



(51) International Patent Classification:

G06F 9/06 (2006.01) **G06F 15/00** (2006.01)
G06F 9/44 (2006.01) **G06F 13/00** (2006.01)
G06F 9/46 (2006.01)

(21) International Application Number:

PCT/US2009/048947

(22) International Filing Date:

26 June 2009 (26.06.2009)

(25) Filing Language:

English

(26) Publication Language:

English

(71) Applicant (for all designated States except US): **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, MS: RNB-4-150, Santa Clara, CA 95052 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **SHEAFFER, Gad** [IL/IL]; 80 Grinboim, 34987 Haifa (IL). **GRAY, Jan** [US/US]; 13658 NE 37th Pl, Bellevue, WA 98005 (US). **SMITH, Burton** [US/US]; 923 21st Ave E, Seattle, WA 98112 (US). **ADL-TABATABAI, Ali-Reza** [US/US]; 5609 Morningside Dr., San Jose, CA 95138 (US). **GEVA, Robert** [US/US]; 10640 Madrid Rd, Cupertino, CA 95014 (US). **BASSIN, Vadim** [IL/IL]; 58, Ahuza St, 43209 Raanana (IL). **CALLAHAN, David** [US/US];

10652 Bartlett Ave NE, Seattle, WA 98125 (US). **NI, Yang** [US/US]; 1642 Peacock Ave, Sunnyvale, CA 98086 (US). **SAHA, Bratin** [US/US]; 931 Cole Place, Santa Clara, CA 95054 (US). **TAILLEFER, Martin** [US/US]; 26345 NE 34th Street, Redmond, WA 98053 (US). **RAIKIN, Shlomo** [IL/IL]; POB 164, 30855 Geva Carmel (IL). **YAMADA, Koichi** [US/US]; 982 Cherry-stone Drive, Los Gatos, CA 95032 (US). **WANG, Landy** [US/US]; 10250 NE 65 St, Kirkland, WA 98033 (US). **KISHAN, Arun** [US/US]; 300 110th Ave NE Unit 310, Bellevue, WA 98004 (US).

(74) Agents: **VINCENT, Lester, J.** et al.; Blakely Sokoloff Taylor & Zafman, 1279 Oakmead Parkway, Sunnyvale, CA 94085 (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.

[Continued on next page]

(54) Title: OPTIMIZATIONS FOR AN UNBOUNDED TRANSACTIONAL MEMORY (UTM) SYSTEM

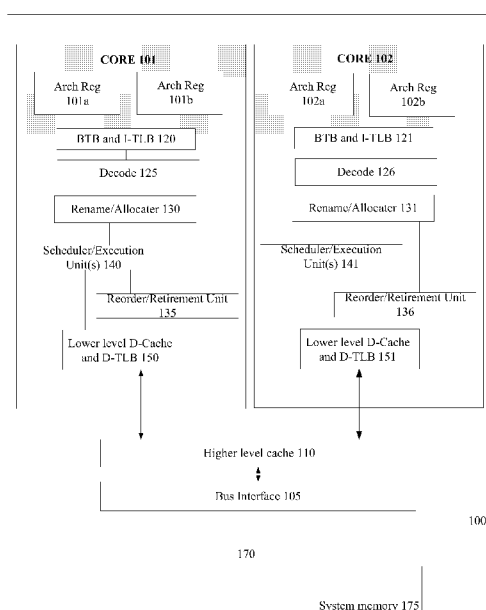


FIG. 1

(57) Abstract: A method and apparatus for optimizing an Unbounded Transactional Memory (UTM) system is herein described. Hardware support for monitors, buffering, and metadata is provided, where orthogonal metaphysical address spaces for metadata may be separate associated with threads and/or software subsystems within threads. In addition, the metadata may be held with hardware in a compressed manner with regard to data transparently to software. Furthermore, in response to metadata access instruction/operations the hardware is capable of supporting a forced metadata value to enable multiple modes of transactional execution. However, if monitors, buffered data, metadata, or other information is lost or conflicts are detected hardware provides for variations of a loss instruction that is able to poll a transaction status register for such loss or conflict and jump execution to a label in response to detecting the loss or conflict. Similarly, multiple variations of a commit instruction are provided for to allow software to define commit conditions and information to clear upon commit. Furthermore, hardware provides support to enable suspension and resume of transactions upon ring level transitions.



(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, TR),

OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

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

OPTIMIZATIONS FOR AN UNBOUNDED TRANSACTIONAL MEMORY (UTM) SYSTEM

FIELD

5 This invention relates to the field of processor execution and, in particular, to execution of groups of instructions.

BACKGROUND

Advances in semi-conductor processing and logic design have permitted an increase in the amount of logic that may be present on integrated circuit devices. As a
10 result, computer system configurations have evolved from a single or multiple integrated circuits in a system to multiple cores and multiple logical processors present on individual integrated circuits. A processor or integrated circuit typically comprises a single processor die, where the processor die may include any number of cores or logical processors.

15 The ever increasing number of cores and logical processors on integrated circuits enables more software threads to be concurrently executed. However, the increase in the number of software threads that may be executed simultaneously have created problems with synchronizing data shared among the software threads. One common solution to accessing shared data in multiple core or multiple logical processor systems comprises the
20 use of locks to guarantee mutual exclusion across multiple accesses to shared data. However, the ever increasing ability to execute multiple software threads potentially results in false contention and a serialization of execution.

For example, consider a hash table holding shared data. With a lock system, a programmer may lock the entire hash table, allowing one thread to access the entire hash
25 table. However, throughput and performance of other threads is potentially adversely affected, as they are unable to access any entries in the hash table, until the lock is released. Alternatively, each entry in the hash table may be locked. Either way, after extrapolating this simple example into a large scalable program, it is apparent that the complexity of lock contention, serialization, fine-grain synchronization, and deadlock
30 avoidance become extremely cumbersome burdens for programmers.

Another recent data synchronization technique includes the use of transactional memory (TM). Often transactional execution includes executing a grouping of a plurality of micro-operations, operations, or instructions. In the example above, both threads

execute within the hash table, and their memory accesses are monitored/tracked. If both threads access/alter the same entry, conflict resolution may be performed to ensure data validity. One type of transactional execution includes Software Transactional Memory (STM), where tracking of memory accesses, conflict resolution, abort tasks, and other transactional tasks are performed in software, often without the support of hardware.

Another type of transactional execution includes a Hardware Transactional Memory (HTM) System, where hardware is included to support access tracking, conflict resolution, and other transactional tasks. Previously, actual memory data arrays were extended with additional bits to hold information, such as hardware attributes to track reads, writes, and buffering, and as a result, the data travels with the data from the processor to memory. Often this information is referred to as persistent, i.e. it is not lost upon a cache eviction, since the information travels with data throughout the memory hierarchy. Yet, this persistency imposes more overhead throughout the memory hierarchy system.

In addition, previous hardware transactional memory (HTM) systems have been fraught with a number of inefficiencies. As a first example, HTMs currently provide no efficient method for transitioning between un-buffered or buffered and not monitored states to a buffered and monitored state to ensure consistency before commit of a transaction. As another example, multiple inefficiencies of a HTM's interface with software exist. Specifically, hardware provides no mechanism to properly accelerate software memory access barriers, which take into account different forms of strong and weak atomicity between transactional and non-transactional operations. In addition, during an attempted commit of a transaction, hardware does not provide any facilities for determining when a transaction is to abort or commit based on loss of monitoring, buffering, and/or other attribute information. Similarly, the instruction set for these previous HTMs do not provide for commit instructions that define information to retain, or clear, upon commit of a transaction. Other exemplary inefficiencies include: HTMs not providing instructions to efficiently vector or jump execution upon detection of a conflict or loss of information and the inability of current HTMs to handle ring level priority transitions during execution of transactions.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not intended to be

limited by the figures of the accompanying drawings.

