(51) **International Patent Classification:**
G06F 13/42 (2006.01)   G06F 13/14 (2006.01)
G06F 13/38 (2006.01)

(21) **International Application Number:**
PCT/US2009/066889

(22) **International Filing Date:**
4 December 2009 (04.12.2009)

(25) **Filing Language:** English

(26) **Publication Language:** English

(30) **Priority Data:**
12/353,905   14 January 2009 (14.01.2009)   US

(71) **Applicant** *(for all designated States except US)*: **MI-CROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, WA 98052-6399 (US).

(72) **Inventors: DADIOMOV, Alexander**; One Microsoft Way, Redmond, WA 98052-6399 (US). **GROFF, Dana**; One Microsoft Way, Redmond, WA 98052-6399 (US). **LEVANONI, Yosseff**; One Microsoft Way, Redmond, WA 98052-6399 (US). **JOHNSON, James, E.**; One Microsoft Way, Redmond, WA 98052-6399 (US).

(81) **Designated States** *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ,

CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, 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, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (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, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

(54) **Title:** TRANSACTION PROCESSING IN TRANSACTIONAL MEMORY
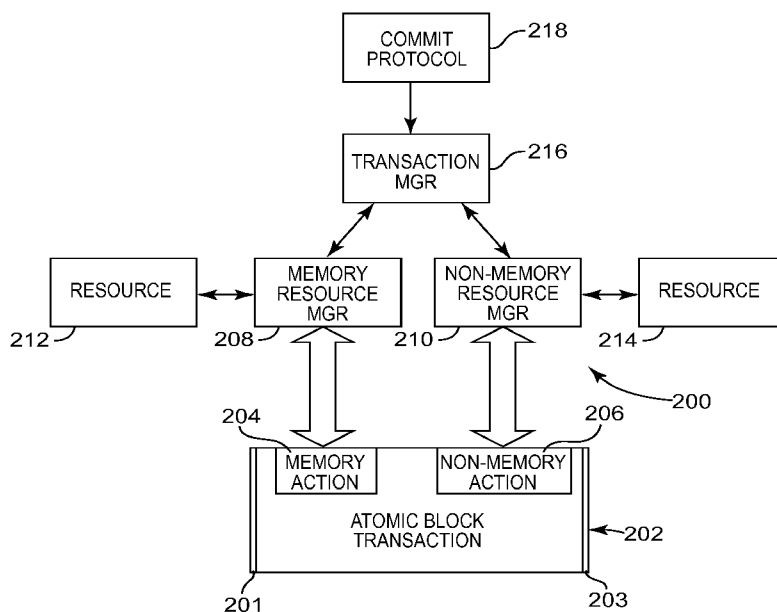

Fig. 2

(57) **Abstract**: A transactional memory processing system provides for the integration of transactional memory concepts at the compiler-level into a higher-level traditional transaction processing system. Atomic blocks at the compiler-level can be specified as atomic block transactions and include the features of atomicity and isolation. Actions within this atomic block transaction include the enlistment of resource managers from a repository. The repository can now include a pre-programmed memory resource manager to manage the transactional memory. As in traditional transactions, a commit protocol can be used to determine if the actions are valid and can be exposed outside of the transaction. Unlike traditional transactions, however, the transaction is not necessarily doomed if all of the actions are not validated. Rather, memory conflicts can cause a rollback and re-execution of the atomic block transaction, which can be repeated as long as necessary, until the memory resource manager votes to commit.
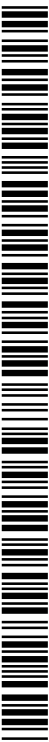
WO 2010/082983 A2 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

TRANSACTION PROCESSING IN TRANSACTIONAL MEMORY

Background

[001]   Concurrent programming for shared-memory multiple processor systems can include the ability for multiple threads to access the same data. The multiple threads execute on multiple processors, multiple processor cores, or other classes of parallelism that are attached to a memory shared between the processors. The shared-memory model is the most commonly deployed method of multithread communication. It allows multithreaded programs to be created in much the same way as sequential programs. In order to implement the shared-memory model, concurrent programming uses care to avoid concurrent access and use of shared data that can create undesirable conditions such as races and the like.

[002]   Locks are a common solution to avoid the problem of concurrent access to shared data. Locks are centered on a premise that other threads may also try to access variables accessed by a certain thread, while the variable can only be used by one thread at a time. Locks allow one thread to take control of a variable and prevent other threads from changing the variable until it is unlocked. Lock-based protocols, while popular, are often considered difficult to use. Using locks in a coarse-grained way protects relatively large amounts of data, but generally their use does not scale. Threads block one another even when they do not interfere, and the locks become a source of contention. Alternatively, using locks in a more fine-grained way can mitigate scalability issues, but the locks introduce other problems because the locking conventions to ensure correctness and avoid deadlocks become complex and error prone.

[003]   Another solution is to implement applications using transactional memory. Transactional memory systems manage the memory accesses of threads by executing the threads in such a way that the effects of a thread can be rolled back or undone if two or more threads attempt to access the same memory location in a conflicting manner. Transactional memory systems can be implemented using hardware and/or software components. A software transactional memory system can provide semantics in a software runtime library and/or runtime execution environment and/or using compilers. Transactional memory is frequently

implemented as a compiler-level concurrency control mechanism for controlling access to shared memory based on the premise that variables read by one thread will likely not be modified by other threads, and thus the variable can be shared without harsh ramifications to the scalability of the program. Tracking memory access in transactional memory systems, however, can possibly add overhead to the execution of programs.

