paradigm of computer science.

[0007] Embodiments of the invention relate to field specialization broadening to multiple application domains, and more particularly is related to systems and methods for broadening field specialization to extend particular values of the invariants beyond the source code of the application to the data sources themselves.

[0008] In accordance with one embodiment of the present invention, an ecosystem specification is disclosed to enable field specialization broadening. This specification states which (potentially multiple) applications are involved, what (potentially multiple) input data sources are read by one or more applications, what intermediate and final data products are produced by those applications, and what services are invoked by the applications, thereby providing valuable information on how intermediate results are communicated among the various applications.

[0009] Capturing in the ecosystem specification the input, intermediate, and final data sources along with the multiple applications participated in the computing allows field specialization to be broadened. The flow of particular values of the invariants may be extended beyond the source code of the application to the data sources themselves, across file read and write operations and even via network communication, such as passing the resulting data to the operating system (e.g., Linux), which then passes this data into the networking component, which then passes the data to other destinations.

[0010] In some embodiments, a static and/or a dynamic analysis determines that an invariant within a specialization opportunity arose even earlier than an assignment to a C variable within the application; rather, it arose from a data component at a particular location within an input data file, such as oil field data or a config parameter utilized by an oil-field simulation software. Once such a value flow is

determined, spiffs are created in the appropriate applications and, when given a particular oil field data file, the spiffs are executed to produce a speccode of a specialization opportunity within the oil field simulator application. This speccode could be associated with the particular oil field data file, to be loaded dynamically into the oil field simulator application (or passed to the operating system or stored in the IBM cluster) when that application is run. In this way, all of the applications involved in a computation can participate and be optimized, via a broadened conception of field specialization.

[0011] In some embodiments, the method of field specialization broadening comprises a sequence of steps: (i) deciding what (extended) invariants are there, (ii) deciding which of these extended invariants should be specialized upon, (iii) deciding what code sequence within a specialization opportunity should actually be specialized, (iv) deciding when to perform the specialization (and thus, within which application), (v) deciding where to store the generated speccode, and finally, (vi) deciding how to transfer the speccode to the site (application) where it will be invoked.

[0012] In some embodiments, the method of field specialization broadening comprises extending step (i) to include invariants that flow from external data or external applications, step (iv) to include spiffs located in an application separate from the application containing the source of the invariant that is being utilized by the spiff, or even autonomous from any application, and steps (v) and (vi) to include speccode stored separately from the application being specialized.

[0013] In some embodiments, a method is disclosed to estimate cache pressure given a specific workload and to solve the problem of instruction cache misses in the presence of code added at run-time, such as in the context of micro-specialization, when a specialized version of an often-executed portion of the DBMS can speed query evaluation. Several novel run-time code placement algorithms are implemented. The algorithms include a MaxResidency algorithm, which uses information from dynamic analysis of provided workloads prior to DBMS compilation as well as the specific structure of the query evaluation plan to place run-time code wisely, thus retaining most of the possible run time improvement of that code. It is demonstrated that the algorithm minimizes I-cache pressure for workloads that utilize as many as 1000 dynamically-linked functions. Via this algorithm,

aggressive micro-specialization of a large number of functions invoked during query evaluation can effectively mitigate an increase in I-cache pressure and thus also L2-cache pressure.

[0014] Other systems, methods, features, and advantages of the present disclosure will be or become apparent to one with skill in the art upon examination of the following drawings and detailed description. It is intended that all such additional systems, methods, features, and advantages be included within this description, be within the scope of the present disclosure, and be protected by the accompanying claims

[0015] Reference will be made to exemplary embodiments of the present invention that are illustrated in the accompanying figures. Those figures are intended to be illustrative, rather than limiting. Although the present invention is generally described in the context of those embodiments, it is not intended by so doing to limit the scope of the present invention to the particular features of the embodiments depicted and described.

[0016] FIG. 1 is a block illustration of a single application field specialization process.

[0017] FIG. 2 is a block diagram illustrating the spiff toolset, in accordance with an exemplary embodiment provided by this disclosure.

[0018] FIG. 3 is a block illustration of a field specialization broadening process involving three extensions in accordance with an embodiment of the present invention.

[0019] FIG. 4 is an illustration of a flow process of broadened field specialization with an embodiment of the present invention.

[0020] FIG. 5 is an illustration of an alternative flow process of broadened field specialization with an embodiment of the present invention.

[0021] FIG. 6 is an illustration of field specialization for elaboration a paradigm of computer science with an embodiment of the present invention.

[0022] FIG. 7 is an exemplary Cache Hierarchy with an embodiment of the present invention.

[0023] FIG. 8 is an illustration of percentage increase of L1 Misses of Query22 with Various Bee Placements with an embodiment of the present invention.

[0024] FIG. 9 is an illustration of Cumulative Cache-Line Reference histogram with an embodiment of the present invention.

[0025] FIG. 10 is a block diagram illustrating workflow of profile-agnostic bee placement algorithms with an embodiment of the present invention.

[0026] FIG. 11 is a block diagram illustrating workflow of SPP-guided bee placement algorithms with an embodiment of the present invention.

[0027] FIG. 12 is a schematic diagram showing stages of a query plan tree with an embodiment of the present invention.

[0028] Fig. 13 is an exemplary flow diagram for the evaluation of cache pressure when introducing code into a DMBS with an embodiment of the present invention.

[0029] FIG. 14 is an illustration showing impact of increasing number of bees on cache miss rate percentage with an embodiment of the present invention.

[0030] Various implementations and embodiments may be practiced in line with the specification. All of these implementations and embodiments are intended to be included within the scope of the invention.

[0031] Many embodiments of the disclosure may take the form of computer-executable instructions, including algorithms executed by a programmable computer. However, the disclosure can be practiced with other computer system configurations as well. Certain aspects of the disclosure can be embodied in a special-purpose computer or data processor that is specifically programmed, configured or constructed to perform one or more of the computer-executable algorithms described below.

[0032] The disclosure also can be practiced in distributed computing environments, where tasks or modules are performed by remote processing devices that are linked through a communications network. Moreover, the disclosure can be practiced in Internet-based or cloud computing environments, where shared resources, software and information may be provided to computers and other devices on demand. In a distributed computing environment, program modules or subroutines may be located in both local and remote memory storage devices. Aspects of the disclosure described below may be stored or distributed on computer-readable media, including magnetic and optically readable and removable computer disks, fixed magnetic disks, floppy disk drive, optical disk drive, magneto-optical disk drive, magnetic tape, hard-disk drive (HDD), solid state drive (SSD), compact flash or non-volatile memory, as well as distributed electronically over networks including the

cloud. Data structures and transmissions of data particular to aspects of the disclosure are also encompassed within the scope of the disclosure.

[0033] Moreover, while field specialization is disclosed herein primarily with respect to a DBMS, it should be understood that the concepts provided herein may be applied to any program that manipulates data and in particular, performs complex analysis on that data.

[0034] A "spiff," which stands for *specializer* in the field, is code that dynamically creates specialized code at DBMS runtime. "Field specialization" is the process of inserting spiffs into DBMS code so that the DBMS can specialize itself by exploiting runtime invariants. The specialized code (which may be referred to herein as "speccode") is both smaller and faster than the original unspecialized code. Field specialization gets its name from the fact that the speccode is generated and invoked "in the field," i.e., after the DBMS has been deployed and is running at the end user's site. A spiff uses the actual value of a runtime invariant—which is obtained at runtime—to dynamically produce code that is specialized to that particular value of the runtime invariant.

[0035] In Applicants' co-pending U.S. Patent Application serial number 14/368,265, which the present application claims priority to, the term "micro-specialization" is equivalent to the term "field specialization" as used herein; the term "bee" is equivalent to the term "spiff" as used herein; an instantiated bee is equivalent to "specialized code" as used herein, which is the result of a spiff; and the HRE (hive runtime environment) is equivalent to "SRE" (spiff runtime environment) as used herein.

[0036] FIG. 1 is a block illustration of a single application field specialization process. The exemplary single application field specialization process **100** is related to an oil field over time. The single-application field specialization process has been disclosed and included in co-pending patent application with application number 62/142,325 (Dataware 15.01-P). Both the initial field specialization and the broader one discussed here apply to any application domain that processes the same data multiple times.

[0037] Referring to Figure 1, a simulator 130 receives inputs from a field dependent source data **120** and a user-defined parameter **110**. As shown in Figure 1, the field dependent source data **120** ("oil field data") specifies a structure of an oil

field, a three-dimensional assemblage of rock structures, some of which hold various mixtures of oil, gas, water at various pressures and some of which are impermeable. The user-defined parameter **110** (also referred as "config params" or workload) specifies wells to be drilled at various times and fluids to be extracted at various rates over time. The simulator **130** then computes how the fluids flow throughout the oil field over time, over simulated time of perhaps many years, producing a "simulation result". The simulations may be implemented many times, each time with a different user-defined parameter **110** and with a single simulation result **140** as an output. In some embodiments, the simulator may pass the simulation result **140** to a Linux operation system **150**, which then passes the data to a computer cluster (such as an IBM cluster) **160**.

[0038] Although Figure 1 is shown with the exemplary oil field related application, it is understood that the single application field specialization process may be applicable to various fields.

[0039] In some embodiments, the oil field simulator may be written in a high-level language such as C. A Spiff toolset (shown in Figure 2) that analyzes the simulator can follow the flow of values through this C program, flowing from cold code into hot code as a data value is copied from one C variable, perhaps within a complex data structure, to another. The Toolset can then identify specialization opportunities: segments of hot code, that is, a program component that's frequently executed, containing variables that are invariant at runtime. The values of these variables are known to have originated in cold code (i.e., written once) and are used (i.e., read many) in branching or other statements in hot code, such that by knowing the particular value of an invariant, the identified hot code segment may be specialized to run faster, say by removing a branch statement along with the dead branch(es) from hot code. The specialization opportunities enable the creation of spiffs, which are placed in the cold code and can produce specialized code (abbreviated as speccode).

[0040] The aforementioned field specialization focuses on the source code of the application, such as the oil field simulator shown in Figure 1, performing static and dynamic analysis to identify invariants (C variables) and specialization opportunities (segments of code) that can then be specialized given a particular value for each relevant invariant.

[0041] FIG. 2 is a block diagram illustrating a spiff toolset, in accordance with an exemplary embodiment provided by this disclosure. The spiff tool toolset **200** receives inputs as an Application's source code (or application code) **210**, one or more workload **220**, and an Ecosystem Specification **230** (referred to herein as "Ecosystem Spec") and outputs a Specialized Source Code **240**. In some embodiments, a graphical user interface (GUI) is placed at a higher level than the Spiff Tools to be used by the application developer to request that the tools be executed in sequence and to visualize how the field specialization was achieved, by showing in an appropriate manner the intermediate and generated results. This GUI also provides actions that the developer can perform that will impact the operation of other tools. In some embodiments, the spiff tool toolset **200** is constructed as a field specialization broadening module. In one embodiment, the source code **210** is a DBMS source code comprised within a DBMS **212** of one application.

[0042] The spiff tool toolset **200** provides an end-to-end solution that takes source files for a program or suite of related programs and automatically provides field-specialized versions of those source files, including the code for spiff generation, spiff compilation, spiff instantiation, spiff invocation, and spiff garbage collection. In some embodiments, the spiff toolset **200** includes a number of tools, including but not limited to an invariant finder **201**, an invariant checker **202**, a summarizer **203**, a snipper finder **204**, and a spiff maker **205**. The invariant finder **201** may couple to both the Application's source code **210** and one or more workloads **220** and perform static analysis on the AST and output zero or more Invariant Intervals. The invariant checker **202** ensures the validity of output Invariant Intervals from the invariant finder **201**. The summarizer **203** produces execution summaries, which provide output as a list of functions, statements, and variables along with their execution statistics. Such information indicates "hot spots" within the application that could benefit from field specialization. The Snippet Finder **204** first expands invariants to across program executions by tracking which variables are read from files and where those values are put into files. This results in invariant intervals that span multiple executions. The tool also tracks when the value was first written to the file and when it is deleted. The Spiff maker **205** takes one or more Candidate Snippets as input and output the Specialized Source Code **240**.

[0043] In some embodiments, the aforementioned tools within the spiff toolset **200** are placed within a single computer apparatus. In some embodiments, the aforementioned tools within the spiff toolset **200** are placed among distributed computers.

[0044] The Ecosystem Specification **230** provides additional information that enables the analysis performed by the simulator **130** to be broadened considerably. The specification **230** states which (potentially multiple) applications are involved, what (potentially multiple) input data sources are read by one or more applications, what intermediate and final data products are produced by those applications, and what services are invoked by the applications, thereby providing valuable information on how intermediate results are communicated among the various applications. Thus the Ecosystem Specification **230** may capture a bigger picture beyond the single oil field simulator **130**, which only takes a single "oil field data" source and one "config params" data source as inputs to produce a single "simulation result" data product.

[0045] Capturing in the Ecosystem Specification **230**, the input, intermediate, and final data sources along with the multiple applications that participate in the computing process allows field specialization to be broadened such that the flow of particular values of the invariants may be extended beyond the source code of a single application. Furthermore, field specialization broadening also enables across file RW (read / write) operations and even operations via network communication, such as passing the resulting data to an operating system such as Linux and to other destinations. In some embodiments, static and dynamic analysis may determine that an invariant within a specialization opportunity arose even earlier than an assignment to a C variable within the single application. Instead, the invariant is extracted from a data component at a particular location within an input, such as the oil field data or the config parameters. Once such a value flow is determined, spiffs may be created in the appropriate applications. When an oil field data file is given, the spiffs is executed to produce speccode of a specialization opportunity within the oil field simulator application. This speccode could be associated with the particular oil field data file, to be loaded dynamically into the oil field simulator application (or passed to the Linux OS **150** or stored in the IBM cluster **160**) when that application is run. In this way, all of the applications involved in a computation can participate and be optimized, via a broadened conception of field specialization.

[0046] At least three exemplary extensions may be implemented for field specialization broadening.

[0047] The first extension to field specialization is **cross-application value flows** (where a value transfers out of one application and subsequently into another application), specified by the ecosystem spec for how data travels from one application to another. Within a single application, as viewed by field specialization, a value flows from one C variable to another via a data copy, implemented with low-level load and store operations. With the availability of an ecosystem spec, additional kinds of transfers can be accommodated in the invariant flow analysis. For instance, one such transfer is passing a value as a parameter (which can be a structure) to an operating system call and receiving a return value from a system call can establish data flow between an application and the OS. In addition, the ecosystem spec can support specification of other means of communication, such as exchanging data across devices. For example, a graphics processing unit (GPU) may communicate with a central processing unit (CPU) over a memory bus, with values copied via network communication channels, such as sockets. The ecosystem spec annotates a plurality of connecting paths with the means for such value transfers and the Spiff Toolset analysis utilizes such transfers to extend the value flows, thereby identifying additional cross-application value flows and additional specialization opportunities within each of the participating applications.

[0048] The second extension to field specialization is an **inter-application analysis**. The static and dynamic analysis is performed by the Spiff Toolset not just on the source code of a single application, but also across the data read and written by that application. That analysis may be similarly extended to other applications or operating systems invoked by an application, such as the Linux operating system, or storing invariant values, such as within an IBM cluster, shown in Figure 1. A particular value within the oil field data flows into the oil field simulator application, then flows into the Linux operating system as a system call parameter or as data passed through a system call, then flows into the IBM cluster. This same value may flow back into the simulator from the Linux O/S as an operating system return value or data value returned. A complex data flow may travel back and forth between the three applications illustrated here, across different system calls and within different C functions in each of these applications. Inter-application analysis combines the per-

application value flow analysis, performed by the Spiff Toolset, with analysis of data read by application or sent from one application to another. Inter-application analysis need to understand the semantics of the cross-application value flows and be able to combine value flows within a single application to compute value flows across applications. Furthermore, inter-application analysis is able to determine where the speccode that is associated with particular invariants can be best instantiated.

[0049] The third extension to field specialization is **invariant cross-application termination**, verifying the possibility of an invariant originating in an application (or data source or product) and terminating in a specialization opportunity in a separate application. This enables flexibility in placing the spiff, in associating speccode with a data source or product or application, and in shipping the speccode to the destination application. The Spiff Toolset can determine where the speccode can be best stored and subsequently communicated to the relevant application where it will be installed. The Spiff Runtime Environment needs mechanisms to handle the variations enabled by the structure provided by an ecosystem spec.