Figure 1 illustrates an embodiment of a processor including multiple processing elements capable of executing multiple software threads concurrently.

Figure 2 illustrates an embodiment of associating metadata for a data item.

5 Figure 3 illustrates an embodiment of multiple orthogonal metaphysical address spaces for separate software subsystems within a plurality of processing elements.

Figure 4 illustrates an embodiment of compression of metadata to data.

Figure 5 illustrates an embodiment of a flow diagram for a method of accessing metadata.

10 Figure 6 illustrates an embodiment of a metadata storage element to support acceleration of transactions within strong and weak atomicity environments.

Figure 7 illustrates an embodiment of a flow diagram for accelerating non-transactional operations while maintaining atomicity in a transactional environment.

15 Figure 8 illustrates an embodiment of a flow diagram for a method efficiently transitioning a block of data to a buffered and monitored state before commit of a transaction.

Figure 9 illustrates an embodiment of hardware to support a loss instruction to jump to a destination label based upon a status value in a transaction status register.

20 Figure 10 illustrates an embodiment of a flow diagram for a method of executing a loss instruction to jump to a destination label based upon a conflict or loss of specific information.

Figure 11 illustrates an embodiment of hardware to support definition of commit conditions and clear controls in a commit instruction.

25 Figure 12 illustrates an embodiment of a flow diagram for a method of executing a commit instruction, which defines commit conditions and clear controls.

Figure 13 illustrates an embodiment of hardware to support handling privilege level transitions during execution of transactions.

DETAILED DESCRIPTION

30 In the following description, numerous specific details are set forth such as examples of specific hardware structures for transactional execution, specific types and implementations of access monitors, specific types cache coherency models to detect access conflicts, specific data granularities, and specific types of memory accesses and locations, etc. in order to provide a thorough understanding of the present invention. It

will be apparent, however, to one skilled in the art that these specific details need not be employed to practice the present invention. In other instances, well known components or methods, such as coding of transactions in software, insertion of operations to perform enumerated functions by a compiler, demarcation of transactions, specific and alternative
5 multi-core and multi-threaded processor architectures, specific compiler methods/implementations, and specific operational details of microprocessors, have not been described in detail in order to avoid unnecessarily obscuring the present invention.

The method and apparatus described herein are for optimizing hardware and software for unbounded transactional memory (UTM) execution. Specifically, the
10 optimizations are primarily discussed in reference to a supporting a UTM system. However, the methods and apparatus described herein may be utilized within any form a transactional memory system, such as within hardware to support or accelerate software transactional memory systems (STMs), pure hardware transactional memory systems (HTMs), or a hybrid thereof, which differs in implementation from a UTM system.

Referring to **Figure 1**, an embodiment of a processor capable of executing multiple
15 threads concurrently is illustrated. Note, processor **100** may include hardware support for hardware transactional execution. Either in conjunction with hardware transactional execution, or separately, processor **100** may also provide hardware support for hardware acceleration of a Software Transactional Memory (STM), separate execution of a STM, or
20 a combination thereof, such as a hybrid Transactional Memory (TM) system. Processor **100** includes any processor, such as a micro-processor, an embedded processor, a digital signal processor (DSP), a network processor, or other device to execute code. Processor **100**, as illustrated, includes a plurality of processing elements.

In one embodiment, a processing element refers to a thread unit, a process unit, a
25 context, a logical processor, a hardware thread, a core, and/or any other element, which is capable of holding a state for a processor, such as an execution state or architectural state. In other words, a processing element, in one embodiment, refers to any hardware capable of being independently associated with code, such as a software thread, operating system, application, or other code. A physical processor typically refers to an integrated circuit,
30 which potentially includes any number of other processing elements, such as cores or hardware threads.

A core often refers to logic located on an integrated circuit capable of maintaining an independent architectural state wherein each independently maintained architectural

state is associated with at least some dedicated execution resources. In contrast to cores, a hardware thread typically refers to any logic located on an integrated circuit capable of maintaining an independent architectural state wherein the independently maintained architectural states share access to execution resources. As can be seen, when certain
5 resources are shared and others are dedicated to an architectural state, the line between the nomenclature of a hardware thread and core overlaps. Yet often, a core and a hardware thread are viewed by an operating system as individual logical processors, where the operating system is able to individually schedule operations on each logical processor.

Physical processor **100**, as illustrated in **Figure 1**, includes two cores, core **101** and
10 **102**, which share access to higher level cache **110**. Although processor **100** may include asymmetric cores, i.e. cores with different configurations, functional units, and/or logic, symmetric cores are illustrated. As a result, core **102**, which is illustrated as identical to core **101**, will not be discussed in detail to avoid repetitive discussion. In addition, core **101** includes two hardware threads **101a** and **101b**, while core **102** includes two hardware
15 threads **102a** and **102b**. Therefore, software entities, such as an operating system, potentially view processor **100** as four separate processors, i.e. four logical processors or processing elements capable of executing four software threads concurrently.

Here, a first thread is associated with architecture state registers **101a**, a second thread is associated with architecture state registers **101b**, a third thread is associated with
20 architecture state registers **102a**, and a fourth thread is associated with architecture state registers **102b**. As illustrated, architecture state registers **101a** are replicated in architecture state registers **101b**, so individual architecture states/contexts are capable of being stored for logical processor **101a** and logical processor **101b**. Other smaller resources, such as instruction pointers and renaming logic in rename allocator logic **130**
25 may also be replicated for threads **101a** and **101b**. Some resources, such as re-order buffers in reorder/retirement unit **135**, ILTB **120**, load/store buffers, and queues may be shared through partitioning. Other resources, such as general purpose internal registers, page-table base register, low-level data-cache and data-TLB **115**, execution unit(s) **140**, and portions of out-of-order unit **135** are potentially fully shared.

30 Processor **100** often includes other resources, which may be fully shared, shared through partitioning, or dedicated by/to processing elements. In **Figure 1**, an embodiment of a purely exemplary processor with illustrative functional units/resources of a processor is illustrated. Note that a processor may include, or omit, any of these functional units, as

well as include any other known functional units, logic, or firmware not depicted.

As illustrated, processor **100** includes bus interface module **105** to communicate with devices external to processor **100**, such as system memory **175**, a chipset, a northbridge, or other integrated circuit. Memory **175** may be dedicated to processor **100** or shared with other devices in a system. Higher-level or further-out cache **110** is to cache recently fetched elements from higher-level cache **110**. Note that higher-level or further-out refers to cache levels increasing or getting further way from the execution unit(s). In one embodiment, higher-level cache **110** is a second-level data cache. However, higher level cache **110** is not so limited, as it may be associated with or include an instruction cache. A trace cache, i.e. a type of instruction cache, may instead be coupled after decoder **125** to store recently decoded traces. Module **120** also potentially includes a branch target buffer to predict branches to be executed/taken and an instruction-translation buffer (I-TLB) to store address translation entries for instructions.