[004]   One benefit of transactional memory over coarse-lock-based protocols is increased concurrency. In transactional memory, no thread needs to wait for access to data, and different threads can safely and simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock. Despite the overhead of retrying transactions that fail, in many realistic concurrent programs conflicts arise rarely enough that there can be a performance gain over course-grained lock-based protocols starting from certain number of processors or processor cores.

[005]   Despite the promise of transactional memory and being the subject of extensive research, obstacles remain to its widespread use and acceptance. For example, programmers can be reluctant to use transactional memory, because of unfamiliarity and lack of a practical, user-friendly implementation of transactional memory.

Summary

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

[007]   The disclosure is directed to an integration of transactional memory concepts at the compiler-level into a higher-level traditional transaction (such as database transactions) processing. Atomic blocks at the compiler-level can be specified as atomic block transactions and include the features of atomicity and isolation. Actions within this atomic block transaction include the enlistment of resource managers from a repository of available resource managers. The

repository can now include a pre-programmed memory resource manager to manage the transactional memory. As in traditional transactions, a commit protocol can be used to determine if the actions were successful and the results can be exposed outside of the transaction. Unlike traditional transactions, however, the transaction is not necessarily doomed if some of the actions are not validated. Rather, memory conflicts cause a rollback of all resource managers, including memory, and re-execution of the atomic block transaction. Re-executions can be repeated as long as necessary until all operations, including the memory resource manager operations, are validated successfully.

Brief Description of the Drawings

[008]   The accompanying drawings are included to provide a further understanding of embodiments and are incorporated in and constitute a part of this specification. The drawings illustrate embodiments and together with the description serve to explain principles of embodiments. Other embodiments and many of the intended advantages of embodiments will be readily appreciated, as they become better understood by reference to the following detailed description. The elements of the drawings are not necessarily to scale relative to each other. Like reference numerals designate corresponding similar parts.

[009]   Figure 1 is a block diagram illustrating one of many possible examples of computing devices implementing the features of the present disclosure.

[0010]   Figure 2 is a block diagram illustrating an example transactional memory processing system implemented in the example computing system of Figure 1.

[0011]   Figure 3 is a flow diagram illustrating an example process used in the transactional memory processing system of Figure 2

[0012]   Figure 4 is a schematic diagram illustrating an example process of a commit protocol of Figure 2 in relation to a timeline.

Detailed Description

[0013]   In the following Detailed Description, reference is made to the accompanying drawings, which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural or logical

changes may be made without departing from the scope of the present invention. The following detailed description, therefore, is not to be taken in a limiting sense, and the scope of the present invention is defined by the appended claims. It is also to be understood that features of the various exemplary embodiments described

5   herein may be combined with each other, unless specifically noted otherwise.

[0014]   Figure 1 illustrates an exemplary computer system that can be employed as an operating environment and includes a computing device, such as computing device 100. In a basic configuration, computing device 100 typically includes a processor architecture having at least two processing units, (i.e., processors 102),

10 and memory 104. Depending on the exact configuration and type of computing device, memory 104 may be volatile (such as random access memory (RAM)), non-volatile (such as read only memory (ROM), flash memory, etc.), or some combination of the two. This basic configuration is illustrated in Figure 1 by line 106. The computing device can take one or more of several forms. Such forms

15 include a personal computer, a server, a handheld device, a consumer electronic device (such as a video game console), or other.

[0015]   Computing device 100 can also have additional features/functionality. For example, computing device 100 may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or

20 solid state memory, or flash storage devices such as removable storage 108 and non-removable storage 110. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any suitable method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Memory 104,

25 removable storage 108 and non-removable storage 110 are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile discs (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, universal serial bus

30 (USB) flash drive, flash memory card, or other flash storage devices, or any other medium that can be used to store the desired information and that can be accessed

by computing device 100. Any such computer storage media may be part of computing device 100.

[0016]   Computing device 100 includes one or more communication connections 114 that allow computing device 100 to communicate with other computers/applications/users 115. Computing device 100 may also include input device(s) 112, such as keyboard, pointing device (e.g., mouse), pen, voice input device, touch input device, etc. Computing device 100 may also include output device(s) 111, such as a display, speakers, printer, etc.

[0017]   Computing system 100 can be configured to run an operating system software program and one or more software applications, which make up a system platform. In one example, computing system 100 includes a software component referred to as a runtime environment. The runtime environment can be included as part of the operating system or can be included later as a software download. The runtime environment typically includes pre-coded solutions to common programming problems to aid software developers to create software programs such as applications to run in the runtime environment, and it also typically includes a virtual machine that allows the software applications to run in the runtime environment so that the programmers need not consider the capabilities of the specific processors 102. Examples of runtime environments include .Net Common Language Routine from Microsoft, Inc., of Redmond, Washington, and also a C++ routine, among many others.