[0050] FIG. 3 is a block illustration of a field specialization broadening process involving the three extensions just described, in accordance with an embodiment of the present invention. Comparing to the field specialization process **100** for single application shown in Figure 1, the field specialization broadening process **300** comprises additionally a router **370** (a hardware device that now comes with significant computation capacity) and a cloud service **380** (including data analytics). The router **370** receives inputs from the Linux operation system **150** and passes the inputs to the cloud service **380**.

[0051] The stars in Figure 3 represent speccode created by spiffs (not shown) that were added through the broadened field specialization process. The spiffs are added to utilize the value flows described above. The field specialization process determines where the Speccode should be stored. For example, speccode **310** may involve invariants both from the oil field data **120** and the oil field simulator **130**. Speccode **320** may involve invariants both from the config params **110** and the oil field simulator **130**. Speccode **330** stored in the Linux operating system **130** may involve invariants from the simulator and oil field data. Speccode **340** stored in the router **370** and Speccode **350** stored in the cloud **380** may also involve invariants from the simulator and the oil field data.

[0052] The speccode stored within the simulator may originate from the oil field data and from the simulator itself, and may be identified through basic field specialization. Other spiffs are stored with the operating system, router, and cloud, specializing code found in the indicated applications. In some embodiments, speccodes may flow from where they are stored to where they may be invoked (the application that provided the specialization candidate from which they were subsequently specialized). For example, the oil field data may store the router speccode **340**. In some embodiments, speccode identifiers can reside with data or with applications and can also be included in communications with subsequent applications, indicating the relevant speccode to (later) invoke.

[0053] Below are two exemplary fields which the original oil field specialization may be broadened to cover.

[0054] **Web applications:** A backend web server is responsible for processing input data obtained from user activities at the front-end, i.e., in web browsers that run javascript applications. The user inputs can vary and hence present different patterns within the data collected at the backend. The backend server can then produce client (javascript) speccode for different value patterns that can then be pushed to the (appropriate) browsers.

[0055] **Data center work distribution:** A distributor for work packets is common in big data applications (such as Hadoop). The distributor is an application that can parcel out the work based on certain attributes, which can then serve as invariants to specialize the tasks running on different worker nodes. In this case also, the spiff and the speccode it generates is in an application separate from the invoked application.

[0056] FIG. 4 is an illustration of a flow process of broadened field specialization with an embodiment of the present invention. In broadened field specialization, there is a sequence of decisions to be made. The flow process of broadened field specialization **400** shown in Figure 4 comprising a plurality of steps to address the sequence of decisions. Step **410** is deciding what (extended) invariants are present; Step **420** is deciding which of these extended invariants should be specialized upon; Step **430** is deciding what code sequence within a specialization opportunity should actually be specialized; Step **440** is deciding when to perform the specialization (and thus, within which application); Step **450** is deciding where to store the generated speccode, and finally, Step **460** is deciding how to transfer the speccode to the site

(application) where it will be invoked. Although the steps shown in Fig. 4 are related to broadened field specialization, it is understood that the steps are also applicable to conventional field broadening, since the steps are necessary for field specialization in general.

[0057] FIG. 5 is an illustration of an alternative flow process of broadened field specialization with an embodiment of the present invention. Compared to the field specialization process shown in Figure 4, step **510** in Figure 5 is extended to invariants that flow from external data or external applications; step **540** in Figure 5 is extended to spiffs located in an application separated from the application containing the source of the invariant that is being utilized by the spiff, or even autonomous from any application, and steps **550** and **560** in Figure 5 are extended to speccode stored separately from the application being specialized.

[0058] The flow process of broadened field specialization disclosed within this disclosure may be stored as computer executable instructions within a non-transitory computer readable medium and executable by a processor of a computing device. In some embodiments, the specialization broadening process may be executed by a plurality of computers with different computers executing different process steps.

[0059] FIG. 6 is an illustration of field specialization for elaboration a paradigm of computer science with an embodiment of the present invention. The diagram comprises four quadrants **610, 620, 630** and **640** for scenarios of data represented as data, code represented as data, data represented as code, and code represented as code respectively.

[0060] In the early stages of computer architecture, from the Babbage machine through the 1930's, data was differentiated from code. The data was what was manipulated and the program code was the instructions for how to manipulate the data to effect a computation. This is represented in quadrants **610** and **640** in Figure 6 as data represented in binary format in a computer memory or storage device, that is, data stored as data, and source code represented in some other way (e.g., patch cords), that is, code represented as code.

[0061] In the 1940's, John von Neumann proposed a revolutionary architecture that stored the program in machine code in the computer's memory as numbers, mixing code and data. (Indeed, the code can be manipulated as data, and even

changed during the running of the program.) This architecture is represented in quadrant **620**, with code (machine instructions) represented as data.

[0062] In the 1960's there were some initial forays into combining code, in the form of a Lisp function, with data, say the value of one of the arguments, to produce a List continuation, which is a Lisp function (code) paired with the value of that parameter, which is just a function with one less argument. This is in a very particular way that data are stored/encapsulated in code, as represented in quadrant **630**.

[0063] In the 1980's the Postscript language was invented. This was code that when executed would create an image. Postscript is produced by a formatter, taking a document such as a Microsoft Word file, which is data, and converting to a program, again, code as data, as represented in quadrant **620**.

[0064] Field specialization takes this idea further. Field specialization takes the values of invariants, that is, data, and uses these values to create a specialized code version of a portion of an application, such as a DBMS, which is code that can be executed. So a relation speccode is the result of specializing DBMS code using the schema of a relation (data). A tuple speccode is the result of using the data values within a tuple (row of a table). An O/S speccode is a specialization of a snippet of an operating system based on particular data values of particular invariants within that snippet; ditto for router speccodes.

[0065] This data-represented-as-code (as represented in quadrant **630**) can be created in one application from a snippet in that application or another application, passed around between applications, and invoked by the target application when appropriate. The field specialization technology provides the means for identifying when such speccode are effective in increasing performance, when they should be created, with which invariants should they be specialized upon, how they can be communicated across applications, and when they should be invoked.

[0066] The implication is that for any coherent region within a data file, it is possible to ascertain the invariant values within that region, follow those values into areas of the application code, and then make speccodes out of those areas, then associate those speccodes back to their regions. This perspective thus focuses on the originating data, rather than starting with the code and specializing it, which then opens up new opportunities and technology.

Extension 4: Introducing Code into a DBMS at Run-Time While Minimizing Cache Pressure

[0067] Disclosed below is a fourth extension for field specialization broadening process besides the three extensions described above. The fourth extension involves introducing code into a DBMS at running-time while minimizing cache pressure simultaneously.

[0068] Instruction cache misses have contributed in the past to the execution time for DBMSes. This problem can arise again in the presence of code added at run-time, say in the context of micro-specialization, when a specialized version of an often-executed portion of the DBMS can speed query evaluation. In cases where this run-time code causes even a small increase in the I-cache miss rate, the benefit of such optimizations may be obviated. Judiciously placing newly-generated code in main memory can result in a substantial reduction in execution time.

[0069] A related challenge is that the code placement algorithm should not itself introduce much overhead. This implies that classic code placement algorithms commonly used at compile time are generally too slow to be used at run time to place dynamically-generated code.

[0070] As long as there has been a memory hierarchy (e.g., in Fig. 7), there have been strategies for minimizing the performance hit of accessing the slower (and larger) portions of this hierarchy within a DBMS. Initially the battle was on minimizing disk I/O latency. This challenge was addressed through a combination of sophisticated disk-based data structures such as B+-trees and hash indexes and sophisticated query evaluation methods such as sort-merge join. Later attention has been moved to the data caches in modern architectures.

[0071] FIG. 7 is an exemplary cache hierarchy with an embodiment of the present invention. As shown in Fig. 7, the cache hierarchy **700** shows a communication between a CPU **710** comprising CPU cache(s) and a main memory **740**. A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction cache (I-cache) and data caches (D-cache), where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.). As shown in

Fig. 7, the CPU **710** comprises one or more cores **720** with each core having its own cache hierarchy. Each core **720** comprises a L1 I-cache **722** and a L1 D-cache **724**, which are coupled to a L2 cache **726** inside the core. An L3 cache **730** couples to L2 cache **726** of each core and also couples to the main memory **740** for date /instruction exchange.

[0072] In the method of *micro-specialization*, small often-executed functions within query evaluation are specialized into smaller versions called *bees* (equivalent to "spiff" as used herein, e.g., an instantiated bee is equivalent to "specialized code" as used herein, which is the result of a spiff) that are appropriate just for that specific query, realizing a significant improvement in performance. This performance benefit, though, may come with an unfortunate increase in I-cache conflicts (termed *cache pressure*) and thus these approaches may actually experience performance degradation due solely to these cache misses. The reason is simple: even a small increase in number of I-cache (instruction cache) misses, say if 1% of instruction accesses do not hit the first-level instruction cache, the required subsequent accesses of the secondary cache and possibly of main memory will result in a substantial performance degradation in query run time, due to the significant I-cache penalty in modern architectures (that of 10 or more cycles for an L2 access and perhaps hundreds of cycles for a main memory access). Therefore, minimizing cache pressure is an important consideration when code is dynamically generated within a DBMS.

[0073] In this disclosure, a variety of placement algorithms are disclosed that mitigate the I-cache pressure in the presence of run-time code generation within a DBMS. These algorithms use a combination of off-line profiling and on-line code placement that takes into account the query plan, to compute a novel kind of profile called the Slot Pressure Profile, which estimates the pressure at each cache slot. The MaxResidency algorithm, which achieves the best performance among these placement algorithms, first collects the profiles of operators off-line, then examines the query plan and estimates the I-cache's pressure by taking into consideration all involved operators. Finally it determines the memory location for the generated code that will be least harmful to I-cache performance.

[0074] In this disclosure, the following features/methods will be disclosed in details to minimize cache pressure when introducing code into a DBMS at a run time:

[0075] 1. Differentiating profile-guided placement algorithms from algorithms that do not use profiles. It is shown that the extra information within several kinds of profiles is helpful for placement.

[0076] 2. A novel kind of profile, termed the ***Slot Pressure Profile*** (SPP), is disclosed. SPP associates a value with each cache slot, such that a slot with higher value is more likely to incur cache pressure when the address of code added at run-time maps to that slot. It is fundamentally different from a traditional ***Call Graph Profile***, which captures only the interaction between functions, from which a statically-determined layout attempts to infer the cache pressure. Several ways to compute the SPP are also presented to better predict cache pressure.

[0077] 3. A hybrid profile-guided placement algorithm is disclosed that performs the expensive profiling off-line and does on-line profile computation from the off-line profiles very efficiently.

[0078] 4. A way to detect stages within the execution of a query via its query plan is disclosed. This allows better dynamic code placement by utilizing information concerning the structure of the query.

[0079] It is showed that each kind of information listed above contributes to the effectiveness of the placement. The MaxResidency algorithm combines all of these sources of information to place a large of number of bees that add little overall cache pressure, thus demonstrating that cache pressure is no longer a concern when considering aggressive micro-specialization.

[0080] To verify the effectiveness of this algorithm, an experiment is ran on a 32KB, 128-slot, 4-way-associative I-cache and measured cache miss rate with cachegrind. With 200 dynamically-inserted bees running within PostgreSQL, the I-cache miss rate was reduced from 2.02% (when the bees were placed by a naive replacement algorithm) to 0.101% by the disclosed MaxResidency algorithm. With 400 bees, the I-cache miss rate was reduced from 3.33% to 0.144%.

[0081] In embodiments, the disclosure concerns the efficiency challenges of a confluence of three generally disparate threads: **a) dealing with cache pressure, b) handling large amounts of data via a DBMS**, and **c) just-in-time (JIT) compilation**. In the following, related previous work on these threads in isolation and in various combinations is reviewed.

Related Previous Work

[0082] When hardware caches between main memory and the CPU first appeared, additional hardware was introduced to improve their efficiency. This engendered work on better exploiting this hardware, including the *next cache line* and *set predictor* approaches, where pointers are used to find the target instruction of branches; alternative approaches of LRU; a proposal for a *cache miss lookaside buffer* where cache references are first stored in the LRU segment, to be promoted to a *hot* segment if they are referenced often; and *predictive instruction fetch*, where history information is used to predict the correct path of future instructions.

[0083] Software approaches usually require profiles that reflect run-time information. Notable examples include the IMPACT C profile-accepting compiler; *profile guided code positioning*, where functions are aggregated based on a weighted function call graph; *procedure mapping*, where the occupied cache lines are colored to guide the placement of remaining functions; using a *temporal relation graph*, where weights in the call graph predict cache misses accurately through the simulation of a full-associative cache; *code reordering* based on labels and call graph, where the call graph is traversed as a tree and loops are labeled to avoid conflicts; using a *conflict miss graph*, where edges in the graph represent worst-case number of misses; and *code size reduction*, where the overhead of code padding is reduced without cache miss increase. Modern compilers that employ one or a suite of these approaches are now very effective at ensuring that cache misses are a small component of the running time, with almost all instruction and data accesses being handled by the cache.

[0084] The database community initially was concerned with the upper portion of the memory hierarchy, between main memory and disk, as a database generally did not fit in main memory. But more recently, data cache optimization (be- tween the main memory and the data cache) has been explored in detail, including *data structure transformations*, where data accesses are aggregated into chunks whose size is related to platform cache size in various ways; *cache-conscious data structures*, such as search trees that take cache size into consideration; *result buffering for operators*, where results from child operators get aggregated based on the type of parent operators; *PAX*, which adopts column storage to avoid loading unused data into the cache; and *data morphing*, where query workload information is used to determine the attribute layout.

[0085] Following those notable results researchers then started paying more attention to the instruction cache, including *OLTP code layout optimization*, where Pettis & Hansen's algorithm [K. Pettis and R. C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, June 1990] is applied to database and web servers [A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. *SIGARCH Comput. Archit. News*, 29(2):155–164, May 2001]; *STEPS*, where common instructions are identified and kept cache-resident so that a new thread does not have to reintroduce them [S. Harizopoulos and A. Ailamaki. Improving instruction cache performance in OLTP. *ACM Trans. Database Syst.*, 31(3):887–920, Sept. 2006]; *call graph prefetching*, in which fu- ture instruction sequence is predicted by the call graph and prefetched to avoid misses in a DBMS [M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. *ACM Trans. Comput. Syst.*, 21(4):412–444, Nov. 2003]; and *multi-core optimization*, where queries are assigned to different cores based on the resources they need and the resources avail- able at each core [R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. *International Journal on Very Large Data Bases*, 2(1):373–384, Aug. 2009].

[0086] All of the work just listed makes the assumption that the program is static, which has been characterized as the "com- pile once and run forever" paradigm. Dynamic code generation, however, works in a "code inserted at run-time" fashion. This topic is more often discussed in the context of just-in-time (JIT) compilation, first introduced to increase the efficiency of "run-anywhere" byte-code interpreters.

[0087] One major research topic of JIT compilation is to improve the cache performance when instructions are generated and mapped to different cache slots. State-of-the-art results include: *code reordering*, where the virtual machine separates code and data space, rearranges methods into hot and cold portions, does code padding for frequent caller/caller pairs; *cache line reservation*, where some cache lines are reserved for several particular objects to avoid the eviction of these objects; *partial code compilation*, where dynamic analysis of control flow is utilized to avoid optimizing large functions; *code tiling*, where the call graph is divided into components that each fit within one cache slot in a greedy manner; *JIT co-allocation*, where a

compiler co-allocates neighboring procedures in the call chain rather than procedures that frequently call each other; *conflict matrix coloring*, where a conflict matrix records the potential conflicts for a sequence of instructions and coloring is done based on the matrix; *locality grouping*, where the information of interaction graph and sequence of instruction is combined to group and reorder methods for co-allocation; and *phase prediction*, where the information of reuse distance is used to identify phases and locality in each phase.

[0088] As discussed in details in Section: **COMPARISON TO PREVIOUS WORK**, the above work either (a) doesn't solve the real problem, (b) is too expensive for use in DBMSes, (c) does not apply to I-cache pressure, or (d) does not contend with the complexity of query execution code.

[0089] The JIT approach has made its way into databases, in two parallel guises. One way is *micro-specialization*, in which frequently-executed functions containing branch instructions on identified invariants are specialized at run-time, with the newly-generated code read in (again, at run-time) to improve performance. The other way is *query compilation*, where the DBMS query is compiled into machine code at run-time, then that machine code is loaded and execute. The focus of the disclosure is the former, but the placement algorithms disclosed are applicable to the latter.