Decode module **125** is coupled to fetch unit **120** to decode fetched elements. In one embodiment, processor **100** is associated with an Instruction Set Architecture (ISA), which defines/specifies instructions executable on processor **100**. Here, often machine code instructions recognized by the ISA include a portion of the instruction referred to as an opcode, which references/specifies an instruction or operation to be performed.

In one example, allocator and renamer block **130** includes an allocator to reserve resources, such as register files to store instruction processing results. However, threads **101a** and **101b** are potentially capable of out-of-order execution, where allocator and renamer block **130** also reserves other resources, such as reorder buffers to track instruction results. Unit **130** may also include a register renamer to rename program/instruction reference registers to other registers internal to processor **100**. Reorder/retirement unit **135** includes components, such as the reorder buffers mentioned above, load buffers, and store buffers, to support out-of-order execution and later in-order retirement of instructions executed out-of-order.

Scheduler and execution unit(s) block **140**, in one embodiment, includes a scheduler unit to schedule instructions/operation on execution units. For example, a floating point instruction is scheduled on a port of an execution unit that has an available floating point execution unit. Register files associated with the execution units are also included to store information instruction processing results. Exemplary execution units include a floating point execution unit, an integer execution unit, a jump execution unit, a

load execution unit, a store execution unit, and other known execution units.

Lower level data cache and data translation buffer (D-TLB) **150** are coupled to execution unit(s) **140**. The data cache is to store recently used/operated on elements, such as data operands, which are potentially held in memory coherency states. The D-TLB is to store recent virtual/linear to physical address translations. As a specific example, a processor may include a page table structure to break physical memory into a plurality of virtual pages.

In one embodiment, processor **100** is capable of hardware transactional execution, software transactional execution, or a combination or hybrid thereof. A transaction, which may also be referred to as a critical or atomic section of code, includes a grouping of instructions, operations, or micro-operations to be executed as an atomic group. For example, instructions or operations may be used to demarcate a transaction or a critical section. In one embodiment, described in more detail below, these instructions are part of a set of instructions, such as an Instruction Set Architecture (ISA), which are recognizable by hardware of processor **100**, such as decoders described above. Often, these instructions, once compiled from a high-level language to hardware recognizable assembly language include operation codes (opcodes), or other portions of the instructions, that decoders recognize during a decode stage.

Typically, during execution of a transaction, updates to memory are not made globally visible until the transaction is committed. As an example, a transactional write to a location is potentially visible to a local thread, yet, in response to a read from another thread the write data is not forwarded until the transaction including the transactional write is committed. While the transaction is still pending, data items/elements loaded from and written to within a memory are tracked, as discussed in more detail below. Once the transaction reaches a commit point, if conflicts have not been detected for the transaction, then the transaction is committed and updates made during the transaction are made globally visible.

However, if the transaction is invalidated during its pendency, the transaction is aborted and potentially restarted without making the updates globally visible. As a result, pendency of a transaction, as used herein, refers to a transaction that has begun execution and has not been committed or aborted, i.e. pending.

A Software Transactional Memory (STM) system often refers to performing access tracking, conflict resolution, or other transactional memory tasks within or at least

partially within software. In one embodiment, processor **100** is capable of executing a compiler to compile program code to support transactional execution. Here, the compiler may insert operations, calls, functions, and other code to enable execution of transactions.

5 A compiler often includes a program or set of programs to translate source text/code into target text/code. Usually, compilation of program/application code with a compiler is done in multiple phases and passes to transform hi-level programming language code into low-level machine or assembly language code. Yet, single pass compilers may still be utilized for simple compilation. A compiler may utilize any known compilation techniques and perform any known compiler operations, such as lexical
10 analysis, preprocessing, parsing, semantic analysis, code generation, code transformation, and code optimization.

Larger compilers often include multiple phases, but most often these phases are included within two general phases: (1) a front-end, i.e. generally where syntactic processing, semantic processing, and some transformation/optimization may take place,
15 and (2) a back-end, i.e. generally where analysis, transformations, optimizations, and code generation takes place. Some compilers refer to a middle end, which illustrates the blurring of delineation between a front-end and back end of a compiler. As a result, reference to insertion, association, generation, or other operation of a compiler may take place in any of the aforementioned phases or passes, as well as any other known phases or
20 passes of a compiler. As an illustrative example, a compiler potentially inserts transactional operations, calls, functions, etc. in one or more phases of compilation, such as insertion of calls/operations in a front-end phase of compilation and then transformation of the calls/operations into lower-level code during a transactional memory transformation phase.

25 Nevertheless, despite the execution environment and dynamic or static nature of a compiler, the compiler, in one embodiment, compiles program code to enable transactional execution. Therefore, reference to execution of program code, in one embodiment, refers to (1) execution of a compiler program(s), either dynamically or statically, to compile main program code, to maintain transactional structures, or to
30 perform other transaction related operations, (2) execution of main program code including transactional operations/calls, (3) execution of other program code, such as libraries, associated with the main program code, or (4) a combination thereof.

Often within software transactional memory (STM) systems, a compiler will be

utilized to insert some operations, calls, and other code inline with application code to be compiled, while other operations, calls, functions, and code are provided separately within libraries. This potentially provides the ability of the libraries distributors to optimize and update the libraries without having to recompile the application code. As a specific
5 example, a call to a commit function may be inserted inline within application code at a commit point of a transaction, while the commit function is separately provided in an updateable library. Additionally, the choice of where to place specific operations and calls potentially affects the efficiency of application code. For example, if a filter operation, which is discussed in more detail regarding access barriers in reference to **Figure 6**, is
10 inserted inline with code, the filter operation may be performed before vectoring execution to a barrier instead of inefficiently vectoring to the barrier and then performing the filter operation.

In one embodiment, processor **100** is capable of executing transactions utilizing hardware/logic, i.e. within a Hardware Transactional Memory (HTM) system. Numerous
15 specific implementation details exist both from an architectural and microarchitectural perspective when implementing an HTM; most of which are not discussed herein to avoid unnecessarily obscuring the invention. However, some structures and implementations are disclosed for illustrative purposes. Yet, it should be noted that these structures and implementations are not required and may be augmented and/or replaced with other
20 structures having different implementation details.

As a combination, processor **100** may be capable of executing transactions within an unbounded transactional memory (UTM) system, which attempts to take advantage of the benefits of both STM and HTM systems. For example, an HTM is often fast and efficient for executing small transactions, because it does not rely on software to perform
25 all of the access tracking, conflict detection, validation, and commit for transactions. However, HTMs are usually only able to handle smaller transactions, while STMs are able to handle unbounded sized transactions. Therefore, in one embodiment, a UTM system utilizes hardware to execute smaller transactions and software to execute transactions that are too big for the hardware. As can be seen from the discussion below, even when
30 software is handling transactions, hardware may be utilized to assist and accelerate the software. Furthermore, it is important to note that the same hardware may also be utilized to support and accelerate a pure STM system.

As stated above, transactions include transactional memory accesses to data items

both by local processing elements within processor **100**, as well as potentially by other processing elements. Without safety mechanisms in a transactional memory system, some of these accesses would potentially result in invalid data and execution, i.e. a write to data invalidating a read, or a read of invalid data. As a result, processor **100** potentially
5 includes logic to track or monitor memory accesses to and from data items for identification of potential conflicts, such as read monitors and write monitors, as discussed below.