[0018]   Figure 2 illustrates an example transactional memory processing system 200 that can be called in the runtime environment to support an atomic bock transaction 202, which can be an atomic block of software code in a thread. The atomic block transaction 202 includes transaction boundaries 201, 203 that delineate the atomic block and specify the code to participate in the transactional memory processing system 200. The atomic block transaction 202 includes at least a transactional memory action 204 but is illustrated to also include and a second, or non-memory action 206. The transactional memory action 204 provides for a memory operation with the transactional memory. Other actions can be included in the atomic block transaction such as an action that affects a variable, an action that

affects a side-effect, and so on. The system 200 includes at least one memory resource manager to correspond with at least one memory action, such as a memory resource manager 208 to correspond with the transactional memory action 204. The system 200 can include at least one non-memory resource manager to

5   correspond with at least one non-memory action, such as a non-memory resource manager 210 enlisted into the transaction 202 to correspond with the non-memory second action 206 in the example. The resource managers 208, 210 each manage action-appropriate resources 212, 214 that participate in the transaction 202. The actions of the resource managers 208, 210 are coordinated with a transaction

10   manager 216, which works with the resource managers 208, 210 to ensure atomicity and isolation of the transaction 202. The transaction manager 216 implements a commit protocol 218. The resource managers 208, 210 also participate in the commit protocol 218. In another example, an atomic block transaction can include more than two actions that correspond with a plurality of

15   resource managers and resources.

[0019]   A transaction 202 is a sequence of bound-together actions executed by a single thread. A thread completes modifications to data in the shared memory either with or without regard for other threads concurrently running on the other processors. At any rate, after completing a transaction, the transactional memory

20   verifies that other threads have not concurrently made changes to the accessed data. The changes are validated and, if the validation is successful, made permanent in a commit operation. If the validation fails, the changes are undone or "rolled back," and the transaction 202 is re-executed until the validation is successful.

[0020]   Transaction 202 possesses the features of atomicity and isolation.

25   Transactions are atomic and are logically performed instantaneously. If one action fails then the entire transaction fails. Also, the transaction is isolated from other threads in that no variables are exposed to other threads in their intermediate states. In some embodiments, isolation is only provided to threads which are executing inside transaction, while threads that are accessing the data without the protection

30   of transactions may witness intermediate states of concurrent transactions. When the end of a block is reached, the transaction is committed, aborted or rolled-back

and re-executed. Accordingly, the unit that commits or fails is the transaction 202 rather than the entire process, and the state is returned to its original form rather than exposing intermediate state. In this respect, the transaction 202 is similar to a transaction of the database arts, i.e., a database transaction, except that it is typically expressed at the compiler level and involves a transactional memory. Unlike a typical traditional transaction (e.g. a database transaction), however, transaction 202 can be re-executed until it is successful.

[0021] Resource managers exist in traditional transaction processing and manage resources participating in transactions. But traditional transaction processing so far has not been used to automatically isolate and synchronize concurrent access to memory, which has either used locks or transactional memory mechanisms, unconnected to traditional transactions. A feature of the present example is that management of the transactional memory is part of the traditional transactional system, combining transactional memory with traditional failure atomicity - unlike traditional transactions that do not control access to shared memory. Transactional memory proposals do not provide general failure atomicity across memory and other resources. Error processing complexity in transaction memory proposals often prevents an application from recording from partial failures or a combination of memory conflicts. Developers still provide recovery functionality by manually crafting solutions for a limited set of error cases that they know how to deal with, and as a result suffer the productivity and quality consequences. In addition, crafting manual error processing solutions has proven to be a non-trivial task for developers. The examples of the present disclosure, however, incorporate transactional memory into traditional transaction processing to provide for failure atomicity. Thus, either all the supported actions are executed or nothing will appear to have been executed. The runtime environment can use transactional memory mechanisms as a preprogrammed resource manager for memory. The preprogrammed resource managers for other appropriate actions can be included in a library or repository that can be called in the runtime environment when such resource is to be used.

[0022]   In many cases where an appropriate resource manager does not exist in the preprogrammed library of resource managers, a program developer can write a resource manager for use in the program or add it to the library. This can be done in the standard way used in transitional transaction processing, which is more reliable than manually crafting an error processing solution without a transaction processing system 200.

[0023]   Further, the resource managers used in traditional transactions possess the features of atomicity, consistency, isolation, and durability (ACID), whereas transaction processing in transactional memory does not require the features of consistency and durability. As a result, examples of resource managers 208, 210 can be implemented as volatile resource managers rather than durable ones. Volatile resource managers store their state in a volatile memory and do not support transactional state recovery. Consequently, the development of volatile resource managers is significantly easier than the development of durable resource managers. In the case of the transaction processing system 200, volatile resource managers use fewer system resources than durable resource managers. An example transaction manager 216 that works well with multiple volatile resource managers is an available lightweight transaction manager, which can significantly reduce overhead resulting from more durable transaction managers. Other examples can include durable transaction managers or resource managers.

[0024]   The resource managers 208, 210 automatically enlist in the transaction 202 and commit or roll back changes made to their state according to the outcome of the transaction. The runtime environment can automate the enlistment in the transaction and the management of the transaction with respect to the transactional resources 212, 214. In enlisting a resource 212, 214 in a transaction 202, the resource informs the transaction manager that an action 204, 206 wants to perform transactional work against the resource. The action 204, 206 then performs work against the resource 212, 214, and if no functional error or conflict occurs, the transaction manager 216 applies the commit protocol 218 to ask the resource 212, 214 through the resource manager 208, 210 to commit the changes made to its state. If either of the resource managers 208, 210 encounters an error, the

transaction manager 216 will cause a roll back of all changes made inside the transaction by actions 204, 206. Otherwise, the transaction manager 216 will cause the transaction to commit. In either case, the transaction manager 216 can inform the resource managers 208, 210 of the decision.