[0090] Micro-specialization speeds up a DBMS by specializing a general function, using information not known at compile time. For example, the number of columns in a specific table cannot be known at compile time, so a general function must be able to handle all possible numbers of columns. But the number of columns in the table is known and unchanged (except for ALTER TABLE commands) once the table is created. If this constant is hard-coded into a new function and replace the old one, at least the cost of passing that one parameter is saved, along with any branches on that value. Other invariants, such as the type and size of each column and whether each column is nullable, can also be specialized. However, this can only happen after the DBMS is compiled and the table is created, because the number of possible combinations is simply untenable.

[0091] Then when a query is executed on a particular table, that table's specialized function can be loaded into main memory. Additional specialization

opportunities arise from in- variants located in individual rows, in queries, and elsewhere throughout the DBMS.

[0092] Because of the way micro-specialization works, the DBMS source code and specialized code are not aware of each other, and the benefit of a compiler taking care of code layout for the dynamically-created code may be lost.

[0093] The disclosure of this invention is to address the problem of (a) cache pressure in the context of (b) a DBMS utilizing (c) JIT techniques. The focus of the disclosure is on micro-specialization but the techniques also apply to some kinds of query compilation. In either case, it is desirable to retain the benefits of (i) main-memory/disk performance improvements utilized by conventional DBMSes, (ii) cache pressure reduction techniques commonly utilized by conventional compilers, (iii) any cache pressure reduction techniques utilized by modern DBMSes, and (iv) performance improvements realized by recent JIT approaches, specifically, micro-specialization, thereby achieving a DBMS that is highly efficient with regard to disk accesses, data cache pressure, and instruction cache pressure.

Preliminaries

[0094] During the execution of a program, each instruction executed is first fetched into the CPU cache. The slot in the cache (whether the first level instruction cache, the second level integrated cache, or even the third level integrated cache shared across CPUs) where an instruction is located, namely a cache line, is determined by the instruction's ad- dress in the virtual memory space. When multiple instructions sharing the same cache line are executed, cache conflicts may occur, resulting in expensive cache-miss penalties (cache-line eviction and instruction fetching from slower levels in the memory hierarchy.

[0095] Within DBMSes, query evaluation usually involves executing a large number of instructions over many iterations. This substantial footprint of query evaluation code can potentially lead to performance degradation due to high instruction-miss penalty. While compilers such as *gcc* apply basic-block reordering techniques to improve code locality thus reducing instruction cache misses, the bee code does not benefit from this optimization in that bees do not exist when the DBMS is compiled.

[0096] A 3C model [M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38(12):1612–1630, Dec. 1989] provides terminology for the types of cache misses in each situation:

- A *compulsory miss* occurs for the first access and can- not be eliminated.
- A *capacity miss* occurs when the cache simply can't fit all the accessed memory locations; such misses can be reduced with a larger cache, in which the accessed memory locations associated with a particular cache slot get spread across several slots in the larger cache.
- A *conflict miss* occurs when an entry that was present in the cache gets evicted by an access that would have associated the same slot even with a larger cache; such misses can be reduced by a higher cache associativity.

[0097] Several additional terms are also introduced in this disclosure.

- A function F *associates* a cache slot S if the memory address of any piece of F's machine code maps to S.
- The working *set* of a collection of functions to be executed is defined as the collection of the distinct memory blocks touched over the execution of a sequence of instructions. In this disclosure, the working set at the cache level is of particular interest. The *cache slot working set* refers to the memory blocks that are in that same working set and map to the same cache slot.
- The relationship between a slot's working set size and the cache's associativity is relevant here. If the cache's associativity is less or equal to the size of that slot's working set, a slot is *crowded.*
- A *bee* is a specialized version of a DBMS function that is created and loaded into memory at run-time. Such functions may occupy one memory block, multiple contiguous blocks, or multiple non-contiguous blocks.

[0098] To illustrate the cache effects of executing a bee during query evaluation, Figure 8 presents a study of the relationship between various placements of a single bee (indicated by the x axis) of a bee and the percentage increase in I-cache misses

(shown by the *y* axis), during the evaluation of a query. As described later, the experiment machine is equipped with a 32K I-cache. Given that each cache line is 64 bytes wide and the cache is four-way set associative, there are a total of 128 cache slots, each consisting of four cache lines. 16 placements are sampled uniformly across all 128 possible cache slots. In other words, the same query is evaluated 16 times, each time with a different placement of the bee executed during each query evaluation. As indicated by the figure, when the bee is placed at cache slots addressed at 0x10 and 0x40, respectively, the I-cache miss rate is minimized. However, if the bee is placed at 0x68, a 130% increase in number of cache misses is observed. Overall instruction-cache (I-cache) performance is thus very sensitive to the placement of the bee. As disclosed above, the increase in cache misses results in a substantial increase in overall query evaluation time.

[0099] FIG. 9 is an illustration of Cumulative Cache-Line Reference histogram with an embodiment of the present invention. To avoid placing a bee at slot 0x68 in Figure 8, Fig. 9 suggests determining for a given query the cache pressure at each slot, and realizing a histogram and placing bees in the *valleys* where there is little pressure. In this particular case, the attractive valleys are at ox07–ox0f and ox56–ox60. A clever scheme will be introduced later.

[00100] There are three possibilities describing how the existing DBMS code interacts at the I-cache level with a bee that has just been loaded into memory, when the existing code and the bee associate the same cache slot.

1. The existing DBMS code has a small working set at this slot, so doesn't use up its cache capacity. In this case, the bee only causes compulsory misses, introducing little cache pressure.
2. The existing DBMS code has a large working set at this slot, so uses up the cache capacity and experiences both capacity and conflict misses. In this case, instructions in the bee evict existing DBMS code and cause an increase of conflict misses. Such behavior is called a *DBMS-bee conflict*. This case is fairly predictable, and indeed algorithms will be provided that do a good job of avoiding such conflicts.
3. The existing DBMS code has a medium working set at this slot, so uses up the cache capacity yet experiences no conflict misses. In this

case, because the DBMS will revisit its existing code as well as the bee frequently during query evaluation, instructions in the bee will evict existing DBMS code from the cache and thus perhaps cause a sharp increase of conflict misses. This is the challenge to be addressed.

[00101] What if multiple bees need to be placed? If several bees associate the same slot, DBMS code that associates this slot may cause cache misses, and thus, may exhibit DBMS-bee conflict. If successive bees also associate this slot, those bees could conflict with each other, termed *inter-bee conflicts*. Both kinds of conflicting bee behavior can be avoided by taking into account how crowded a slot is when deciding bee placement. Thus the problem of predicting cache pressure is reduced to the problem of ascertaining the size of working set at each cache slot.

[00102] The average cache miss rate of TPC-H queries with a naive placement rises to 0.912% with 200 bees, 1.66% with 400 bees, and 2.95% with 1000 bees. This is of concern when a cache miss is more than 10 times slower than a cache hit. A more sophisticated placement algorithm is required.

[00103] Disclosed below are the placement algorithms in two categories: profile-agnostic and SPP-guided, based on the information they use.

Profile-Agnostic Algorithms

[00104] Profile-agnostic algorithms may be relatively simple in terms of design but generally offer mediocre performance. They are introduced as warm-up and further opportunities are explored for optimization.

[00105] FIG. 10 shows a block diagram illustrating workflow **1000** of profile-agnostic bee placement algorithms with an embodiment of the present invention. As shown in Figure 10, the workflow comprises a slot selection phase **1005**, a bee insertion phase **1010**, and a query evaluation phase **1015**, all of which occur while the DBMS is running. The slot selection phase **1005** and bee insertion phase **1010** can be done at query optimization time just before evaluating the query. The slot selection phase **1005** receives static information, specifically the machine code of the DBMS, which provides the address of the caller of each bee, and the bee code, which includes the name of its caller, to decide the slot to associate, then it passes that slot number to the bee insertion phase **1010**, which finds an open memory address that maps to that slot, places the bee and checks if all the bees have been placed. In some embodiments, the bee insertion phase **1010** feedbacks a next bee to the slot selection

phase **1005** to start the slot selection process again. In some embodiments, the query evaluation phase **1015** actually performs the query, calling whatever spiffs were created for that query, and then sends a next query to the slot selection phase **1005** to start the slot selection process again.

[00106] These algorithms do not utilize information about the query to be executed. Because of the limited information utilized, they are not expected to perform well under every situation. Yet they serve as a good starting point to explore the problem due to their simplicity.

[00107] **Slot Selection phase 1005:** This phase yields the slot that the bee's address should map to. In some embodiments, this phase involves a *SlotZero Algorithm* and a *NextToCaller Algorithm*. *SlotZero Algorithm* always returns a location that maps to cache slot zero. This algorithm is expected to perform terribly for the following reason: both inter-bee conflicts and DBMS-bee conflicts are going to occur, as long as there are more than four bees (the assumed slot associativity). *NextToCaller Algorithm* first finds the address of the bee's caller function by examining the machine code. Then it finds the slot this address maps to. This is called the *caller slot*. Next, this algorithm calculates the size of bee in terms of cache lines and returns the slot that is immediately before the caller. Specifically, bees are grouped with their callers. *NextToCaller Algorithm* reduces inter-bee conflicts in comparison with *SlotZero*, yet the DBMS-bee conflicts remain unaddressed.

SPP-Guided Algorithms

[00108] In this section of the disclosure, several placement algorithms that utilize a slot pressure profile (SPP) are introduced.

[00109] **Slot Pressure Profile Taxonomy:** A Slot Pressure Profile is an array of integers, one for each cache slot in an identified cache. It is named with the following format: "*per-<granularity> <source> <cache> <what> slot pressure profile*" (or SPP).

granularity indicates at which granularity the numbers are collected. There are five possible granularities: *function* (over the execution of a function), *operator* (over the execution of an operator node in the plan tree), *stage* (over the execution of several operator nodes), *query* (over the execution of the whole plan tree), or *workload* (over the execution of several queries).

source indicates where the numbers come from. A *binary SPP* contains values that are calculated from the binary executable without running it. A *raw SPP* contains values that are output from a profiling tool during the execution of a specified workload. A *reference SPP* contains values that are computed over other SPP(s) by an algorithm.

cache indicates which cache the numbers refer to. It can be (level 1) *I-cache*, (level 1) *D-cache*, or (integrated) *L2-cache*.

what indicates what the numbers represent. In general, they all represent the likelihood of cache pressure at this slot. Details of how and why a specific value is chosen when each algorithm is introduced will be reviewed in details. For example, in disclosure of SPP Generation Phase below, an algorithm is introduced that uses an *function count SPP*, in which each slot is associated with a count of the functions that contribute cache pressure to that slot.

[00110] As examples of the full names of such profiles, the *per-function binary I-cache function score SPP* and the *per-query reference I-cache active-set SPP* are disclosed in two of the algorithms below.

[00111] **Workflow of SPP-guided Algorithms**: FIG. 11 shows a block diagram illustrating workflow **1100** of SPP-guided bee placement algorithms with an embodiment of the present invention. The workflow **1100** comprises six phases: a SPP generation phase **1105**, a SPP processing phase **1110**, a Candidate ordering phase **1115**, a slot selection phase **1120**, a Bee insertion phase **1125**, and a query evaluation phase **1130**.

[00112] In the SPP Generation phase **1105**, the algorithm generates at DBMS compile time one or more SPPs by assigning weights to a binary SPP with external knowledge from the programmer, collecting a raw SPP over the DBMS executing a workload, or computing a reference SPP. This phase takes time but is done statically, before the DBMS starts. The remaining phases are lightweight and occur *within the DBMS* as part of the query optimization phase, so that the bees are placed when query execution commences. Each algorithm creates its own reference SPP from the provided (intermediate) SPP(s) in the SPP Processing phase. Each algorithm then generates a list of *candidate slots* in the Candidate Ordering phase by analyzing the reference SPP from the previous phase. The candidate slot list contains a sequence of

slot numbers that will be used to successively locate bees (a slot could be present several times in this list). (The MaxResidency algorithm emits several lists; the rest emit a single list.) The Slot Selection, Bee Insertion, and Query Evaluation phases work as before.

[00113] The input to SPP Generation phase **1105** is the binary of the DBMS. The output is a single intermediate SPP. In embodiments, the SPP Generation phase **1105** involves three distinct SPP generation algorithms, each utilizing different input workloads.

[00114] 1. Binary Function Score Algorithm

[00115] In this SPP generation algorithm, a simple heuristic is used: often-used functions are the focus, as indicated by the workload. Often-used functions are chosen as fifty functions with the highest exclusive instruction count—that is, without including its children—as measured by callgrind. *Callgrind* is an analysis tool that counts function calls and the CPU instructions executed within each call and builds a function *callgraph*. *Callgrind* may be used for both CPU and cache profiling. One may define often-used functions in other ways. Those often-used functions are given a higher score, 100, to distinguish them from the less used ones, which are given a score of 1. The sum of the scores of all functions that include that slot are then assigned to each slot, to construct a *per-workload binary I-cache function score SPP*.

[00116] 2. Function Count Algorithm

[00117] There's a weakness in the above mentioned Binary Function Score Algorithm: it uses only global information, and therefore the specifics of the query to be eventually evaluated cannot be exploited. During query evaluation, a relational operator implemented by the DBMS (such as hash join) can invoke many functions, through a long call chain, through function pointers, etc. Hence, the first step in this algorithm is to identify all the functions that *may* be invoked by each operator. Because of the complexities of the DBMS that might confuse static analysis, dynamic analysis is used instead, though importantly, at DBMS compile time, to provide an estimate of these functions.

[00118] The input to this stage is the DBMS binary and the workload. The output is a single intermediate SPP per operator. The call graph for DBMS is first obtained for executing the representative workload, which involves the operators being cared about. Then a breadth-first search is performed for each query evaluation operator to

identify all the functions it may call. However, an operator *A* sometimes reaches a function *F* through another operator *B*; in that case, *F* shouldn't be considered to be reachable by *A*.

[00119] Then for each identified function, this algorithm generates a per-function binary I-cache function score SPP by assigning 1 to its associated slots. Adding up the SPPs for an operator yields a *per-operator reference I-cache function count SPP.*

[00120] 3. MaxResidency Algorithm

[00121] There are two problems with the above algorithm. The first is that both hot and cold code may exist within the same function, yet those slots are not differentiated, say, by using instruction counts. The second is even if the algorithm is modified to utilize instruction counts, the number for a particular slot would be an aggregate over those functions, and thus doesn't differentiate a single function with many instructions, which will generate only compulsory misses, from multiple functions, which may generate capacity and conflict misses. Thus in the first case, that number would overestimate cache pressure.

[00122] The input to this stage is the DBMS binary and the workload. In addition to one intermediate SPP per operator, the output has one more SPP involving the bee invocation code. It is first discussed how to compute the per-operator reference I-cache MaxResidency SPPs, that will be the one of the output. As pointed out in Section: Problem to be solved, knowing the size of working set at each slot is a great help in predicting cache pressure. However, this number is not available at DBMS run-time. To make matters worse, state-of-the-art profiling tools only give the aggregate counts for the target process at each *cache level*, which is too coarse-grained for the purposes. The number of counts and misses at the cache *slot* level are able to be measured. Even with this extension, the information is still insufficient for estimating the working set size in the presence of bees. A slot with a low number of misses may still be full, such that adding a bee to that slot will evoke many misses, due to DBMS-bee conflicts and perhaps inter-bee conflicts.

[00123] The MaxResidency algorithm makes a prediction of the size of the working set. In the following discussion, let *W* denote the associativity of the I-cache.

[00124] Define a *residency* of an address accessed within a particular execution (that is, of the workload), to be the number of times that that address was accessed before it was evicted. If an address was read into the cache multiple times, then each

section will contribute a residency, and this address will be associated with a *sequence* of that many residencies.

[00125] Define the *max residency* of a *slot* (not an address) to be the top W residencies of all memory addresses that map to this slot. The W residencies might not all come from different addresses.

[00126] From these W integers, compute an *estimated occupancy*, a number between 0 (no cache lines were occupied by any address, which will be highly rare) and $W+1$ (implying that there was a lot of thrashing at this slot).

[00127] The pseudocode of the MaxResidency algorithm is presented in Algorithms 1 and 2, using the max residency of each cache slot and instruction count SPPs produced by an extended cachegrind. Each instruction count SPP is collected over the execution of a specific DBMS query operator, hence it is a *per-operator raw I-cache Instruction count SPP*. The instruction count SPP and max residency of each cache slot are generated at the same time. From them, a *per-operator reference I-cache MaxResidency SPP* is computed. In some embodiments, an exemplary code for *MaxResidency SPP generation* is shown in Table.1 below.