A data item or data element may include data at any granularity level, as defined by hardware, software or a combination thereof. A non-exhaustive list of examples of
10 data, data elements, data items, or references thereto, include a memory address, a data object, a class, a field of a type of dynamic language code, a type of dynamic language code, a variable, an operand, a data structure, and an indirect reference to a memory address. However, any known grouping of data may be referred to as a data element or data item. A few of the examples above, such as a field of a type of dynamic language
15 code and a type of dynamic language code refer to data structures of dynamic language code. To illustrate, dynamic language code, such as Java™ from Sun Microsystems, Inc, is a strongly typed language. Each variable has a type that is known at compile time. The types are divided in two categories - primitive types (boolean and numeric, e.g., int, float) and reference types (classes, interfaces and arrays). The values of reference types are
20 references to objects. In Java™, an object, which consists of fields, may be a class instance or an array. Given object a of class A it is customary to use the notation A::x to refer to the field x of type A and a.x to the field x of object a of class A. For example, an expression may be couched as a.x = a.y + a.z. Here, field y and field z are loaded to be added and the result is to be written to field x.

Therefore, monitoring/buffering memory accesses to data items may be performed
25 at any of data level granularity. For example, in one embodiment, memory accesses to data are monitored at a type level. Here, a transactional write to a field A::x and a non-transactional load of field A::y may be monitored as accesses to the same data item, i.e. type A. In another embodiment, memory access monitoring/buffering is performed at a
30 field level granularity. Here, a transactional write to A::x and a non-transactional load of A::y are not monitored as accesses to the same data item, as they are references to separate fields. Note, other data structures or programming techniques may be taken into account in tracking memory accesses to data items. As an example, assume that fields x and y of

object of class A, i.e. A::x and A::y, point to objects of class B, are initialized to newly allocated objects, and are never written to after initialization. In one embodiment, a transactional write to a field B::z of an object pointed to by A::x are not monitored as memory access to the same data item in regards to a non-transactional load of field B::z of an object pointed to by A::y. Extrapolating from these examples, it is possible to determine that monitors may perform monitoring/buffering at any data granularity level.

In one embodiment, processor **100** includes monitors to detect or track accesses, and potential subsequent conflicts, associated with data items. As one example, hardware of processor **100** includes read monitors and write monitors to track loads and stores, which are determined to be monitored, accordingly. As an example, hardware read monitors and write monitors are to monitor data items at a granularity of the data items despite the granularity of underlying storage structures. In one embodiment, a data item is bounded by tracking mechanisms associated at the granularity of the storage structures to ensure the at least the entire data item is monitored appropriately.

As a specific illustrative example, read and write monitors include attributes associated with cache locations, such as locations within lower level data cache **150**, to monitor loads from and stores to addresses associated with those locations. Here, a read attribute for a cache location of data cache **150** is set upon a read event to an address associated with the cache location to monitor for potential conflicting writes to the same address. In this case, write attributes operate in a similar manner for write events to monitor for potential conflicting reads and writes to the same address. To further this example, hardware is capable of detecting conflicts based on snoops for reads and writes to cache locations with read and/or write attributes set to indicate the cache locations are monitored, accordingly. Inversely, setting read and write monitors, or updating a cache location to a buffered state, in one embodiment, results in snoops, such as read requests or read for ownership requests, which allow for conflicts with addresses monitored in other caches to be detected.

Therefore, based on the design, different combinations of cache coherency requests and monitored coherency states of cache lines result in potential conflicts, such as a cache line holding a data item in a shared read monitored state and a snoop indicating a write request to the data item. Inversely, a cache line holding a data item being in a buffered write state and an external snoop indicating a read request to the data item may be considered potentially conflicting. In one embodiment, to detect such combinations of

access requests and attribute states snoop logic is coupled to conflict detection/reporting logic, such as monitors and/or logic for conflict detection/reporting, as well as status registers to report the conflicts.

However, any combination of conditions and scenarios may be considered
5 invalidating for a transaction, which may be defined by an instruction, such as a commit instruction, which is discussed below in more detail in reference to **Figures 11-12**. Examples of factors, which may be considered for non-commit of a transaction includes detecting a conflict to a transactionally accessed memory location, losing monitor
information, losing buffered data, losing metadata associated with a transactionally
10 accessed data item, and detecting an other invalidating event, such as an interrupt, ring transition, or an explicit user instruction.

In one embodiment, hardware of processor **100** is to hold transactional updates in a buffered manner. As stated above, transactional writes are not made globally visible until commit of a transaction. However, a local software thread associated with the
15 transactional writes is capable of accessing the transactional updates for subsequent transactional accesses. As a first example, a separate buffer structure is provided in processor **100** to hold the buffered updates, which is capable of providing the updates to the local thread and not to other external threads. Yet, the inclusion of a separate buffer structure is potentially expensive and complex.

20 In contrast, as another example, a cache memory, such as data cache **150**, is utilized to buffer the updates, while providing the same transactional functionality. Here, cache **150** is capable of holding data items in a buffered coherency state; in one case, a new buffered coherency state is added to a cache coherency protocol, such as a Modified Exclusive Shared Invalid (MESI) protocol to form a MESIB protocol. In response to local
25 requests for a buffered data item – data item being held in a buffered coherency state, cache **150** provides the data item to the local processing element to ensure internal transactional sequential ordering. However, in response to external access requests, a miss response is provided to ensure the transactionally updated data item is not made globally visible until commit. Furthermore, when a line of cache **150** is held in a buffered
30 coherency state and selected for eviction, the buffered update is not written back to higher level cache memories – the buffered update is not to be proliferated through the memory system, i.e. not made globally visible, until after commit. Upon commit, the buffered lines are transitioned to a modified state to make the data item globally visible.

Note that the terms internal and external are often relative to a perspective of a thread associated with execution of a transaction or processing elements that share a cache. For example, a first processing element for executing a software thread associated with execution of a transaction is referred to a local thread. Therefore, in the discussion
5 above, if a store to or load from an address previously written by the first thread, which results in a cache line for the address being held in a buffered coherency state, is received, then the buffered version of the cache line is provided to the first thread since it is the local thread. In contrast, a second thread may be executing on another processing element within the same processor, but is not associated with execution of the transaction
10 responsible for the cache line being held in the buffered state – an external thread; therefore, a load or store from the second thread to the address misses the buffered version of the cache line, and normal cache replacement is utilized to retrieve the unbuffered version of the cache line from higher level memory.

Here, the internal/local and external/remote threads are being executed on the same
15 processor, and in some embodiments, may be executed on separate processing elements within the same core of a processor sharing access to the cache. However, the use of these terms is not so limited. As stated above, local may refer to multiple threads sharing access to a cache, instead of being specific to a single thread associated with execution of the transaction, while external or remote may refer to threads not sharing access to the cache.

20 As stated above in the initial reference to **Figure 1**, the architecture of processor **100** is purely illustrative for purpose of discussion. Similarly, the specific examples of translating data addresses for referencing metadata is also exemplary, as any method of associating data with metadata in separate entries of the same memory may be utilized.

Metaphysical Address Spaces for Metadata

25 **Metadata**

Turning to **Figure 2**, an embodiment of holding metadata for a data item in a processor is illustrated. As depicted, metadata **217** for data item **216** is held locally in memory **215**. Metadata includes any property or attribute associated with data item **216**, such as transactional information relating to data item **216**. Some illustrative examples of
30 metadata are included below; yet the disclosed examples of metadata are purely illustrative and do not included an exhaustive list. In addition, metadata location **217** may hold any combination of the examples discussed below and other attributes for data item **216**, which are not specifically discussed.