[0025]   In one example, the memory resource manager 208 initially acts as a transaction manager before another resource manager enlists in the transaction 202. If no other resource action happens, resource manager 208 can execute at its own, without higher level transaction manager 216. Once another resource manager enlists, if at all, such as the non-memory resource manager 210, the transaction processing system 200 promotes the transaction 202 from "memory-only transaction" to one employing the transaction manager 216, and the memory resource 208 becomes a resource manager.

[0026]   Figure 3 illustrates an example process at 300 for use with transaction processing combined with transactional memory. At 302, a transaction 202 is run in an example transactional memory processing system 200. The action or actions in the transaction call the resources. The process automatically enlists the appropriate resource managers to work with the resource at 304. The resource manager checks to determine if it has already been enlisted in the transaction, and if not, the resource manager enlists before the corresponding action is performed. A resource manager 210 manages memory changes affected under the control of transactional memory. The actions can be then applied to the resources at 306. Other actions in the transaction are operated through a properly enlisted resource manager, also at 306.

[0027]   At or near the end of the transaction, the transaction manager 216 calls the commit protocol 218 to determine whether the transaction should be rolled back in the case of conflict with other threads or if the changes are committed at 308. The commit protocol 218 in one example can be a one-phase commit protocol such as when only one resource manager is enlisted into the transaction. The commit protocol 218 in another example can be a two-phase commit protocol including a prepare phase 310 and a commit phase 314.

[0028]   Specifically, upon receiving a request to commit from the application, the transaction manager 216 can use the two-phase commit protocol 218 that begins the prepare phase at 310 of all the enlisted participants by calling a method on each enlisted resource manager 208, 210 in order to obtain a corresponding vote on the transaction. Each of the resource managers 208, 210 will vote to either commit or roll back by calling a prepare method or rollback method, respectively. For example, the memory action resource manager 208 can vote to commit if there are no memory conflicts. In the second phase of the commit protocol 218, the transaction manager appropriately responds depending on whether it has received either votes to commit from all the resource managers or if it has received at least one vote to roll back. If the transaction manager receives unanimous votes to commit from all the resource managers at 312, which have all invoked the prepare method, the transaction manger 216 invokes a commit method for each resource manager at 314. The resource managers can then make the changes durable and complete the commit and then proceed at 316 after releasing the transaction 202. If any resource manager voted not to commit in the prepare phase at 312, the process acts upon the failure as follows.

[0029]   Failures caused by memory conflicts as identified with the memory resource manager 208 are distinguishable from failures identified with the non-memory resource managers enlisted in the transaction. For example, failures caused by memory conflicts can cause an automatic re-execution of the transaction at 318. In general, the application including the transaction 202 is oblivious to re-executions when they are ultimately successful. In some embodiments, failures identified with the other resource managers, such as non-memory resource manager 210, cause the transaction to abort and not re-execute. For example, if one or more of the non-memory resource managers vote against committing the transaction and the memory resource manager votes for committing the transaction, the transaction and does not re-execute. If the memory resource manager 208 votes against committing the transaction at 318, a roll back occurs at 318 to 302 via 320 regardless of whether any or all of the non-memory resource managers vote to commit. The re-execution in this example occurs because a failure in non-memory

resource managers could have been the result of a memory conflict. If every resource manager votes to commit the transaction, the transaction commits at 314.

[0030]   Figure 4 illustrates an example process 400 of the two-phase commit protocol, which can be implemented at or near the end of the transaction 202 and can be performed as elements 308 and 310 above. The actions of the memory resource manager 208 and the transaction manager 216 are illustrated with reference to concurrent timelines 402, 404. The memory resource manager 208 (and other resource managers, such as manager 210 or others if applicable) dispatches an enlistment notification 406 to the transaction manager 216 that it is interested in voting in a specific transaction. When the code comprising the transaction finishes, it notifies the transaction manager 216 that it is ready to begin the commit protocol 218. The transaction manager issues a prepare directive to the memory resource manager 208 and any other resource managers enlisted in the transaction at 408.

[0031]   After receipt of the prepare directive at 408, the resource managers 208, 210 begin the process of validating their respective objects, at 410. In the case of memory resource manager, this can mean that the memory resource manager determines if there were conflicts with other threads. The resource managers report the validation results in a vote provided to the transaction manager 216 as a vote at 412. If there are no conflicts, the memory resource manager 208 votes to commit. Otherwise, the memory resource manager 208 votes not to commit or vetoes any votes to commit from the other resource managers.

[0032]   If all of the resource managers vote to commit, the transaction manager 216 dispatches a commit directive to all of the resource managers enlisted in the transaction at 414, such as memory resource manager 208 and resource manager 210. Any memory changes as a result of the atomic block transaction are committed to the shared state, where they are exposed to other threads. In the case of the memory resource manager, the transactional memory will not expose any of the isolated work of the transaction until receiving the commit directive. Such commit work can include release locks, write back buffered memory changes, or undo tentative memory changes. Once the actions of the transaction are committed

at 416, the application proceeds. The process 400 is able to process other commit requests of the application waiting in a queue at 418.