[00128] Table 1 Algorithm for MaxResidency SPP generation

Algorithm 1 MaxResidency SPP generation

```
1: map<slot, list> values;
2: int[ ] MaxResidency_size(
          int[ ] instruction_countSPP,
          map<slot, list> max_residencies)
3: int[ ] MaxResidencySPP;
4: int numberOfSlots = MaxResidencySPP.length();
5: int instruction_threshold =
     instruction_countSPP.average() × pressure_threshold;
6: for i=0 to numberOfSlots do
7:    MaxResidencySPP[i] = estimate_occupancy(
          max_residencies[i],
          instruction_countSPP[i],
          instruction_threshold);
8: end for
9: return MaxResidencySPP;
```

[00129] Two parameters are used in this algorithm. The first, *pressure threshold*, is the ratio of the instruction count at a slot to the average instruction count of all slots.

This parameter is considered to constitute pressure for the associated slot. A value of 0.05 is used. A low value ensures most cache slots are further considered.

[00130] The second, *residency threshold*, is the ratio of the residency to the total instruction count at this slot that is considered long enough to occupy the cache line. A value of 0.15 is used. This value should not be greater than $1/W$ (for example, 0.25) to be meaningful. The lower it is set to, the more the algorithm considers a residency to be in the main loop. Note that these parameters are subjective. It will be seen that these specific values obtain good performance.

[00131] The MaxResidency algorithm first computes an instruction threshold, to identify slots that are accessed often enough to potentially incur cache pressure. Then it checks the max residency of this slot. If one residency accounts for 15% of the total counts at this slot, this residency is considered to be in the working set and *estimated occupancy* at this slot incremented by 1.

[00132] Table 2 Algorithm for single-slot MaxResidency occupancy

Algorithm 2 single-slot MaxResidency occupancy

```
1: int estimate_occupancy (list max_residency,
       int slot_total_count,
       int instruction_threshold)
2: int occupancy = 0;
3: if slot_total_count<instruction_threshold then
4:    return 0;
5: end if
6: for i=0 to W do
7:    if max_residency[i]/slot_total_count >
           residency_threshold then
8:       occupancy += 1;
9:    end if
10: end for
11: if occupancy == 0 then
12:    return W+1;
13: end if
14: return occupancy;
```

[00133] MaxResidency Algorithm is reviewed with three concrete examples. For simplicity, it is assumed that W is 4 and the cache has only two slots: 0 and 1, with a memory address mapping to a cache slot by last bit.

[00134] Example 1: instruction sequence with a small working set. For this first example, consider an instruction sequence S_1 consisting of ten repetitions of address 11 followed by address 21 (denoted 11, 21), both of which map to slot 1. Thus the size of the working set of S_1 at slot 1 is thus 2 because there are two distinct blocks. How does MaxResidency algorithm infer 2? It first sees that average instruction count is 10 (total instruction count / number of cache slots). It then sets the threshold value to be 0.5 (*pressure threshold* × average instruction count; cf. line 5 in Algorithm 1) to filter out slots with few accesses. Slot 0 has 0 (<0.5) instructions so gets 0 as *estimated occupancy* immediately (cf. line 4 in Algorithm 2). Slot 1 has 20 (>0.5) instructions so should be further processed. The max residency of slot 1 is <10, 10 , 0,0>. 10/20 is larger than the *residency threshold*, so the algorithm concludes that slot 1's *estimated occupancy* is 2 (cf. line 8 of Algorithm 2).

[00135] Now consider S_2, a variant of S_1 with successive accesses to the same memory block: an instruction sequence of ten repetitions of 11, 21, 21, 11, 11. The size of S_2 's working set is known as still 2. For this scenario, however, the instruction count is 50. The algorithm will thus compute an average instruction count of 25 and thus will filter slot 0 out. This time, the max residency at slot 1 is <30, 20, 0, 0>. 30/50 and 20/50 are both larger than the *residency threshold*. So it concludes that slot 1's *estimated occupancy* is 2. In both scenarios, MaxResidency's analysis is accurate.

[00136] Example 2: instruction sequence with a large working set. Consider ten repetitions of instruction addresses 1, 11, 21, 31, 41, 51. The algorithm will compute the average instruction count as 60 / 2 = 30 and so filters slot 0 out. This time, the max residency of slot 1 is <1,1,1,1>. This indicates that thrashing is probably going on at this slot! So it concludes that slot 1's *estimated occupancy* is 5 (cf. line 12 of Algorithm 2). However, the actual size of slot 1's working set is known as 6. This estimate is somewhat inaccurate but nevertheless warns that this slot is already experiencing many conflicts.

[00137] Example 3: instruction sequence with multiple loops. Consider another sequence $S3$ followed immediate by S_4, where S_3 is ten repetitions of instruction addresses 1, 11, 21 and $S4$ is twenty repetitions of instruction addresses 31, 41, 51, 61, 71, 81. The algorithm will compute an average instruction count of 75 and will filter slot 0 out. This time, the max residency at slot 1 is <10,10,10,1>. (The first is

residency of address 1, then address 11, then address 21; all others are 1.) Even the largest 10/150 is smaller than *residency threshold*. This indicates that the top *W* residencies do not come from the major loop. So it concludes that slot 1's *estimated occupancy* is 5. Based on external knowledge, the actual size of slot 1's working set is known as 7 (3+4). This example demonstrates that MaxResidency is not misguided by the small loop.

[00138] The result of this analysis is a *per-operator reference I-cache MaxResidency SPP* for each operator, stating the estimated occupancy of each slot, computed by running the DBMS on the workload at compile time.

[00139] Turning to the *per-workload reference I-cache bee invocation profile*, the second part of this phase's output. In the MaxResidency algorithm, the SPPs are collected over the same executable to make sure the layout doesn't change. This is realistic in that executables are not recompiled to accommodate bees. To obtain the workflow of stock DBMS in the layout with these added bee invocations, all bees are disabled when the SPPs are collected. Thus, there are three groups of code: the stock DBMS code, the bee invocation code, and the bee code itself. The per-operator reference I-cache MaxResidency SPP just obtained is not aware of the bee invocation code. The algorithm though needs to know this information.

[00140] An SPP for the bee invocation code can be created by identifying their machine code fragments and their slots. Note that only one such SPP is computed, termed the *per- workload reference I-cache bee invocation SPP*.

[00141] **SPP Processing Phase 1110**

[00142] SPP Processing Phase **1110** receives the intermediate SPP obtained in SPP Generation phase **1105** and outputs a reference SPP (or profile). In embodiments, the SPP Processing Phase **1110** comprises one or more algorithms selected from Binary Function Score Algorithm, Function Count Algorithm, and MaxResidency Algorithm. These algorithms infer the cache pressure, construct a reference SPP, and give the query plan. In embodiments, each algorithm calculates its reference SPP differently.

[00143] **Binary Function Score Algorithm**

[00144] The Binary Function Score Algorithm doesn't do much in SPP Processing Phase as the per-workload binary I-cache function score SPP is fixed for all queries.

[00145] **Function Count Algorithm**

[00146] The Function Count Algorithm takes into consideration that for each particular query, only the functions related to the plan's operators will be executed, and the number of times a function is executed is exactly that operator's cardinality. The algorithm constructs the reference SPP by multiplying the SPP of each operator in the plan by its input cardinality, then summing up these per-operator SPPs to get a final SPP, the *per-query reference I-cache inner-loop score SPP.*

[00147] **MaxResidency algorithm**

[00148] Many DBMSes use the so-called "Volcano" query evaluation model that delivers a result tuple by traversing nodes in the query plan tree, each returning a tuple for use in the parent node. Each node in the query tree is evaluated to produce a tuple, and so to determine the cache pressure for the entire query, the per-operator SPPs can be added up, for those operators appearing in the query plan, to obtain the SPP of the whole query.

[00149] However, there's another opportunity for optimization. PostgreSQL utilizes the concept of a *blocking operator*, which calls its child operators potentially many times, thereby computing the entire result of its subquery before returning its first result. Subsequent calls will just return successive tuples, without further evaluation of its child operators. Each blocking operator creates a *stage*, a subset of the operators in the plan that all execute to completion, while the rest of the operators wait. The only exception is that the root operator of the query plan tree always creates a stage even it is not a blocking operator. This provides an opportunity to estimate the cache pressure of each stage.

[00150] FIG. 12 is a schematic diagram **1200** showing stages of a concrete query plan tree with an embodiment of the present invention. In this figure, *hash* (**1222**) is emphasized as it is a blocking operator. As shown in Fig. 11, the diagram has a Stage 0 (**1210**) and a Stage 1 (**1220**). Stage 0 (**1210**) is rooted at the root hash *join* **1212** and also comprises a sequential scan operator **1214** (applied to table B) that is coupled to the hash *join* **1112.** Stage 1 (**1220**) is rooted at *hash* **1222** and also comprises a sequential scan operator **1224** (applied to table A) that is coupled to the hash **1222.** The root hash *join* **1212** couples to the root *hash* **1222.** The operator node where a bee is invoked the *bee operator* and the operators that are in the same stage as the bee operator are termed the *active operators.*

[00151] Note that there well could be stages *below* this blocking operator; these stages will be executed in totality during the first call of the blocking operator, but will not generate cache pressure after that, and thus do not contribute cache pressure for this stage. Similarly, for the stage *above* the blocking operator, which ultimately calls the blocking operator, once the lower stage returns, the active operators in that lower stage will no longer contribute cache pressure for the upper stage. Similarly for the stages *beside* this stage, in the query tree. Hence, only operators within the stage, the active operators, contribute cache pressure.

[00152] The MaxResidency algorithm first identifies the stages within the plan tree for the query, computes an SPP for each stage, and then finally uses those SPPs to compute an SPP for each bee caller, which is then used to place bees to be invoked by that caller. There are several subcases to consider:

[00153] Subcase 1: If a particular bee is invoked (either directly or indirectly) in a single stage, the SPP associated with that stage is then used to place that bee.

[00154] Subcase 2: It may be the case that two instances of the same operator within a query plan each call a bee, but each bee would be specific to that operator, and so in this case there are two specific bees, each associated with one of those operator instances. Each of those bees would be placed using the associated per-stage SPP.

[00155] Subcase 3: It may also be the case that a specific bee, specialized from particular invariant values, may be invoked (perhaps indirectly) by multiple operator nodes. In that case, this algorithm will conservatively treat the combination of the requisite SPPs of those stages as the resulting SPP: each slot would have the *maximum* of the associated slots of the constituent SPPs. The rationale is that each stage will execute separately, and so the bee would see only the pressure from one stage at any point in its execution.

[00156] Recalling the bee invocation code in Section MaxResidency Algorithm in SPP generation phase, because operators within the same stage can execute the bee invocation code without conflict and operators in different stages do conflict, the bee invocation SPP should be added to the per-operator SPPs to render a per-stage reference I-cache MaxResidency SPP for each stage.

[00157] **Candidate Ordering Phase 1115**

[00158] The Candidate Ordering Phase **1115** accepts a reference SPP from the previous phase and produces a candidate slot list by sorting the values in the SPP. (For MaxResidency, this is on a per-stage basis.) Algorithm 3 finds the next candidate slot from a SPP. Assuming that the bee takes consecutive blocks, Algorithm 3, as shown in Table 3, will choose the starting slot for the next bee, a slot that exhibits the least estimated cache pressure, according the SPP.

[00159] The inputs to this algorithm are the *beeSize*, which is the number of 64-byte blocks needed to accommodate the bee, and *usedSlots*, which contains slots that have been occupied by earlier bees; this set starts out empty. To find the next slot in which to accommodate a bee, the algorithm calculates for each slot its estimated cache pressure: the maximum value of *slotsTaken* consecutive slots following it, if the slot hasn't been used yet. (Repeating slots will be dealt with soon.) An array *extraImpact* records the estimated cache pressure of each slot. The minimum value in *extraImpact* (of this additional bee) is identified and its index is returned as the candidate slot number. Just before returning the candidate slot, the occupied slot(s) are added to *usedSlots* to avoid conflict with future bee placement.

[00160] In some embodiments, some implementation details are omitted in Algorithm 3 as they are not tightly related to the idea and are straight-forward. For example, a boundary check is needed near the end of array SPP, when locating a large bee in one of the last slots.

[00161] Algorithm 3 returns only the next candidate slot. To perform Candidate Ordering, this algorithm is successively invoked. The caller is responsible for resetting the *usedSlots* variable when it gets full.

[00162] This algorithm works fine for the Binary Function Score Algorithm and the Function Count Algorithm.

[00163] Table 3 Algorithm for SPPGuided Next Candidate

Algorithm 3 SPPGuided_NextCandidate

```
1: int SPPGuided_NextCandidate (int[ ] SPP,
       int beeSize, set usedSlots)
2: set affectedSlots;
3: int startingSlot;
4: int[ ] extraImpact;
5: int numberOfSlots = SPP.length();
6: for i=0 to numberOfSlots do
7:    if i ∈ usedSlots then
8:       continue;
9:    end if
10:   affectedSlots.clear();
11:   for j=0 to beeSize-1 do
12:      affectedSlots.add(SPP[i+j]);
13:   end for
14:   extraImpact[i] = affectedSlots.max();
15: end for
16: for k=0 to beeSize-1 do
17:   usedSlots.add(startingSlot+k);
18: end for
19: startingSlot = extraImpact.min().getIndex();
20: return startingSlot;
```

[00164] The MaxResidency Algorithm, on the other hand, requires some extra effort with the *usedSlots* variable to ensure that the right number of bees associate to each cache slot. As an example, assume the bee's size is 1 block, the cache's associativity is 4, and the MaxResidency SPP for that stage has 3 in slot A and 1 in slot B, which means slot A can accommodate one bee while slot B can accommodate three bees. The algorithm will first suggest using slot B then slot A. At this point, only slot A is needed to be removed, and continue. Slot B would be removed after it had been indicated three times. Once this occurs and slots are ran out, the process simply returns to generating a uniform sequence of bee locations for the remaining bees.

[00165] The evaluation involves several independent and several dependent variables. The hardware configuration is one of the independent variables. An I-cache configuration of a 128-slot, 64B-block, 4-way associative L1 cache (separate instruction and data caches) and a 8192 slot, 64B block, 16-way associative L3 cache (combined instruction and data) is adopted for both *cachegrind* simulation and for empirical evaluations on real hardware. (The physical machine has an additional intermediate-level cache: a 512 slot, 64B block, 8-way associative combined cache.)

The physical machine has 7 GB of main memory. The other independent variables are the training workload, the specific queries being evaluated, the size of bee code, the number of DBMS bee-invocation functions (50), the total number of invoked bees, and the placement algorithm used.

[00166] The dependent variables are the miss rates (that is, scaled by number of references, and thus is a percentage) at L1 I-cache and L2 levels, as those are the metrics that will be most impacted by bee placement, and the bee placement algorithm time.

[00167] The bee code used is a constant sequence of 10 instructions (that is, there is only one bee that is to be replicated as needed) that does not actually speed up the DBMS. The point is, a realistic bee improves DBMS performance, but this improvement is reduced by cache misses as a result of bad placement. Thus the actual bee code is not relevant to the purposes, only that it runs some code, thereby generating instruction fetches.

[00168] **Evaluation Experiment Protocol**

[00169] Fig. 13 shows an exemplary flow diagram **1300** for the evaluation. As shown in Fig. 13, the experiment protocol consists of three consecutive steps. Step **1305** is done statically, at DBMS compile time, and Steps **1310** and **1315** are done dynamically, during query planning.

[00170] The first step **1305** determines which functions will invoke a bee; such functions are termed the *bee callers*. The DBMS (PostgreSQL 9.3.10) are first compiled with gcc (version 4.6 used, with the -O2 option). Since the goal of micro-specialization is to optimize often-used code, the top fifty most-used functions are then identified within query evaluation from executing the training workload sequentially (TPC-H queries, omitting the five queries listed in the next section). For each so-identified function, a call is added in the source code to bee code (one or more calls, depending on the number of bees). A call graph profile is also obtained by running *callgrind* on that same workload. An *operator-caller mapping* is then generated by associating with each operator those functions that are called directly or indirectly by that operator. If two operators can reach the same function, the function will be associated with both operators.