As a first example, metadata **217** includes a reference to a backup or buffer location for transactionally written data item **216**, if data item **216** has been previously accessed, buffered and/or backed up within a transaction. Here, in some implementations a backup copy of a previous version of data item **216** is held in a different location, and as a result, metadata **217** includes an address, or other reference, to the backup location. Alternatively, metadata **217** itself may act as a backup or buffer location for data item **216**.

As another example, metadata **217** includes a filter value to accelerate repeat transactional accesses to data item **216**. Often, during execution of a transaction utilizing software, access barriers are performed at transactional memory accesses to ensure consistency and data validity. For example, before a transactional load operation a read barrier is executed to perform read barrier operations, such testing if data item **216** is unlocked, determining if a current read set of the transaction is still valid, updating a filter value, and logging of version values in the read set for the transaction to enable later validation. However, if a read of that location has already been performed during execution of the transaction, then the same read barrier operations are potentially unnecessary.

As a result, one solution includes utilizing a read filter to hold a first default value to indicate data item **216**, or the address therefore, has not been read during execution of the transaction and a second accessed value to indicate that data item **216**, or the address therefore, has already been accessed during a pendency of the transaction. Essentially, the second accessed value indicates whether the read barrier should be accelerated. In this instance, if a transactional load operation is received and the read filter value in metadata location **217** indicates that data item **216** has already been read, then, in one embodiment, the read barrier is elided – not executed – to accelerate the transactional execution by not performing unnecessary, redundant read barrier operations. Note that a write filter value may operate in the same manner with regard to write operations. However, individual filter values are purely illustrative, as, in one embodiment, a single filter value is utilized to indicate if an address has already been accessed – whether written or read. Here, metadata access operations to check metadata **217** for **216** for both loads and stores utilize the single filter value, which is in contrast to the examples above where metadata **217** includes a separate read filter value and write filter value. As a specific illustrative embodiment, four bits of metadata **217** are allocated to a read filter to indicate if a read barrier is to be accelerated in regards to an associated data item, a write filter to indicate if

a write barrier is to be accelerated in regards to an associated data item, an undo filter to indicate undo operations are to be accelerated, and a miscellaneous filter to be utilized in any manner by software as a filter value.

A few other examples of metadata include an indication of, representation of, or a reference to an address for a handler – either generic or specific to a transaction associated with data item **216**, an irrevocable/obstinate nature of a transaction associated with data item **216**, a loss of data item **216**, a loss of monitoring information for data item **216**, a conflict being detected for data item **216**, an address of a read set or read entry within a read set associated with data item **216**, a previous logged version for data item **216**, a current version of data item **216**, a lock for allowing access to data item **216**, a version value for data item **216**, a transaction descriptor for the transaction associated with data item **216**, and other known transaction related descriptive information. Furthermore, as described above, use of metadata is not limited to transactional information. As a corollary, metadata **217** may also include information, properties, attributes, or states associated with data item **216**, which are not involved with a transaction.

Continuing the discussion of illustrations for metadata, the hardware monitors and buffered coherency states described above are also considered metadata in some embodiments. The monitors indicate whether a location is to be monitored for external read requests or external read for ownership requests, while the buffered coherency state indicates if an associated data cache line holding a data item is buffered. Yet, in the examples above, monitors are maintained as attribute bits, which are appended to or otherwise directly associated with cache lines, while the buffered coherency state is added to cache line coherency state bits. As a result, in that case, hardware monitors and buffered coherency states are part of the cache line structure, not held in a separate metaphysical address space, such as illustrated metadata **217**. However, in other embodiments monitors may be held as metadata **217** in a separate memory location from data item **216**, and similarly, metadata **217** may included a reference to indicate that data item **216** is a buffered data item. Conversely, instead of an update-in-place architecture, where data item **216** is updated and held in a buffered state as described above, metadata **217** may hold the buffered data item, while the globally visible version of data item **216** is maintained in its original location. Here, upon commit the buffered update held in metadata **217** replaces data item **216**.

Lossy Metadata

Similar to the discussion above with reference to buffered cache coherency states, metadata **217**, in one embodiment, is lossy – local information that is not provided to external requests outside memory **215**'s domain. Assuming for one embodiment that memory **215** is a shared cache memory, then a miss in response to a metadata access operation is not serviced outside cache memory **215**'s domain. Essentially, since lossy metadata **217** is only held locally within the cache domain and does not exist as persistent data through out the memory subsystem, there is no reason to forward the miss externally to service the request from a higher-level memory. As a result, misses to lossy metadata are potentially serviced in a quick and efficient fashion; immediate allocation of memory in the processor may be allocated without waiting for an external request for the metadata to be generated or serviced..

Metaphysical Address Space

As the illustrated embodiment depicts, metadata **217** is held in a separate memory location – a distinct address – from data item **216**, which results in a separate metaphysical address space for metadata; the metaphysical address space being orthogonal to the data address space – a metadata access operations to the metaphysical address space does not hit or modify a physical data entry. However, in the embodiment where metadata is held within the same memory, such as memory **215**, the metaphysical address space potentially affects the data address space through competition for allocation in memory **215**. As an example, a data item **216** is cached in an entry of memory **215**, while metadata **217** for data **216** is held in another entry of the cache. Here, a subsequent metadata operation may result in the selection of data item **216**'s memory location for eviction and replacement with metadata for a different data item. As a result, operations associated with metadata **217**'s address do not hit data item **216**, however, a metadata address for a metadata element may replace physical data, such as data item **216** within memory **215**.

Even though, in this example, metadata potentially competes with data for space in the cache memory, the ability to hold metadata locally potentially results in efficient support for metadata without expensive cost of proliferating persistent metadata throughout a memory hierarchy. As inferred by the assumption of this example – that metadata is held in the same memory, memory **215**; however, in an alternative embodiment, metadata **217** for/associated with data item **216** is held in a separate memory structure. Here, addresses for metadata and data may be the same, while a metaphysical portion of the metadata address indexes into the separate metadata storage structure

instead of the data storage structure.

In a one-to-one ratio of metadata to data, a metaphysical address space shadows the data address space, but remains orthogonal as discussed above. In contrast, as discussed below, metadata may be compressed with regard to physical data. In this case, the size of a metaphysical address space for metadata does not shadow the data address space in size, but still remains orthogonal.

Metaphysical Address Translation

Continuing the discussion of metaphysical address spaces, any method of translating a data address, such as an address for data item **216**, within a data address space to a metaphysical address, such as a metadata address for metadata **217**, within a metaphysical address space may be utilized. In one embodiment, metaphysical translation logic **210** is utilized to translate an address, such as data address **200** to a metadata address. As depicted address **200** includes an address that is associated with, or references, data item **216**. Normal data translation, such as translation between physical, or linear, and virtual addresses may be utilized to index to data item **216** within memory **215**. In addition, association of metadata **217** with data item **216** includes similar translation of address **200**, which references data item **216**, into another distinct address that references metadata **217**; therefore, translation of address **200** into a data address with data translation logic **205** and a distinct metaphysical address with metaphysical translation **210** results in separate accesses without interference from one another – creating the orthogonal nature of the two address spaces. As discussed in more detail below, use of data translation **205** or metaphysical translation **210**, in one embodiment, is based on the type of operation to access address **200** – a normal data access operation to access data item **216** utilizes data translation **205**, while a metadata access operation to access metadata **217** utilizes metaphysical translation **210**, which may be identified through a portion of instruction/operation operation code (opcode).