[0033]   In certain examples, the actions of the two-phase commit protocol occur in the order of process 400. In these examples, the transaction manager commit order is synchronized with the prepare directive and the receipt of the vote from the resource manager. In another example, the transactional memory processing system can convert to pessimistic locking and avoid participation in the process 400. Alternatively, a transactional memory processing system that can ensure the memory involved in the transaction is disjoint from all other memory involved in transactional memory atomic blocks can perform the actions out of order.

[0034]   The transactional memory processing system 200 can also provide for the use of transactions within transaction, or nested transactions. In such cases, the commit protocol 218 can cause the nesting to be flattened or respected as in the case of database transactions. In flattening, the transaction with the nested transaction appears as if it were one atomic block. If the transaction is rolled back, the entire transaction including the nested transaction will re-execute. Alternatively, the transactional memory processing system can support partial rollback, where the nested transaction is re-executed until committed.

[0035]   Combining transaction processing with the transactional memory provides benefits to productivity. A developer can rely on preprogrammed code of resource managers, rather than writing specialized code, for error processing and achieving failure atomicity. All memory accesses can be automatically included into the scope of an atomic block transaction by the transactional memory implementation because the lexical scopes of the atomic block and the transaction coincide. It also saves the developer from having to write code to address all of the combination of cases when some actions succeed and others do not.

[0036]   Transactional resources are supported on platforms supported by Microsoft, Inc., of Redmond, Washington, which can include databases such as that sold under the trade designation of Microsoft SQL Server, its transactional message queues (MSMQ), as well as that available under the trade designation of Windows Vista Transactional File System and Transactional Registry. Transactions can be

created and managed using transaction management products such as those provided by Microsoft, Inc. under the trade designations of Distributed Transaction Coordinator (DTC) available in the Windows operating system since version NT4, MTS in versions of Windows prior to Windows XP, or the Kernel Transaction

5   Manager (KTM) first available in Windows Vista. In managed code such as .NET available from Microsoft, the System Transactions feature provides management application program interfaces that use the DTC and provides its own lightweight transaction manager (LTM). The LTM also provides mechanisms to create volatile resource managers, such as memory resource managers, that participate in the

10   transactions with other resource managers such as database SQL.

[0037]   Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that a variety of alternate and/or equivalent implementations may be substituted for the specific embodiments shown and described without departing from the scope of the present invention.

15   This application is intended to cover any adaptations or variations of the specific embodiments discussed herein. Therefore, it is intended that this invention be limited only by the claims and the equivalents thereof.

WHAT IS CLAIMED IS:

1.   A method of controlling an atomic transaction 202 having access to a shared memory, the method comprising:

enlisting 304 at least one non-memory resource manager 210;

5   enlisting 304 a memory resource manager 208 configured to manage a transactional memory 212;

calling a commit protocol configured to determine whether the non-memory and memory resource managers 208, 210 vote to commit the transaction 308;

committing the transaction if the non-memory and memory resource

10   managers 208, 210 vote to commit the transaction 314;

re-executing the transaction 302, 318 if the memory resource manager 208 does not vote to commit the transaction 312; and

aborting the transaction 318 if at least one of the non-memory resource managers 208 does not vote to commit the transaction 312.

15   2.   The method of claim 1 wherein the shared memory includes an original state prior to enlisting the memory resource manager and wherein the re-executing the transaction includes rolling the shared memory back to the original state.

3.   The method of claim 2 wherein changes to the original state are made durable and in the public state upon committing the transaction.

20   4.   The method of claim 1 wherein re-executing the transaction includes repeatedly re-executing the transaction until all resource managers in the transaction vote to commit or the transaction is aborted.

5.   The method of claim 1 wherein the memory resource manager votes to commit if no memory conflicts occurs during the transaction.

25   6.   The method of claim 1 wherein re-executing the transaction occurs when the non-memory resource managers vote to commit the transaction and at least one memory resource manager votes to abort.

7.   The method of claim 1 wherein the commit protocol includes a prepare phase and a commit phase.

30   8.   The method of claim 1 including defining boundaries of the atomic transaction.

9.   The method of claim 1 wherein the atomic transaction includes a nested atomic transaction.

10.   The method of claim 9 wherein the commit protocol flattens the atomic transaction and the nested atomic transaction.

11.   The method of claim 9 wherein the nested atomic transaction is executed until it is committed before the atomic transaction is executed.

12.   The method of claim 1 wherein the resource managers are volatile resource managers.

13.   A runtime environment operating on a computing device 100 and configured to operate an application having an atomic transaction 202 having access to a shared memory, the runtime environment comprising:

        a memory control mechanism 212 configured to control access to variables in the shared memory through optimistic concurrency;

        a library of resource managers 208, 210, wherein at least two of the resource managers are configured to be enlisted in the transaction and loaded into a memory 104 on the computing device 100, wherein the at least two of the resource managers includes a memory resource manager 208 configured to enlist in managing the memory control mechanism 212; and

        a transaction manager 216 loaded into the volatile memory 104 and coupled to the enlisted resource managers 304, 208, 210, wherein the transaction manager 216 is configured to receive a vote 218 from the enlisted resource managers 208, 210 as to whether to commit the transaction 310, wherein the memory resource manager 208 votes depending on whether a memory conflicts occurred during the transaction 312.