[00171] The second step **1310** and third step **1315** are performed when the stock and specialized versions of PostgreSQL are ran, with the latter having the bee callers

invoke the specified number of bees, generating relevant performance statistics. Here the TPC-H queries are also used, so that the requested number of bees will indeed be called.

[00172] In some embodiments, a constant number of bees (400) is placed for each query to generate a significant but representative amount of cache pressure for the algorithms to contend with. Note that more than 512 bees could overwhelm the physical instruction cache (as that is the number of slots).

[00173] The second step **1310** occurs during query planning, for each query. It computes the number of bees needed to be called by each bee caller (an integer for each query), to ensure that the required total number of bees (an independent variable) are indeed invoked. Specifically, using the operator-caller mapping from the previous step, this step divides the total number of bees by the number of bee callers. (Functions that can be reached from multiple operators will be appropriately given multiple sets of bees.) When bees remain, that many bee callers are picked randomly to call one more bee. This implementation guarantees that each query will have exactly the same number of bees, though different sets of bees, determined by the operator(s) used.

[00174] Take query 6 as an example. This query utilizes 81 bee callers. Each bee caller is thus assigned at least 400 / 81 = 4 bees, with 400 mod 81 = 76 effective bee callers chosen randomly to invoke one additional bee.

[00175] In the third step **1315**, also during query planning, the set of bees to be executed for each bee caller associated with that operator, computed in the previous step, are placed in each node of the query plan. The selected placement algorithm is then invoked to place those specific bees (some of the algorithms utilize the query in this placement).

[00176] The focus is on I-cache miss rate, when bees are added to the stock query execution. Note that this is different from what will happen in practice, in two ways. First, each query will involve a different number and different set of bees. Due to the complexity of the DBMS, and the desire to have some uniformity in the experiments, the simpler arrangement of 400 identical one-slot bees are set up to be invoked by each query. Secondly, in proactive each query will *replace* stock code with a more efficient bee, whereas the bees are *added* to the stock code, which actually puts a greater bur- den on the placement algorithms, as there was more cache pressure with

the added bees. Given this experimental arrangement, it doesn't make sense to measure query time. Rather, the focus is on I-cache miss rate for each query, as that query invokes 400 added bees at 50 invocation sites within the DBMS.

[00177] In summary, the objective of the disclosed cache placement algorithms in this setting is to affect the minimum cache pressure, as indicated by I-cache miss rate. Using a placement algorithm that does so should also perform well in the more realistic setting of a variable number of variable-sized bees, as all the algorithms attempt (with varying degrees of success) to place the added bees in valleys identified by the profiles available to that algorithm.

[00178] **Algorithm-Specific Details**

[00179] In the experiments, each bee can fit in 64 bytes, or one slot. To evaluate the algorithms, TPC-H scale factor 1 is used as the representative workload omitting queries 2, 17, 20 and 21 because they ran too slowly, and omitting query 15 because that query is a view. Because cache performance are optimized, the scale factor is not relevant, as long as the data does not fit in the L2 cache, which is the case for scale 1.

[00180] For the SPP-guided algorithms, except for MaxResidency, TPC-H is also used, scale factor 1, as the training workload. This is of course unrealistic. This is done because it presents the best-case scenario for these algorithms, of having the training workload be the same as the evaluation workload.

[00181] For the MaxResidency algorithm, for the training workload, simple queries are manually created on a simple database, with each query ideally having a single operator, in order to generate the per-operator profiles. (In PostgreSQL, some operators always are associated with another, such as the aggregate operator is always associated with some kind of scanning operator. Collecting this information required additional modifications to cachegrind.)

[00182] Thus, *best-case* performance of other algorithms is compared to a realistic case for the MaxResidency algorithm.

[00183] **Results and Analysis**

[00184] The weighted and unweighted average I-cache miss rate across the TPC-H queries is first examined, provided in the last two rows of Table 4. The unweighted average is just the average of cache miss rate over the queries (thus treating them equally), while the weighted average is weighted by each query's instruction count, thus queries 1 and 18 con- tribute more than rest to that average. The fastest algorithm

for each query in boldface is indicated. Note that MaxResidency performs significantly better than all of the other algorithms in both cases.

Table 4: I-cache miss rate (as a percentage) across queries, with lowest highlighted in boldface)

query	*Instruction*	*Cache Placement Algorithms*				
		SlotZ	NextToC	BinaryFunctio	Function	MaxReside
1	247,660,74	2.70	3.71	0.424	0.518	0.452
3	19,353,663	0.94	0.607	0.491	0.270	0.213
4	23,510,390	1.48	0.773	0.313	0.279	0.0564
5	15,196,049	0.86	0.399	0.205	0.194	0.111
6	19,676,968	3.33	4.05	0.415	0.168	0.144
7	18,415,720	0.93	0.408	0.211	0.201	0.150
8	10,761,252	0.81	0.136	0.0946	0.103	0.0673
9	56,701,641	0.39	0.217	0.151	0.137	0.153
10	29,830,976	0.77	0.249	0.216	0.145	0.106
11	2,337,877,	0.74	0.326	0.242	0.263	0.218
12	20,837,423	1.42	0.306	0.0864	0.0546	0.0511
13	18,137,990	0.78	0.657	0.408	0.365	0.0746
14	13,760,553	1.99	0.467	0.122	0.0841	0.0847
16	32,161,032	0.18	0.163	0.0907	0.0737	0.0809
18	62,068,812	0.59	0.321	0.0885	0.197	0.163
19	16,833,167	1.40	0.283	0.150	0.141	0.120
22	5,538,705,	0.84	0.191	0.188	0.270	0.115
weighted	36,046,057	1.66	1.83	0.288	0.312	0.256
unweigh	*N/A*	1.19	0.780	0.229	0.204	0.139

[00185] Notice that NextToCaller is inferior to SlotZero in terms of the weighted average metric, which is unexpected. This anomalous result occurs because SlotZero only affects thecache pressure on one slot. NextToCaller, however, spreads the cache pressure across no more than 50 cache slots (recall that this experiment introduces 50 bee callers into the DBMS); thus slots could suffer from thrashing and attain even worse performance than SlotZero.

[00186] It is shown that out of the 17 queries in the benchmark, MaxResidency excels for 12 queries. For queries 9, 14, and 16, the performance difference between MaxResidency and that of the best algorithm (for that query) was minimal. Query 1 happens to have only two phases, so MaxResidency's ability to handle complex query trees is not fully utilized. (Note that even though this query incurs a lot of instructions, MaxResidency's miss rate for this query is close to that of BinaryFunctionScore.) Finally, for query 18, It is not sure why BinaryFunctionScore is so much better. The MaxResidency threshold setting adopted (0.15) might play a role. That said, for most queries MaxResidency significantly bettered BinaryFunctionScore.

[00187] The number of bees is also varied: 200 as a light work- load; 400 as a medium workload; and 500 and 1000 as heavy workloads. With all, MaxResidency produced the best results, for both weighted and unweighted. To indicate how well MaxResidency works in other cases, the weighted cache miss rate average is mentioned: 0.213% for 200 bees; 0.275% for 500 bees; and 0.498% for 1000 bees.

[00188] When the number of bees inserted for a query is varied, it can be seen how MaxResidency cache miss rate changes in Figure 14. Query 4 is marked with square while weighted average is marked with diamond (it is shown query 4 as it has the most significant change as the number of bees is varied). At 1000 bees, their cache miss rates between q4 and the average, demonstrating that for query 4 the cache miss rate goes up sharply after the I-cache slots have been filled up. The transition appears to be between 400 and 500. This makes sense, because the I-cache has 512 slots and DBMS stock code of course (partially) occupies many of them.

[00189] Several other interesting phenomena are observed:

- In an earlier experiment with only one bee caller, NextToCaller achieved the minimum I-cache miss rate for query 6. However, its execution time was almost twice as longer as the others. The reason for such significant slow-down is that NextToCaller's L2-cache miss rate is significantly higher than others algorithms. As L2-cache misses are much more expensive that I-cache misses, this effect dominates the run time. L2-cache misses in the table discussed above (with fifty bee callers) is not shown because the I-cache misses in Table 3 are spread across a large number of L2 cache slots, resulting in a uniformly low L2-cache miss rate, less than 0.012% in all cases.
- Each algorithm is implemented in a separate C file and one binary is installed per algorithm. The assumption is that all binaries will have similar cache behavior if no bee is invoked. This assumption was proven wrong when MaxResidency's performance is studied and it is found that half the cache misses came from code placed by gcc. The conflict was between the stock DBMS code, bee invocation code, and actual bee execution during query evaluation. This problem is

addressed by optimizing the length of bee invocation code so it has minimal conflicts with stock DBMS code.

- It is concerned that a placement algorithm's execution time could offset its benefit in terms of reduced cache miss rate. In the end, it turned out that even the most complicated placement algorithm takes very little time, 0.02 seconds in the worst case. Since it has constant number of input profiles (one per available operation), all of the same length (number of slots), the its complexity is linear in the number of bees, and for MaxResidency, also linear (additively) in the number of stages in query plan tree, both of which are small.

[00190] In summary, these optimizations result in MaxResidency performing quite well across the range of TPC-H queries, for a rather large number of bees.

[00191] Some of the algorithms introduced in *Section: Related Previous Work* are reviewed again. The algorithms include those that could possibly be applied in a dynamic setting, and evaluate specifically how the presented MaxResidency algorithm compares.

[00192] Huang et al. [X. Huang, S. M. Blackburn, D. Grove, and K. S. McKinley. Fast and efficient partial code reordering: Taking advantage of dynamic recompilation. In *Proc. International Symposium on Memory Management*, pages 184–192, New York, NY, 2006] proposed Code Reordering, focusing on direct-mapped caches, with an associativity of 1. Sherwood et al. [T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proc. International Conference on Supercomputing*, pages 155–164, New York, NY, 1999] proposed a Conflict Matrix based coloring, this is very similar to the Temporal Relational Graph [N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, Sept. 1999] idea. They also fail to consider the cache associativity. In contrast, the disclosed algorithms apply to caches with higher associativity.

[00193] Whaley et al. [J. Whaley. Partial method compilation using dynamic profile information. *ACM SIGPLAN Notices*, 36(11):166–179, Oct. 2001] proposed that dynamic analysis of control flow, by collecting profiles at run-time, which can be very expensive. The disclosed algorithms, for those that use profiles, require only that they be of a representative workload and are created at compile time.

[00194] Chen et al. [J. B. Chen and B. D. Leupen. Improving instruction locality with just-in-time code layout. In Proc. USENIX Windows NT Workshop, pages 25–32, 1997] developed a greedy placement strategy that relies on the invocation sequence instead of the invocation times used in Pettis & Hansen's. This approach is essentially grouping into cache slots from a finite-depth stack of activation frames. It is believed that their approach if extended to the dynamic situation would suffer from the same problem as LRU when there is a deep query plan.

[00195] Ding et al. [C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, New York, NY, 1999] proposed Locality Grouping. But this algorithm focuses on data cache accesses and thus does not help reduce I-cache pressure.

[00196] Ding et al. [C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, pages 245–257, 2003] propose Phase Prediction, which uses knowledge of reuse distance. Their algorithm applies only to simple dynamic patterns, whereas quite specific and complex patterns within query plans can be exploited while determining the placement.

[00197] Five placement algorithms are evaluated on I-cache miss rate and L2 cache miss rate, and one experiment on overall execution time. Concerning the I-cache miss rate, MaxResidency was the best. Concerning the L2 miss rate, all were low, below 0.012%. Overall query time for SlotZero and BinaryFunctionScore for one query, q6, are also compared from an actual full implementation. NextToCaller demonstrated the benefit of micro-specialization at a 15% improvement (this included some slowdown from cache misses); for SlotZero, the increased cache pressure entirely obviated the benefit, due to a 1% cache miss rate. The overall placement time was minimal, less than 0.02sec to place 400 bees, for all algorithms.

[00198] The evaluation has shown that the MaxResidency dynamic bee placement algorithm minimizes I-cache pressure for workloads with as many as 1000 bees. Thus, aggressive micro-specialization of a large number of functions invoked during query evaluation can improve query evaluation performance without concern for increased

I-cache pressure and thus also L2-cache pressure, through the use of this highly-effective bee placement algorithm.

[00199] The algorithms presented in this paper all attempt to minimize instruction cache misses. Because the L2 cache is generally an integrated cache (not differentiating between instructions and data), bee placement algorithms should consider the L2 cache pressure due to first-level *data* cache misses. This might be able to be accommodated via appropriate L2 and data profiles, as future work. That said, when considering up to even 1000 bees, the best algorithms are able to minimize the number of L2 misses. It would be interesting to determine the number of bees needed to produce significant L2-cache pressure.

[00200] Although the foregoing invention has been described in some detail for purposes of clarity of understanding, it will be apparent that certain changes and modifications can be practiced within the scope of the appended claims. Accordingly, the present embodiments are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope and equivalents of the appended claims.

CLAIMS

1. A computer-implemented method for field specialization broadening, the method comprising:

receiving inputs from source code and one or more workload;

identifying at least one invariant within the source code while executing the one or more workload;

dynamically creating at least one specializer in applications receiving the at least one identified invariant;

executing the at least one specializer for specialization to produce at least one specialized code; and

dynamically loading the at least one specialized code into at least one of the applications receiving the at least one identified invariant.

2. The computer-implemented method of claim 1 wherein the computer-implemented method further comprises specifying cross-application value flows by an ecosystem specification for how data travels from one application to another.

3. The computer-implemented method of claim 2 wherein the ecosystem specification specifies cross-application value flows by annotating a plurality of connecting paths with means to extend the value flows.

4. The computer-implemented method of claim 2 or claim 3, wherein the computer-implemented method further comprises extending an inter-application analysis to understand semantics of the cross-application value flows and to combine value flows within a single application to compute value flows across applications, and wherein the inter-application analysis preferably is implemented to determine where the at least one identified invariant can be best instantiated.

5. The computer-implemented method of claim 1 wherein the computer-implemented method further comprises verifying the possibility of the identified at least one invariant originating in an application and terminating in a specialization opportunity in a separate application, and/or

wherein the computer-implemented method further comprises identifying invariants that flow from external data or external applications, and/or

wherein the computer-implemented method further comprises deciding when to perform the specialization for a specializer located in an application separated from the applications containing the identified at least one invariant, and/or

wherein dynamically loading the at least one specialized code comprises storing the at least one specialized code separately from the application being executed with the at least one specializer.

6. A non-transitory computer readable medium comprising computer executable instructions that, when executed by a processor of a computing device, cause the computing device to:

receive an application code and one or more workload from an application;

identify at least one invariant within the application code and the one or more workload code;

dynamically create at least one specializer in applications receiving at least one identified invariant;

execute the at least one specializer for specialization to produce at least one specialized code; and

output the at least one specialized code.

7. The non-transitory computer readable medium of claim 6 wherein the field specialization broadening module is further configured to specify cross-application value flows by an ecosystem specification for how data travels from one application to another, and/or wherein the field specialization broadening module is further configured to extend an inter-application analysis to understand semantics of the cross-application value flows and to combine value flows within a single application to compute value flows across applications, wherein the inter-application analysis preferably is implemented to determine where the identified at least one invariant can be best instantiated; and/or

wherein the field specialization broadening module is further configured to verify the possibility of the identified at least one invariant originating in an application and terminating in a specialization opportunity in a separate application; and/or

wherein the field specialization broadening module is further configured to identify invariants that flow from external data or external applications; and/or

wherein the field specialization broadening module is further configured to decide when to perform the specialization for a specializer located in an application separated from the applications containing the identified at least one invariant; and/or

wherein at least one specialized code is stored separately from the application being executed with the at least one specialize; and/or

wherein at least one specialized code is dynamically loaded into at least one of the applications receiving the identified at least one invariant.

8. A method for field specialization broadening, the method comprising:

identifying values of at least one invariant within an originating application;

creating within the originating application a specialized code version of a portion of a target application based on the identified values;

passing the specialized code to a target application; and

invoking the specialized code in the target application.

9. The method for field specialization broadening of claim 8 wherein the target application receives the values of at least one invariant.

10. A computer-implemented method to minimize cache pressure, the method comprising:

receiving a machine code of a database management system (DBMS) and a bee code, in a slot selection phase, to decide a slot number in a cache to associate the bee code, the bee code being a specialized version of a DBMS function created and loaded into the cache at run-time;

passing the decided slot number, in a bee insertion phase, to find an open memory address mapping to the decided slot number; and

executing, in a query evaluation phase, the bee code using the open memory address.