In another embodiment, an instruction, as identified by its opcode, may potentially access both data and metadata for a given metadata address, and thus, perform complex operations, such as a conditional store to data based on metadata. As an example, an instruction is decoded into a test and set metadata operation to test metadata and set it to a value, as well as an additional operation to set data to a value if the test of metadata succeeded. As another example, a data item may be moved based on a data read from data memory to the matching metadata address.

Examples of translating data address **200** to a metadata address for metadata **217** are included immediately below. As a first example, translating a data address to a metadata address includes utilizing a physical address or a virtual address – after normal data translation **205** – plus addition of a metaphysical value with metaphysical translation logic **210** to separate data addresses from metadata address. In the situation where a virtual address is utilized without translation, metaphysical translation logic **210** includes logic to combine the virtual address with a metaphysical value. However, in the case where normal virtual to physical address translation is utilized, then normal data translation **205** is utilized to obtain a translated address from address **200** and then metaphysical translation logic **210** includes logic to combine the translated address with a metaphysical value to form a metadata address. As another example, data address **200** may be translated utilizing separate translation structures, tables, and/or logic within metaphysical translation **210** to obtain a distinct metadata address. Here, metaphysical translation logic **210** may mirror, or include separate logic – logic to combine address **200** with a metaphysical value, in comparison to data translation logic **205**, but metaphysical translation logic **210** includes page table information to translate address **200** to a different, distinct metadata address. It can be seen that either through addition of information in, extension with information appended to, replacement of information within, or translation of a data address to obtain a metadata address, the resulting distinct metadata address is associated with the data item through the algorithm of addition, extension, replacement, or translation, while remaining orthogonal from incorrectly updating or reading the data item.

A few specific illustrative examples of translating a data address to a metadata address, or in other words determining a metadata address from/based on a data address, are described below. (1) translating a first data address to a second data address utilizing normal virtual to physical address translation and adding, appending, or including a metaphysical value to or within the data address to form the metadata address;

(2) not performing virtual to physical address translation on the data address and adding, appending, or including a metaphysical value to or within the data address to form the metadata address; (3) translating a data address to a translated metadata address utilizing metaphysical translation table logic, which may also include, but is not required to include, adding, appending, or including a metaphysical value to or within the translated metadata address to form the metadata address. Furthermore, any of the

aforementioned translation techniques may incorporate, i.e. be based on, a compression ratio of data to metadata so as to separately store metadata for each compression ratio.

Here, an address may be modified for translation and/or compression, such as though disregarding specific bits of an address, removing specific bits of an address, changing what bit ranges with an address are used for selection of different granularities of data, translating specific bits, and adding or replacing specific bits with metadata related information. Compression is discussed in more detail below in reference to **Figure 4**.

Multiple Metaphysical Address Spaces

Turning to **Figure 3** an embodiment of supporting multiple metaphysical address spaces is illustrated. In one embodiment, each processing element is associated with a metaphysical address space, such that each processing element is capable of maintaining independent metadata. Four processing elements **301-304** are depicted. As discussed above, a processing element may encompass any of the elements described above in reference to **Figure 1**. As a first example, processing elements include cores of a processor. However, as an illustrative example to further the discussion below, processing elements **301-304** will be discussed in reference to hardware threads (threads) within a processor; each hardware thread to execute a software thread and potentially multiple software subsystems.

Therefore, it is potentially advantageous to allow individual threads of threads **301-304** to maintain separate metadata. In one embodiment, metaphysical translation logic **310** is to associate accesses from different threads **301-304** with their appropriate metaphysical address spaces. As an example, a thread identifier (ID) utilized in conjunction with an address referenced by a metadata access operation indexes into the correct metaphysical address space.

To illustrate, assume a metadata access operation, which is associated with thread **302** and references data address **300** for data item **316**, is received. Any method of translation, as described above, may be utilized to translate the data address for data item **316** to a metadata address. However, the translation additionally includes combination with thread ID **302**, which, for example, may be obtained from a control register for thread **302** or an opcode of the received instruction from thread **302**. The combination may include appending thread ID **302** to the address, replacement of bits in the address, or other known method of associating a thread ID with an address. As a result, metaphysical

translation logic **310** is able to select/index into the metaphysical address space associated with data item **316** for processing element **302**.

Extrapolating from the example, by utilizing the thread ID for threads **301-304** as part of the translation into a metaphysical address, then each processing element **301-304** is capable of maintaining independent metadata for data item **316**. Yet, a programmer
5 does not need to individually manage the metaphysical address spaces, because the hardware is capable of keeping them separate through use of the thread ID in a transparent manner to software. Moreover, the metaphysical address spaces are orthogonal – one metadata access from one thread does not access metadata from another thread because
10 each metadata access is associated with a separate set of addresses, which include a reference to a unique thread ID.

Yet, as discussed below, in regard to instructions/operations to access metadata, there may be certain situations where a metadata access from one thread is provided access to another thread's metadata. In other words, in some implementations an access
15 across PEIDs and/or MDIDs (as discussed below) may be advantageous. For example, to determine if hardware has detected conflicts, to check monitor metadata from another thread to determine if an associated data item is monitored by another thread, to clear other thread's metadata, or to determine commit conditions a thread may need to check, modify or clear other thread's metadata associated with data item **316**.

Here, a specific opcode for the operations to access another thread's metadata is recognized, and as a result, metaphysical translation logic **310** performs the translation of address **300** to all metadata addresses for the metadata to be accessed. As a specific illustrative example, where four bits are appended to address **300** with each bit representing one of processing elements **301-304** and a metadata access operation, such as
25 a clear operation, is to clear all metadata for data item **316**, then metaphysical translation logic **310** sets each of the four bits to access all metadata **317**. Here, the lookup logic for memory **315** may be designed where a single access with all four bits set accesses all metadata **317**, or metaphysical translation logic **310** may generate four separate accesses with a different thread ID bit of the four bits set to access all metadata **317**. As an
30 illustrative example, a mask may be applied to an address value to allow one thread to hit metadata of another thread.

Additionally, as illustrated, each processing element **301-304** may be associated with multiple metaphysical address spaces to interleave multiple contexts or software

subsystems within a single thread to multiple metadata address spaces. For example, in some situations, it is potentially advantageous to allow multiple software subsystems within a single processing element to maintain independent metadata sets. Therefore, in one example, orthogonal metadata address spaces may be provided at multiple processing
5 element levels, such as at a core level, hardware thread level, and/or software subsystem level. In the illustration, each processing element **301-304** is associated with two metaphysical address spaces, where each one of the two metaphysical address spaces is to be associated with software subsystems to execute on one of the processing elements.

A software subsystem includes any task or code to be executed on a processing
10 element, which may utilize a separate metaphysical address space. As an illustrative example, four subsystems that may be associated with individual metaphysical address spaces include a transactional runtime subsystem, a garbage collection runtime subsystem, a memory protection subsystem, and a software translation subsystem, which may be executed on a single processing element. Here, each software subsystem may have
15 control of the processing element at different times. As another example, a software subsystem includes individual transactions executed within a single processing element. In fact, it may be desirable for nested transactions executing on the same thread to be associated with separate metaphysical address spaces. To illustrate, a filter test for an access to a data item within an outer transaction may fail, yet it is potentially
20 advantageous to provide a second, distinct filter for an access to the same data item within an inner nested transaction, which may separately succeed to accelerate the access within the inner transaction. Furthermore, when a nested inner transaction aborts, to ensure that the metadata for the outer transaction is maintained, each nested transaction – subsystem – is associated with distinct metadata space, such that a clear of the inner nested
25 transaction's metadata does not affect the outer transaction's metadata. However, a software subsystem is not so limited, as it may be any task or code capable of managing metadata.