14.   The runtime environment of claim 13 wherein the computing device includes a plurality of processors arranged in one of a plurality of classes of parallelism, and wherein the atomic transaction is included on a thread of an application having a plurality of threads running on at least two of the processors.

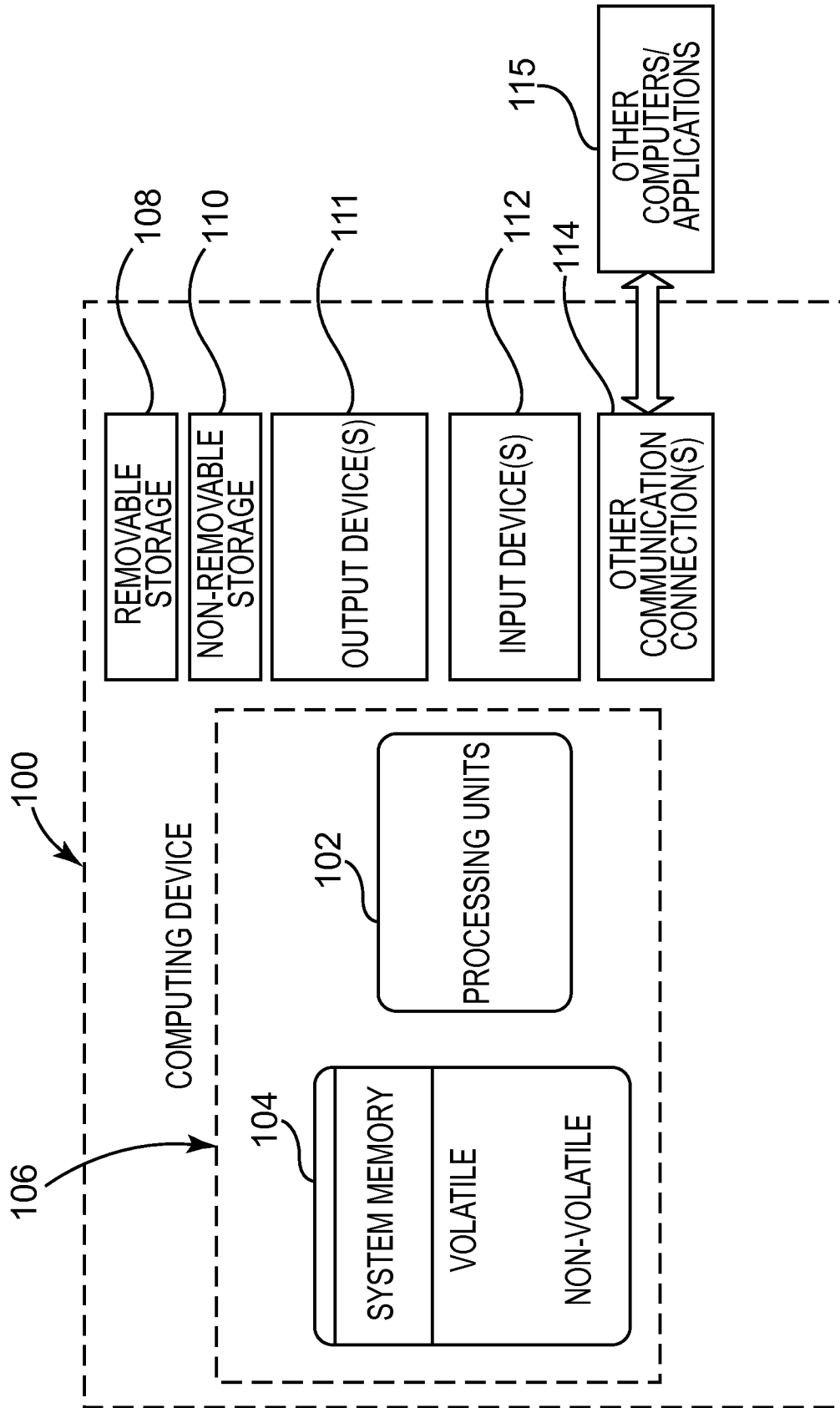15.   The runtime environment of claim 13 wherein the