11. The computer-implemented method of claim 10 wherein the slot selection phase, the bee insertion phase and the query evaluation phase all occur while the DBMS is running; and/or

wherein the cache comprises an instruction cache (I-cache) or a Level 2 cache.

12. A computer-implemented method to minimize cache pressure using slot pressure profile (SPP)-guided Algorithm, the method comprising:

generating, in a SPP generation phase, an intermediate SPP from a binary code of a database management system (DBMS);

receiving, in a SPP processing phase, the generated intermediate SPP obtained in the SPP Generation phase and outputting a reference SPP;

accepting, in a candidate ordering phase, the reference SPP and producing a candidate slot list related to a cache by sorting the values in the reference SPP;

deciding, in a slot selection phase, a slot number among the candidate slot list;

passing the decided slot number, in a bee insertion phase, to find an open memory address mapping to the decided slot number for a bee code, the bee code being a specialized version of a DBMS function created and loaded into the cache at run-time; and

executing, in a query evaluation phase, a query with the bee code using the found open memory address.

13. The computer-implemented method of claim 12 wherein the SPP generation phase further comprising the step of:

identifying functions that may be invoked by operators implemented by the DBMS, preferably wherein dynamic analysis is used at DBMS compile time to provide an estimate of the cache pressure of the identified functions; and/or wherein the SPP generation phase further comprising the steps of:

obtaining a call graph for the DBMS for executing representative workload;

performing a breadth-first search to identify all functions the query may call;

for each identified function, generating a per-function binary instruction cache (I-cache) function score SPP;

adding up the function score SPPs for an operator to yield a per-operator reference I-cache function count SPP; and/or

wherein the SPP generation phase further comprising the steps of:

generating at DBMS compile time one or more SPPs by assigning weights to a binary SPP; and

collecting a raw SPP over the DBMS executing a workload; and/or

wherein the SPP processing phase, the candidate ordering phase, the slot selection phase, the bee insertion phase and the query evaluation phase all occur with running time of the DBMS; and/or

wherein the SPP processing phase comprises a Binary Function Score Algorithm to infer a cache pressure; and/or

wherein the SPP processing phase comprises a Function Count Algorithm to construct a reference SPP; and/or

wherein the SPP processing phase comprises a MaxResidency Algorithm to give a query plan, wherein the MaxResidency algorithm preferably comprises steps of:

computing an instruction threshold to identify slots that are accessed enough to potentially incur cache pressure;

checking a max residency of the identified slots; and

in response to the max residency reaches the instruction threshold of total counts at the identified slot, incrementing an estimated occupancy at this slot by 1, wherein the instruction threshold preferably is a ratio of a residency to a total instruction count at a slot that is considered long enough to occupy a cache line.

14. A computer-implemented method to evaluate cache pressure, the method comprising:

determining functions that will invoke a bee as bee callers, the bee being a specialized version of a database management system (DBMS) function created and loaded into a cache at run-time of the DBMS;

for a query plan, computing the number of bees needed to be called by each bee caller;

placing in each node of the query plan the called bees for each bee caller associated with an operator; and

invoking selected placement algorithms to place the called bees.

15. The computer-implemented method of claim 14 wherein the cache comprises an instruction cache (I-cache) or a Level 2 cache; and/or

wherein the query plan is manually created on a training workload; and/or

wherein determining bee callers is done at DBMS compile time.

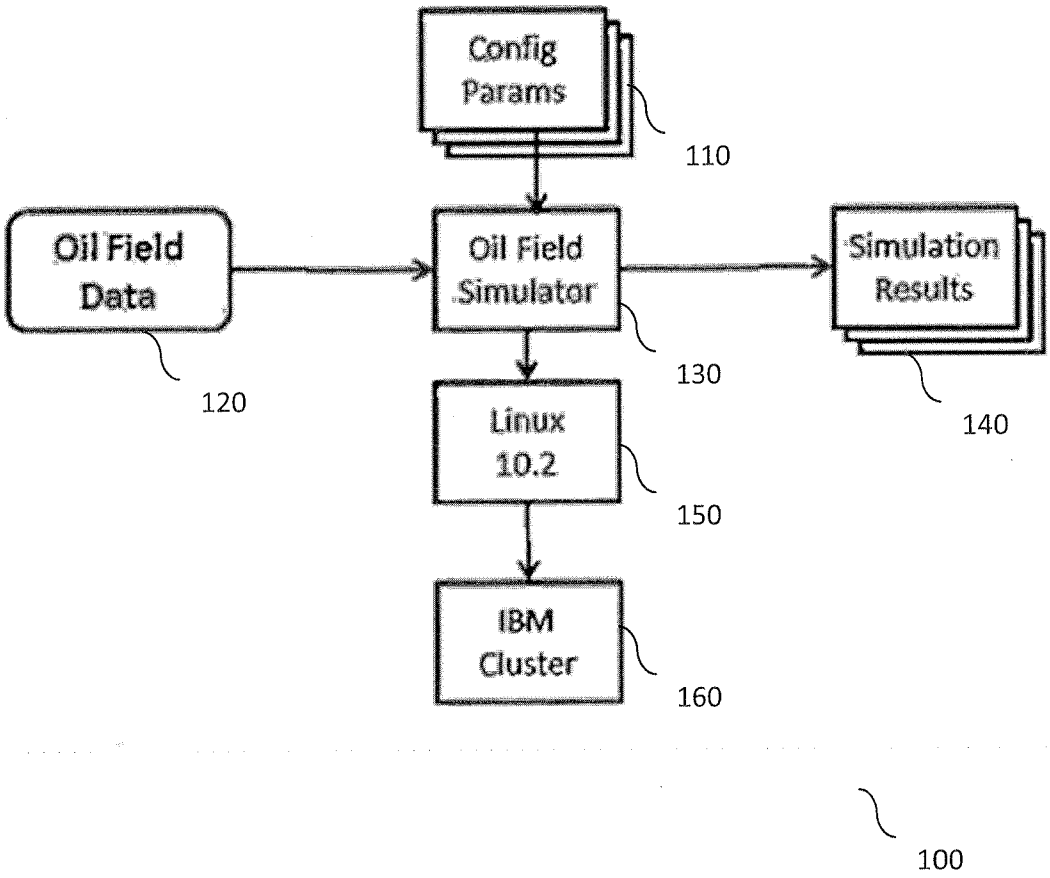

Config Params
110
Oil Field Data
120
Oil Field Simulator
130
Simulation Results
140
Linux 10.2
150
IBM Cluster
160
100

FIG. 1

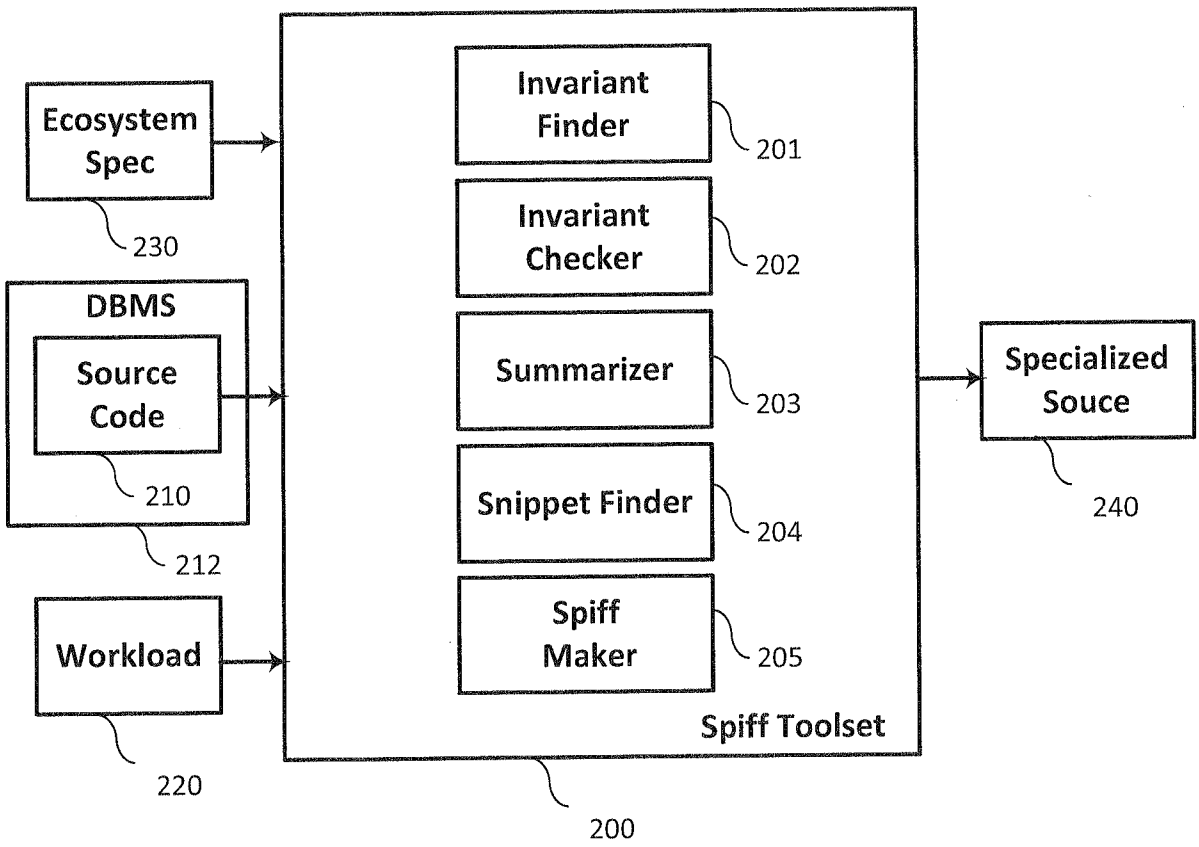

Ecosystem Spec
230
DBMS
Source Code
210
212
Workload
220
Invariant Finder
201
Invariant Checker
202
Summarizer
203
Snippet Finder
204
Spiff Maker
205
Spiff Toolset
200
Specialized Souce
240

FIG. 2

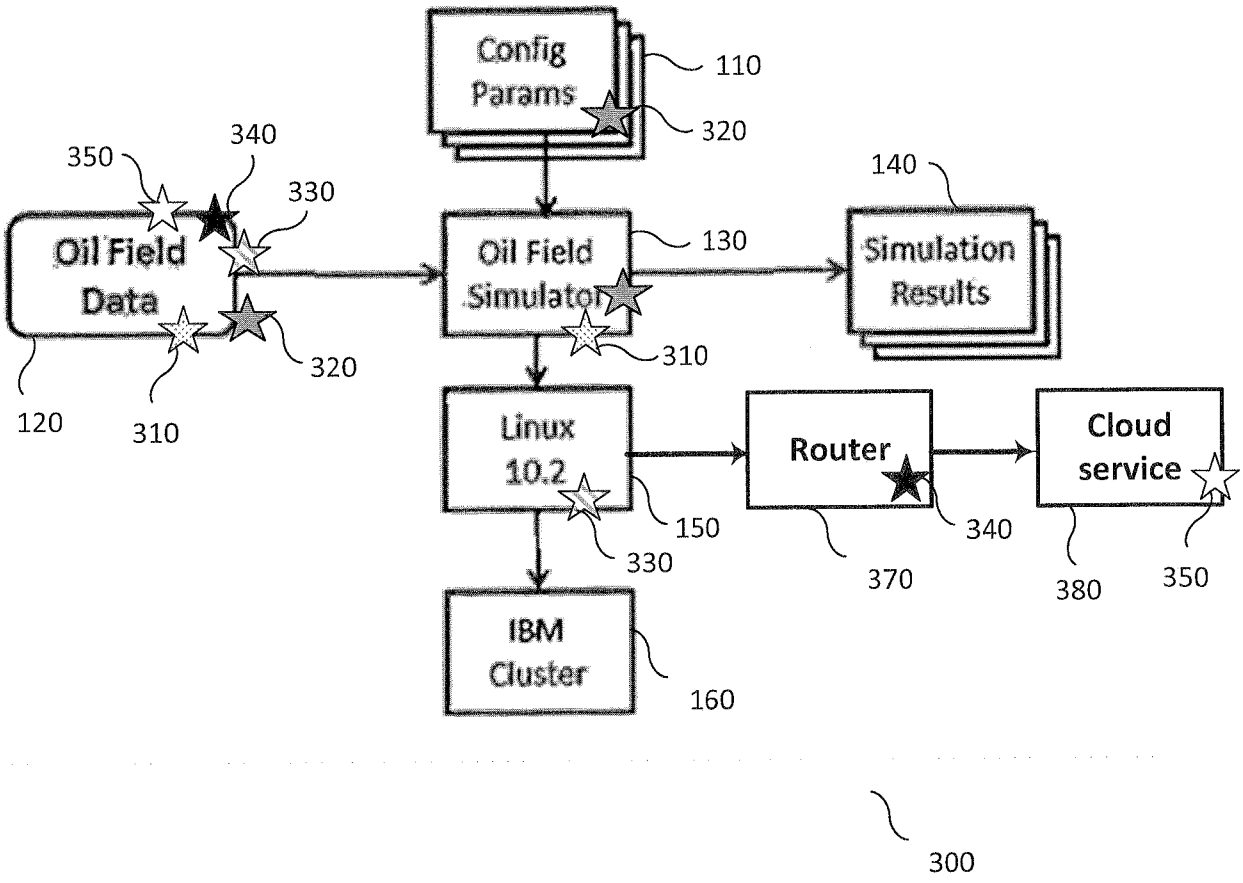

Config Params
110
320
340
350
330
140
Oil Field Data
Oil Field Simulator
130
Simulation Results
320
310
120
310
Linux 10.2
Router
Cloud service
150
340
330
370
380
350
IBM Cluster
160
300

FIG. 3

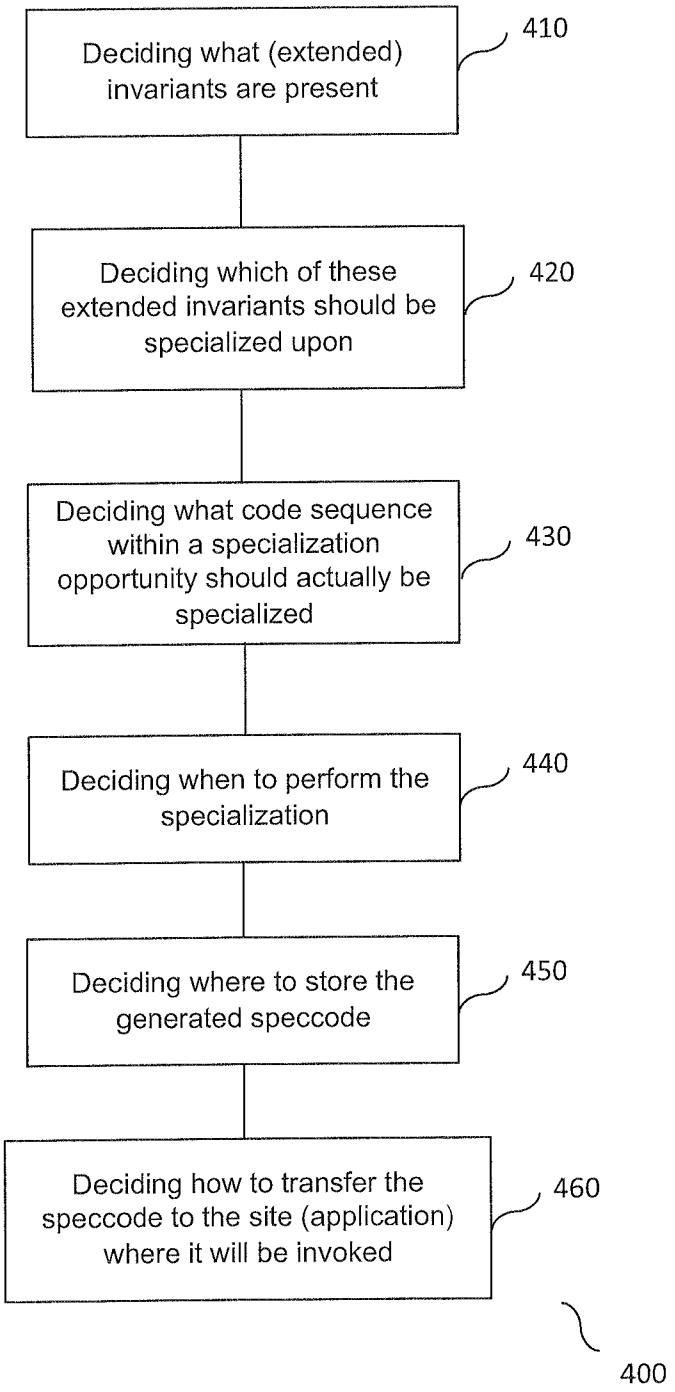

Deciding what (extended) invariants are present
410
Deciding which of these extended invariants should be specialized upon
420
Deciding what code sequence within a specialization opportunity should actually be specialized
430
Deciding when to perform the specialization
440
Deciding where to store the generated speccode
450
Deciding how to transfer the speccode to the site (application) where it will be invoked
460
400