In one embodiment, to provide orthogonal metaphysical address spaces at the software subsystem level, the address is combined with the processing element ID (PEID)
30 as discussed above; and in addition, is combined with a metadata ID (MDID), or a context ID. Therefore, separate metadata may be uniquely identified for a subsystem within a processing element. Utilizing an example from above, assume processing elements **301-304** are hardware threads, and that thread **302** is executing an outer transaction and an

inner transaction nested within the outer transaction. For the outer transaction, metadata **317c** is associated with data item **316** through metaphysical translation **310** translating data address **300** of data item **316** to an address plus a thread ID (TID) and a metadata ID (MDID) for the outer transaction, which references metadata **317c**.

5 As a purely illustrative example, metadata **317c** includes four filter values – read filter value, write filter value, undo filter value, and a miscellaneous filter value, a pointer or other reference to a backup location for data item **316**, a monitoring value to indicate if monitors on data item **316** have been lost, a transaction descriptor value, and a version of data item **316**. Similarly, the inner transaction is associated with metadata **317d** for data
10 item **316**, which includes the same metadata fields as those in metadata **317c**. As above, metaphysical translation **310** translates data address **300** for data item **316** to an address combined with the thread ID and the metadata ID for the inner transaction, which references metadata **317d**.

 Here, the only difference between the metadata address, which references metadata
15 **317c**, and the metadata address, which references metadata **317d**, may be the metadata ID for the outer transaction and the inner transaction; yet, this difference in address ensures the address spaces are disjoint/orthogonal – an access to metadata from the inner transaction will not affect metadata from the outer transaction because the MDID for an access from the inner transaction will be different from the outer transaction. As referred
20 to above, this may be advantageous for rolling back nested transactions or holding different metadata values for different level transactions. Specifically, if the inner transaction is aborted, the backup data for data item **316** held in metadata **317d** may be cleared or used to roll-back data item **316** to an entry point before the inner transaction without clearing or affecting the backup data for the outer transaction held in metadata
25 **317c**.

 Note that the metadata ID (MDID) to separate software subsystem metaphysical address spaces may be any size and may come from many sources. As an oversimplified illustrative example, with four processing elements (PEs) **301-304**, a PEID may be from a combination of two bits – 00, 01, 10, 11. Similarly, if four separate metaphysical address
30 spaces are supported, an MDID of two bits – 00, 01, 10, 11, is similarly able to distinguish between four subsystems. To illustrate, a value to represent processing element **302** and subsystem two within PE **302** includes 0101 (first two bits are 01 for PE **302** and the second two bits are 01 for the second subsystem). In this example, metaphysical

translation logic combines this value with data address **300**, or a translation thereof, to reference PE **302** MDID 01, which includes metadata location **317d**.

However, both thread IDs and MDIDs may be more complex. For example, assume threads **301-302** share access to memory **315**, while threads **303-304** are remote processing elements that do not share access to memory **315**. In addition, assume that threads **301-302** each support two software subsystems for a total of four orthogonal address spaces for threads **301-302** – PE 301 MD0, PE 301 MD1, PE 302 MD0, and PE 302 MD1 address spaces. In this case, a value for the combined thread ID and MDID utilized to obtain the metadata address may come from an opcode, a control register, or a combination thereof. To illustrate, an opcode provides one bit for context/MDID, a control register provides one bit for a processing element ID (PEID) – assuming only two processing elements, and a metadata control register, such as MDCR **320**, provides four bits to identify a specific software subsystem/context for greater granularity. Therefore, when a metadata access operation referencing address **300** for data item **316** is received from the second thread -- PE **302**, then the one bit from the opcode – the first bit including a 1 to indicate a second context, and a second bit from a control register for processing element **302**—the second bit including a 1 to indicate processing element **302**, is combined with a MDID from metadata control register (MDCR) **320** associated with the second thread; the MDCR to have been previously updated by the current subsystem's MDID, which is controlling the second thread – 0010 – to identify the proper subsystem associated with the received operation. Metaphysical translation logic takes the combined value, such as 110010, and further combines it with referenced data address **300**, or a translation thereof, to obtain a metadata address. However, the 110010 part of the metadata address is unique to the subsystem that the access operation originated from, so it will only hit or modify metadata address **317d** in memory **315** without hitting or affecting metadata addresses **317a, b, c, e, f, g, h** – the orthogonal metaphysical address spaces for other subsystems both within the second thread and other threads.

As a specific illustrative example a discussion of a specific form of MDCR is included. In some embodiments an ISA may be extended with a per-thread metadata identifier register (MDID register), which sources an MDID to MDID-sensitive metadata load/store/test/set instructions. In some embodiments it is convenient to have a plurality of such registers. For example, MDCR: Metadata Control Register is a 32-bit read-write register contains the current metadata context ID (MDID). It may be updated by a CR

MOV. Exemplary bit field definitions are as follows:

Bit	Field	Content
3:0	MDI D0	Metadata context ID 0
7:14	MDI D1	Metadata context ID 1
1:28	MDI D_size	Number of MDID bits provided by the processor

Table A: Exemplary Embodiment of bits for MDCR

MDID 0 and MDID1 are the metadata IDs concurrently accessible to the instruction set. The number of bits actually used out of these fields is MDID_size, which in one embodiment, is read only at any permission level, as it is specified by processor design. However, in other embodiments different level privilege levels may be able to modify the size. There may be no hardware checks that ensure the MDID fits within the size bit allotment. In one embodiment, MDID0 and MDID1 are capable of being written and read at any permission level. It may also be possible to use special MDID values to designate special metadata spaces which always read as zero or one. This might be used by software to force all metadata tests in a block to be true or false in a similar fashion to the discussion of a register to force a metadata value in reference to **Figure 6 and 7**.

However, in another example, as mentioned above, metaphysical translation logic **310** in conjunction with decoders (not illustrated) are capable of recognizing metadata access operations from thread **302**, which are intended to access metadata from thread **301's** metadata address space, and allow access for those specific instructions/operations to read or modify thread **301's** metadata.

Compression of Metadata to Data

Above, a one-to-one mapping of data to metadata – uncompressed metadata – has been discussed; however, in some circumstances it is more efficient to utilize a smaller amount of metadata in comparison to data -- compression of metadata where the size of metadata is smaller than data. Note that the metaphysical address translation logic **210** and **310** from **Figures 2-3** may take compression into account when performing translation and modification of an address to reference compressed metadata, accordingly. Referring to **Figure 4**, an embodiment of modifying an address to achieve compression of metadata is illustrated; specifically, an embodiment of a compression ratio of 8 for data to metadata is depicted. Control logic, such as metaphysical address translation logic **210**

and **310** from **Figures 2-3**, is to receive data address **400** referenced by a metadata access operation. As an example, compression includes shifting or removing $\log_2(N)$ number of bits within or from address **400**, where N is the compression ratio of data to metadata. In the illustrated example, for a compression ratio of 8, three bits are shifted down and
5 removed for metadata address **405**. Essentially, address **400** that includes 64 bits to references a specific data byte in memory is truncated by three bits to form the metadata byte address **405** used to reference metadata in memory on a byte granularity; out of which a bit of metadata is selected using the three bits previously removed from the address to form the metadata byte address.