memory control mechanism is a transactional memory.
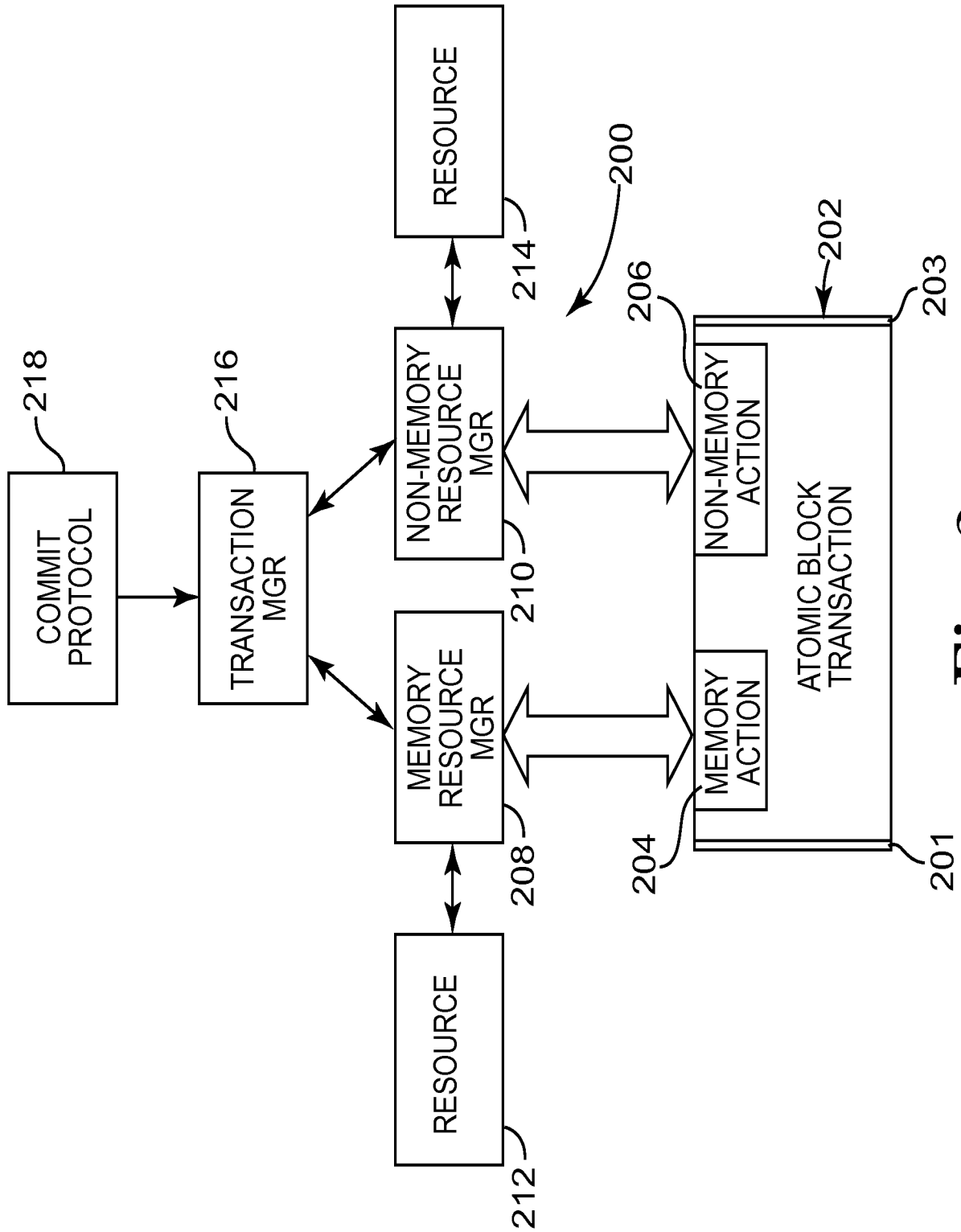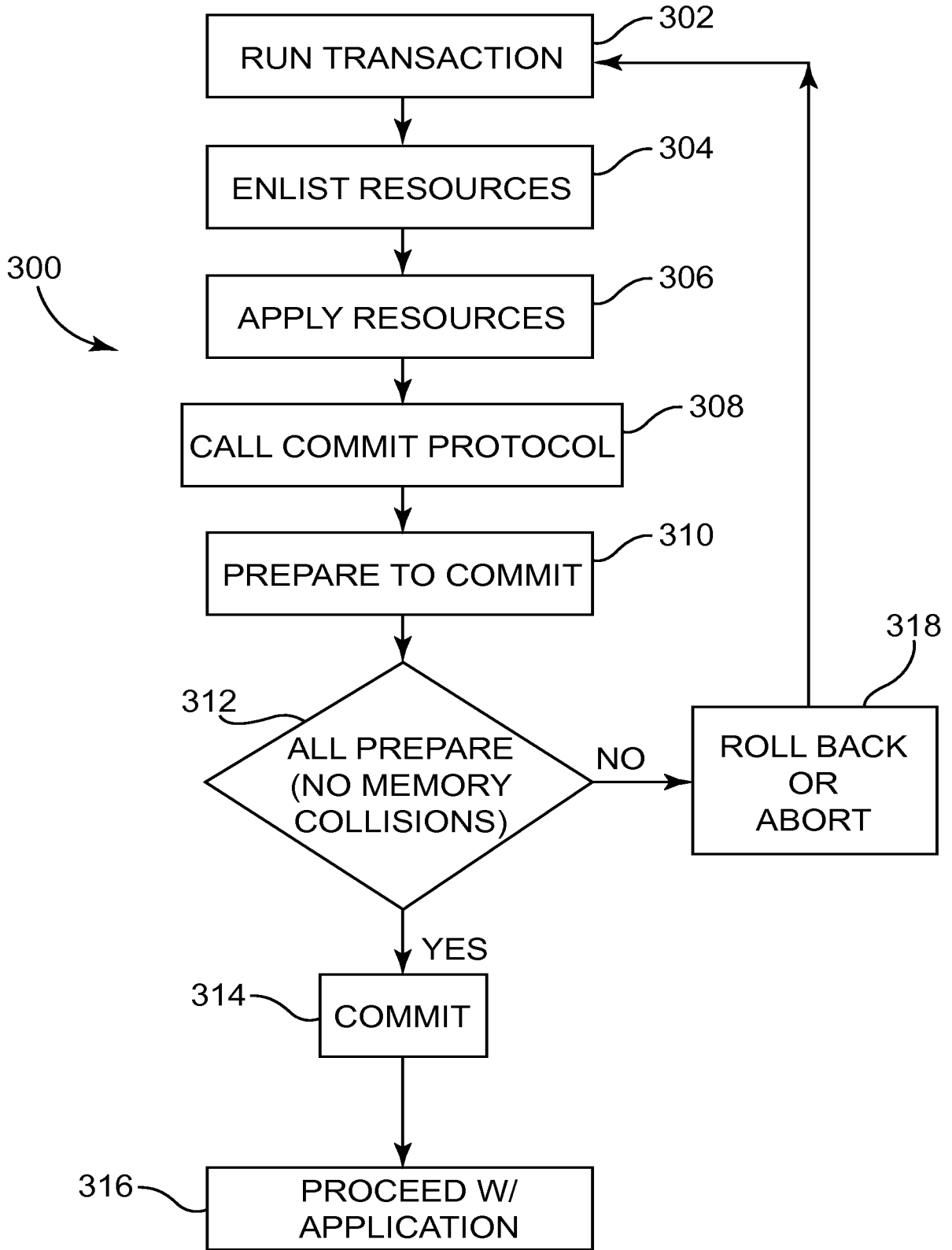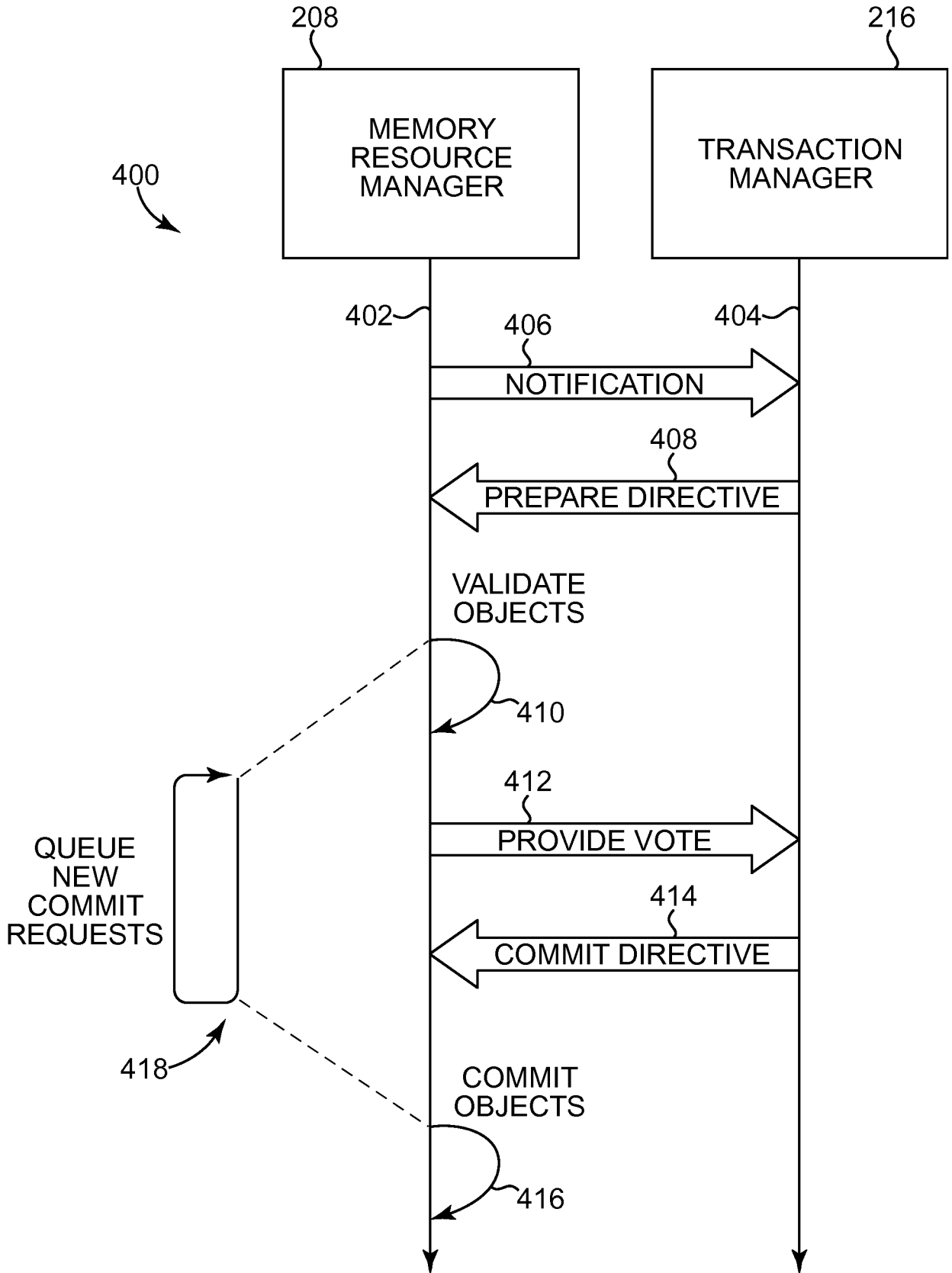
Fig. 1

Fig. 2

Fig. 3

Fig. 4