FIG. 4

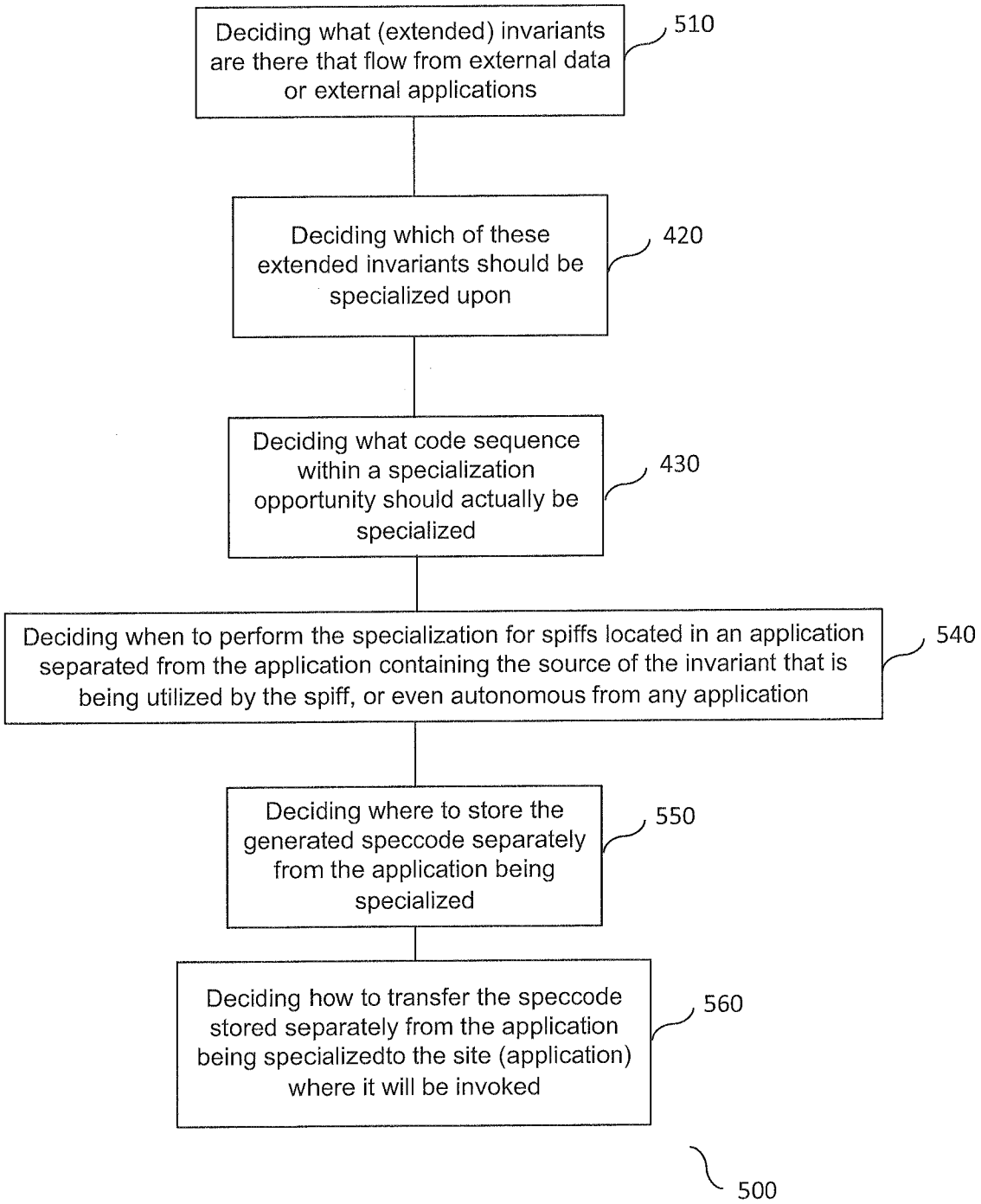

Deciding what (extended) invariants are there that flow from external data or external applications
510
Deciding which of these extended invariants should be specialized upon
420
Deciding what code sequence within a specialization opportunity should actually be specialized
430
Deciding when to perform the specialization for spiffs located in an application separated from the application containing the source of the invariant that is being utilized by the spiff, or even autonomous from any application
540
Deciding where to store the generated speccode separately from the application being specialized
550
Deciding how to transfer the speccode stored separately from the application being specializedto the site (application) where it will be invoked
560
500

FIG. 5

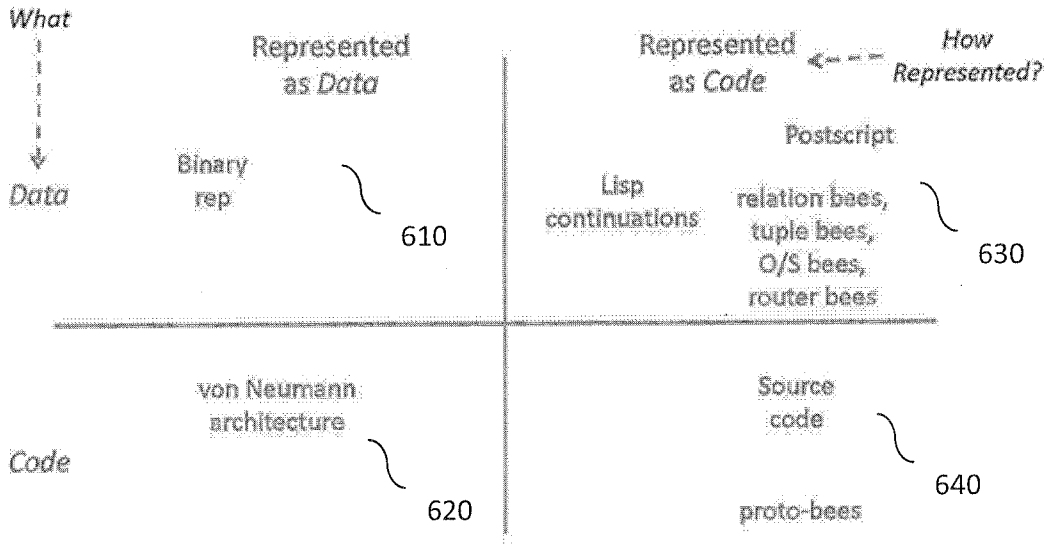

What
Data
Represented as Data
Represented as Code
How Represented?
Binary rep
610
Postscript
Lisp continuations
relation bees, tuple bees, O/S bees, router bees
630
von Neumann architecture
620
Code
Source code
640
proto-bees

FIG. 6

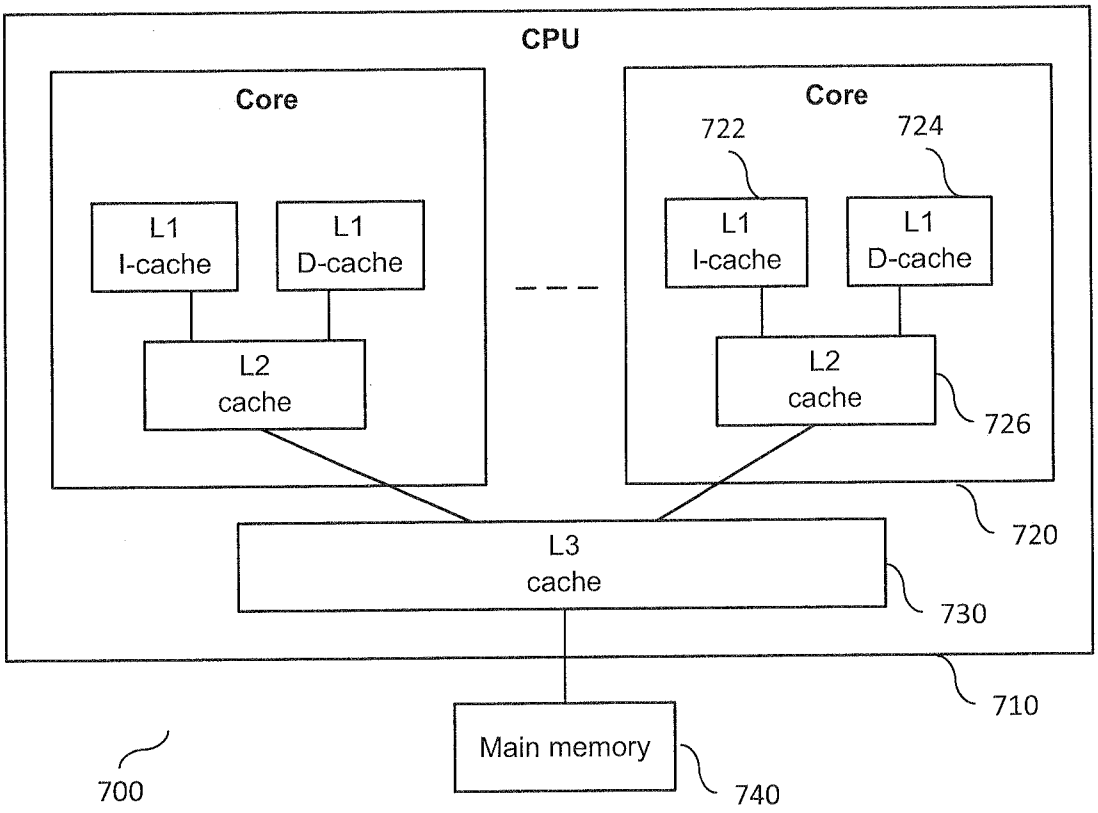

CPU
Core
L1
I-cache
L1
D-cache
L2
cache
Core
722
724
L1
I-cache
L1
D-cache
L2
cache
726
720
L3
cache
730
710
Main memory
740
700

FIG. 7

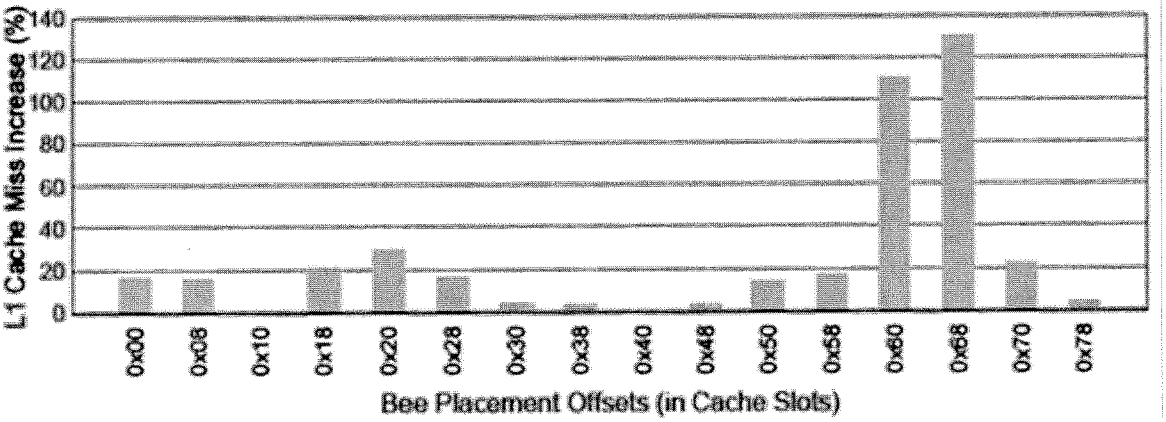

L1 Cache Miss Increase (%)
140
120
100
80
60
40
20
0
0x00
0x08
0x10
0x18
0x20
0x28
0x30
0x38
0x40
0x48
0x50
0x58
0x60
0x68
0x70
0x78
Bee Placement Offsets (in Cache Slots)

FIG. 8

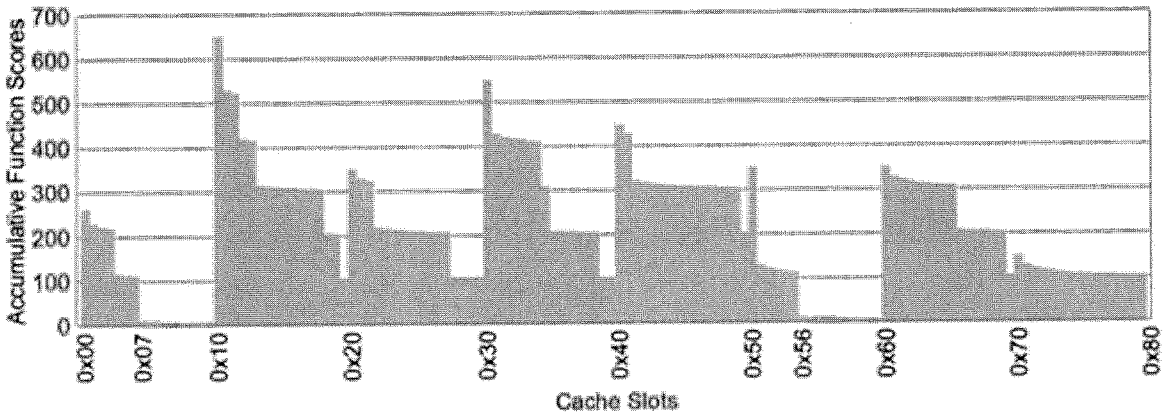

Accumulative Function Scores
700
600
500
400
300
200
100
0
0x00
0x07
0x10
0x20
0x30
0x40
0x50
0x56
0x60
0x70
0x80
Cache Slots

FIG. 9

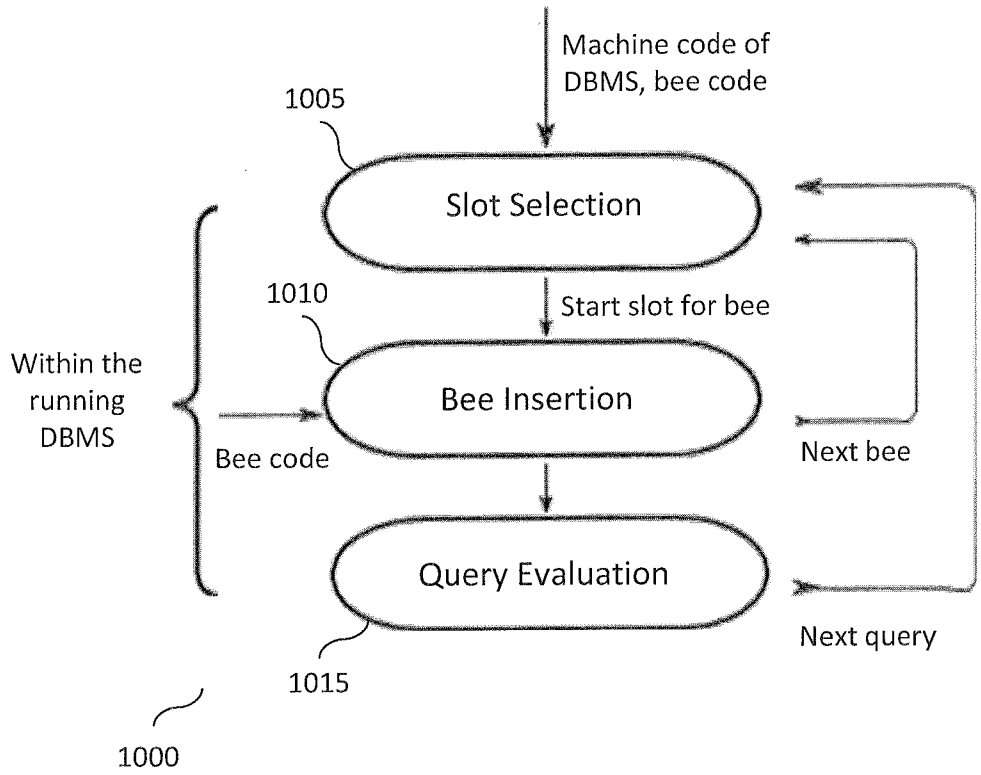

Machine code of
DBMS, bee code
1005
Slot Selection
1010
Start slot for bee
Within the
running
DBMS
Bee Insertion
Bee code
Next bee
Query Evaluation
Next query
1015
1000

FIG. 10

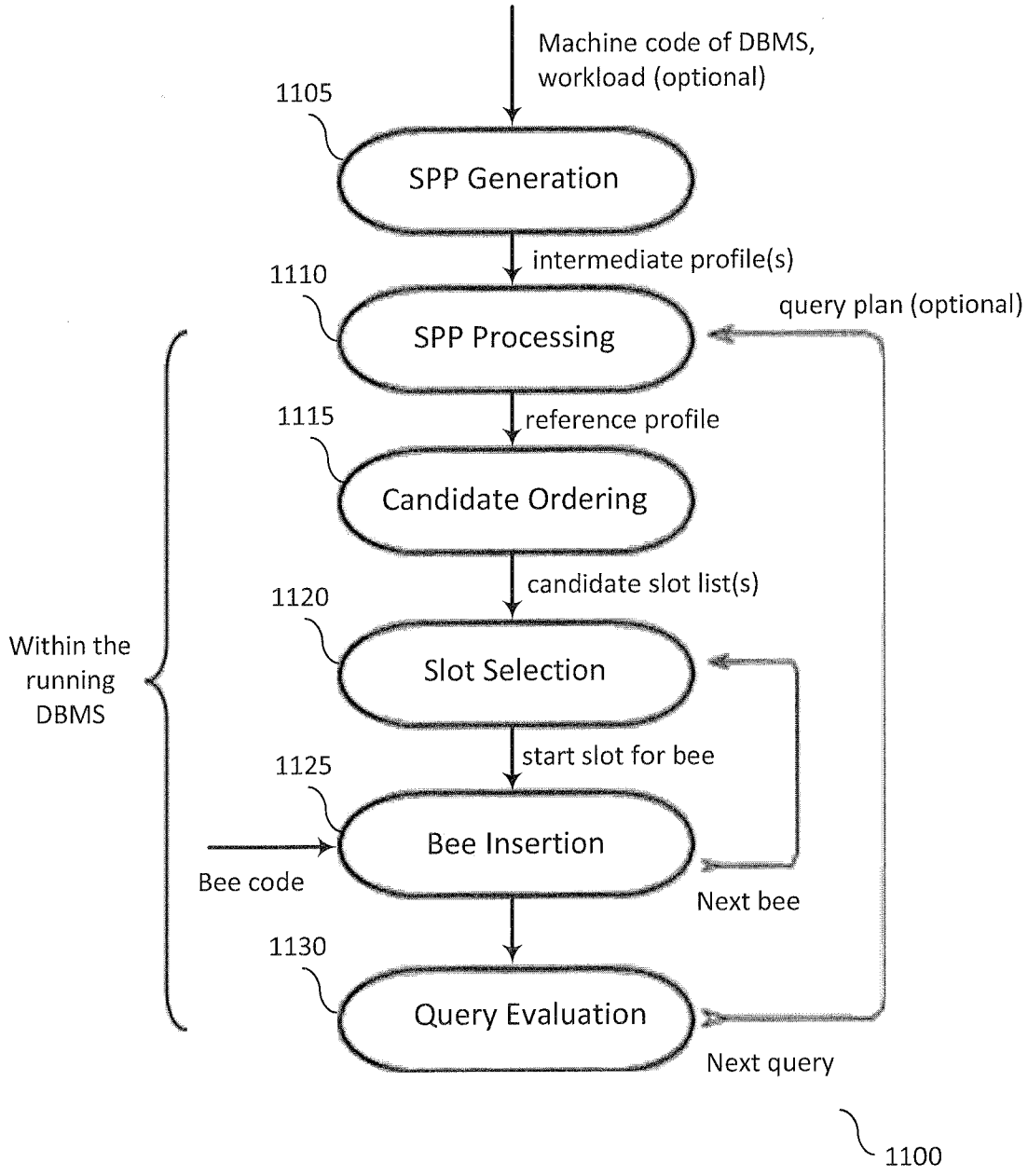

Machine code of DBMS, workload (optional)
1105
SPP Generation
intermediate profile(s)
1110
SPP Processing
query plan (optional)
1115
reference profile
Candidate Ordering
candidate slot list(s)
1120
Slot Selection
start slot for bee
1125
Bee Insertion
Bee code
Next bee
1130
Query Evaluation
Next query
Within the running DBMS
1100

FIG. 11

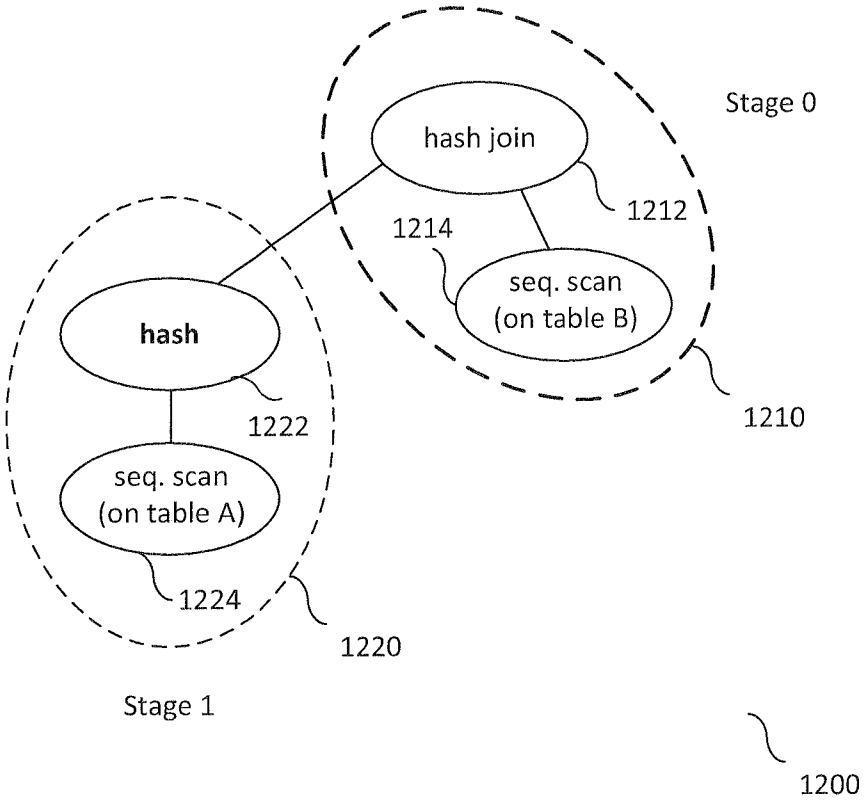

Stage 0
hash join
1212
1214
seq. scan
(on table B)
1210
hash
1222
seq. scan
(on table A)
1224
1220
Stage 1
1200

FIG. 12

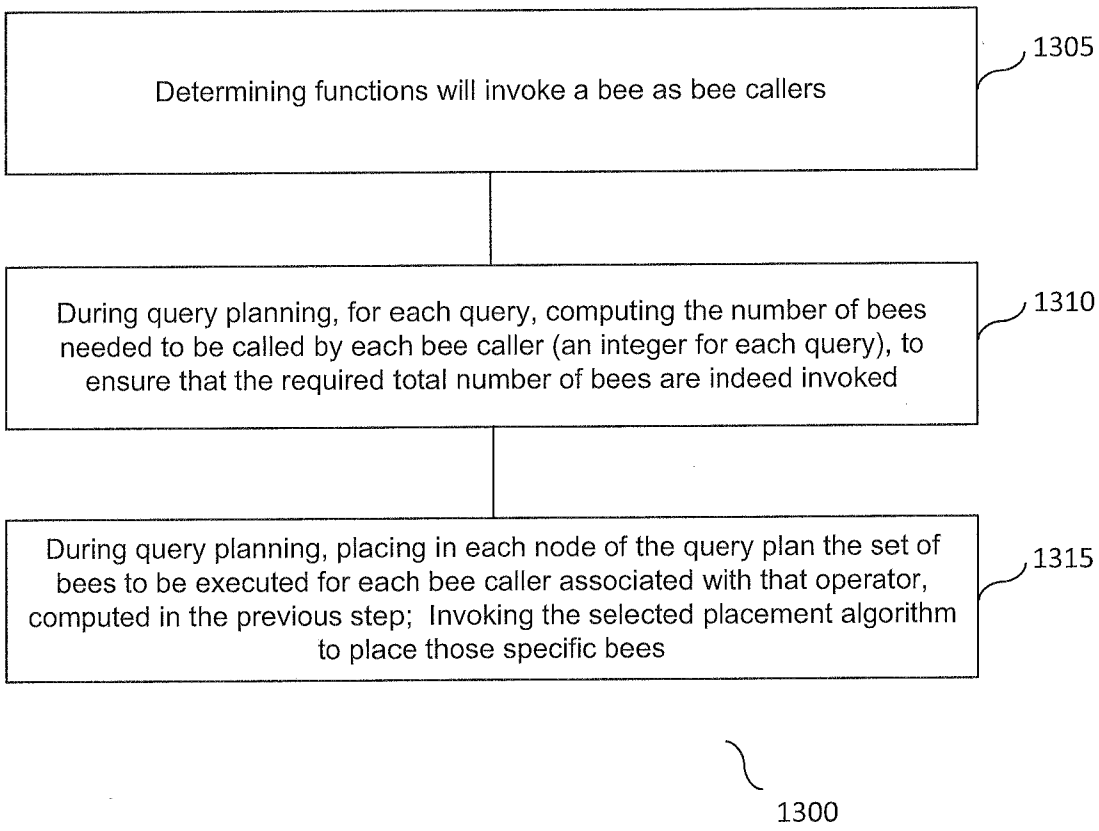

Determining functions will invoke a bee as bee callers
1305
During query planning, for each query, computing the number of bees needed to be called by each bee caller (an integer for each query), to ensure that the required total number of bees are indeed invoked
1310
During query planning, placing in each node of the query plan the set of bees to be executed for each bee caller associated with that operator, computed in the previous step; Invoking the selected placement algorithm to place those specific bees
1315
1300

FIG. 13

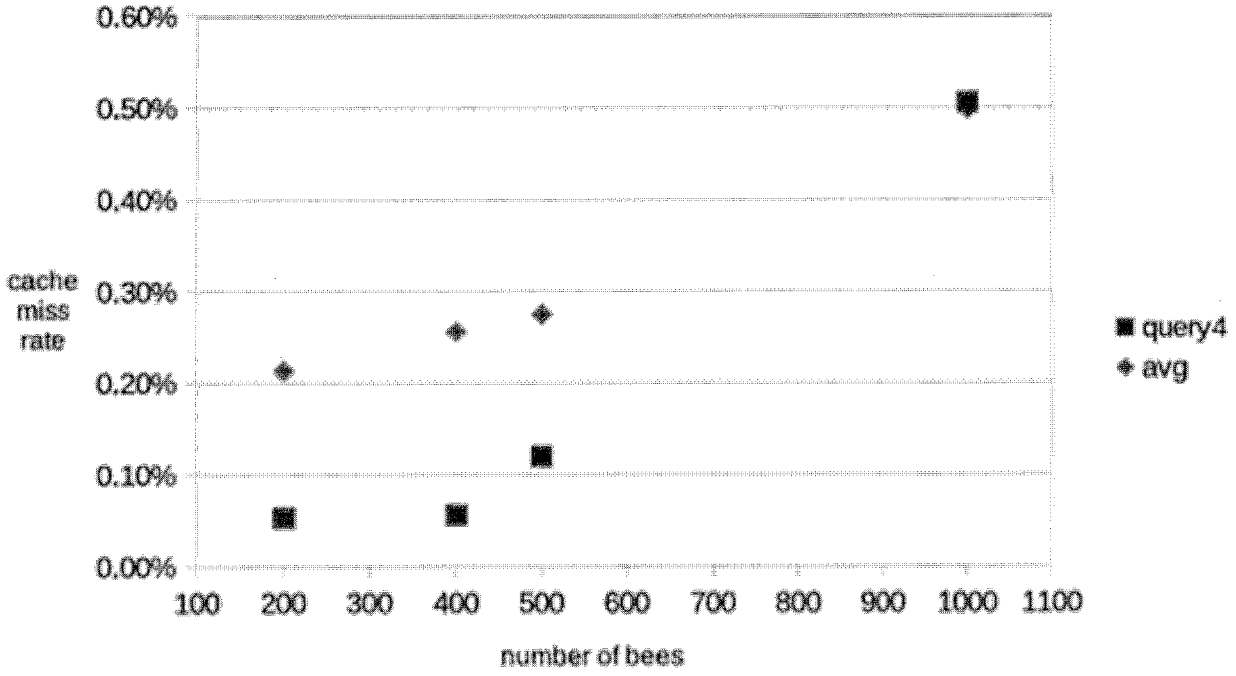

0.60%
0.50%
0.40%
0.30%
0.20%
0.10%
0.00%
cache miss rate
100
200
300
400
500
600
700
800
900
1000
1100
number of bees
query4
avg

FIG. 14

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/066665

A. CLASSIFICATION OF SUBJECT MATTER
IPC(8) - G06F 7/00; G06F 9/445; G06F 9/45; G06F 12/00; G06F 17/30 (2016.01)
CPC - G06F 8/36; G06F 8/433; G06F 9/44521; G06F 17/30289; G06F 17/30474 (2016.08)

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)
See Search History document

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
USPC - 707/803; 717/153; 717/166 (keyword delimited)

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
See Search History document

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X --- Y --- A	US 2014/0365533 A1 (THE ARIZONA BOARD OF REGENTS ON BEHALF OF THE UNIVERSITY OF ARIZONA) 11 December 2014 (11.12.2014) entire document	8-11, 14, 15 --- 1, 5-7 --- 2-4, 12, 13
Y --- A	US 2013/0054649 A1 (POTAPOV et al) 28 February 2013 (28.02.2013) entire document	1, 5-7 --- 2-4
A	US 2004/0117359 A1 (SNODGRASS et al) 17 June 2004 (17.06.2004) entire document	12, 13
A	US 2009/0248725 A1 (BHATTACHARJEE et al) 01 October 2009 (01.10.2009) entire document	12, 13
A	US 5,999,924 A (BAIR et al) 07 December 1999 (07.12.1999) entire document	1-15
A	US 2003/0200537 A1 (BARSNESS et al) 23 October 2003 (23.10.2003) entire document	1-15
A	US 2012/0317647 A1 (BRUMLEY et al) 13 December 2012 (13.12.2012) entire document	1-15
A	US 2008/0189277 A1 (MEIJER et al) 07 August 2008 (07.08.2008) entire document	1-15

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
05 April 2017	27 APR 2017
Name and mailing address of the ISA/US Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, VA 22313-1450 Facsimile No. 571-273-8300	Authorized officer Blaine R. Copenheaver PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

Form PCT/ISA/210 (second sheet) (January 2015)

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/066665

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:
See Extra Sheet(s)

1. ☒ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

☒ No protest accompanied the payment of additional search fees.

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/066665

Continued from Box No. III Observations where unity of invention is lacking

This application contains the following inventions or groups of inventions which are not so linked as to form a single general inventive concept under PCT Rule 13.1. In order for all inventions to be examined, the appropriate additional examination fees must be paid.

Group I, claims 1-9, drawn to field specialization broadening.
Group II, claims 10-15, drawn to minimize cache pressure.

The inventions listed as Groups I-II do not relate to a single general inventive concept under PCT Rule 13.1 because, under PCT Rule 13.2, they lack the same or corresponding special technical features for the following reasons: the special technical feature of the Group I invention: creating at least one specializer in applications receiving the at least one identified invariant; executing the at least one specializer for specialization to produce at least one specialized code as claimed therein is not present in the invention of Group II. The special technical feature of the Group II invention: receiving a machine code of a database management system (DBMS) and a bee code, in a slot selection phase, to decide a slot number in a cache to associate the bee code as claimed therein is not present in the invention of Group I.

Groups I and II lack unity of invention because even though the inventions of these groups require the technical feature of processing source code, this technical feature is not a special technical feature as it does not make a contribution over the prior art.

Specifically, US 2012/0317647 A1 (BRUMLEY et al) 13 December 2012 (13.12.2012) teaches processing source code (Paras. 19 and 25).

Since none of the special technical features of the Group I or II inventions are found in more than one of the inventions, unity of invention is lacking.