10 The bits shifted/removed, in one embodiment, are replaced by other bits. As illustrated, the high order bits, after address **400** is shifted, are replaced with zeros. However, the removed/shifted bits may be replaced with other data or information, such as a processing element ID, context identifier (ID), and/or a metadata ID (MDID) associated with the metadata access operation. Although the lowest number bits are removed in this
15 example, any position of bits may be removed and replaced based on any number of factors, such as cache organization, cache circuit timing, locality of metadata to data, and minimizing conflicts between data and metadata

For example, a data address may not be shifted by $\log_2(N)$, but rather address bits 0:2 are zeroed. As a result, bits of the physical address and virtual address that are the
20 same are not shifted as in the example above, which allows for pre-selection of a set and a bank with unmodified bits, such as bits 11:3.

Note that the discussion with regard to translation may be combined with compression. In other words, a compression ratio may be an input into metaphysical address translation logic **210** and **310** from **Figures 2-3** and the translation logic utilizes
25 the compression ratio in conjunction with a PEID, CID, MDID, metaphysical value, or other information to translate a data address into a metadata address. The metadata address is then utilized to access a memory holding the metadata. As discussed above, since metadata is a local construct – lossy, misses to the memory based on the metadata address may be serviced quickly and efficiently – allocation of a memory location without
30 generating an external miss service request and without waiting for the external request to be serviced. Here, an entry is allocated in a normal fashion for the metadata. For example, an entry, such as entry **217** from **Figure 2**, is selected, allocated, and initialized to the metadata default value based on metadata address **405** and a cache replacement

algorithm, such as a Least Recently Used (LRU) algorithm. As a result, metadata potentially competes with regular data for space, but remains compressed and disjoint from other software subsystems/processing elements.

Note that a compression ratio of eight is purely illustrative and any compression ratio may be utilized. As another example, a compression ratio of 512:1 is used – a bit of metadata represents 64 bytes of data. Similar to above, a data address is translated/modified to form metadata address through shifting the data address down by $\log_2(512)$ bits – 9 bits. Here, bits 6:8 are still utilized to select a bit, instead of bits 0:2, effectively creating the compression through selection at a granularity of 512 bits. As the data address has been shifted by 9 bits, the high order portion of the data address has 9 open bit locations to hold information. In one embodiment, the 9 bits are to hold identifiers, such as context ID, thread ID, and/or MDID. In addition, metaphysical space values may also be held in these bits or the address may be extended by the metaphysical value.

In one embodiment, multiple concurrent compression ratios are supported by hardware. Here, a representation of a compression ratio is held as part of a metaphysical value combined with a data address to obtain a metadata address. As a result, during a search of a memory with the data address, the compression ratio is taken into account and does not match addresses of different compression ratios. Furthermore, software may be able to rely on hardware to not forward store information to loads of a different compression ratio.

In one embodiment, hardware is implemented utilizing a single compression ratio, but includes other hardware support to present multiple compression ratios to software. As an example, assume cache hardware is implemented utilizing an 8:1 compression ratio, as illustrated in **Figure 4**. Yet, a metadata access operation to access metadata at different granularities, is decoded to include a micro-operation to read a default amount of metadata and a test micro-operation to test an appropriate part of the metadata read. As an example, the default amount of metadata read is 32-bits. However, a test operation for a different granularity/compression of 8:1 tests correct bits of the 32 bits of metadata read, which may be based on a certain number of bits of an address, such as a number of LSBs of a metadata address, and/or a context ID.

As an illustration, in a scheme supporting metadata for unaligned data for a bit of metadata per byte of data, a single bit is selected from the least significant eight bits of the

32 read bits of metadata based on the three LSBs of a metadata address. For a word of data, two consecutive metadata bits are selected from the least significant 16 bits of the 32 bits of read metadata based on the three LSBs of the address, and continuing all the way to 16 bits for a 128 bit metadata size.

5 Metadata Access Instruction/Operations

Turning to **Figure 5** a flow diagram for a method of accessing metadata associated with data is illustrated. Although the flows of **Figure 5** are illustrated in a substantially serial fashion, the flows may be performed at least partially in parallel, as well as potentially in a different order.

10 In flow **505** a metadata operation referencing a data address for a given data item is encountered. In the discussion above it was mentioned that metadata instructions/operations may be supported in hardware to read, modify, and/or clear metadata. In other words, instructions may be supported in a processor's Instruction Set Architecture (ISA), such that decoders of the processor recognize operation codes
15 (opcodes) of instructions to access data and logic to perform the accesses, accordingly. Note that use of instruction may also refer to an operation. Some processors utilize the idea of a macro-instruction, which is capable of being decoded into a plurality of micro-operations to perform individual tasks, such as a test and set metadata macro-instruction, which is decoded into a metadata test operation/micro-operation to test the metadata and if
20 the correct Boolean value is obtained as a result of the test operation, then a set operation updates the metadata to a specific value.

However, metadata access operations are not limited to explicit software instructions to access metadata, but rather may also include implicit micro-operations decoded as part of a larger more complex instruction that includes an access to a data item
25 associated with metadata. Here, the data access instruction may be decoded into a plurality of operations, such as an access to the data item and an implicit update of the associated metadata.

As previously discussed, in one embodiment, the physical mapping of metadata to data in hardware is not directly visible to software. As a result, metadata access
30 operations, in this example, reference data addresses and relies on the hardware to perform the correct translations, i.e. mapping, to access the metadata appropriately. Yet, metadata access operations may individually reference separate metaphysical address spaces depending on which thread, context, and/or software subsystem they originate from.

Therefore, a memory may hold metadata for data items in a transparent fashion with regard to the software. When the hardware detects an access operation to metadata, either through explicit operation code (op code of an instruction) or decoding of an instruction into a metadata access micro-operation(s), the hardware performs the requisite translation of the data address referenced by the access operation to access the metadata accordingly.

As this example illustrates, a program may include separate operations, such as a data access operation or a metadata access operation, that reference the same address of a data item, such as data items **216** and **316** from **Figure 2-3**, and the hardware may map those accesses to different address spaces, such as a physical address space and a metaphysical address space. In some embodiments the ISA may be extended with instructions to load/store/test/set metadata for a given virtual address, MDID, compression ratio, and operand width. Any of these parameters may be explicit instruction operands, may be encoded in the opcode, or may be obtained from a separate control register. Instructions may combine the metadata load/store operation with other operations, for example, loading some data, testing some bits of it, and setting a condition code for a subsequent conditional jump. Instructions may also flush all metadata, or just metadata for a particular MDID. Below are listed a number of illustrative metadata access operations. Note that some of the exemplary instructions are in reference to specific 64X compression ratio instructions, but similar instructions may be utilized for different compression ratios, as well as uncompressed metadata, even though they are not specifically disclosed.

Metadata Bit Test and Set (MDLT)

The metadata load and test instruction (MDLT) has 2 arguments: the data address to which the metadata is associated as a source operand and a register (destination operand) into which the byte, word, dword, qword or other size of metadata containing the bit is written. The value of the tested metadata bit is written into the register. The programmer should not assume any knowledge about the data stored in the destination register of the MDLT instruction, and should not manipulate this register. This register is to be used solely as a source operand to a metadata store and set instruction (MDSS) to the same address. In one embodiment, the MDLT instruction will combine the test and set operations, but will squash the set operation if the test succeeds.

Metadata Store and Set (MSS)

The metadata store and set instruction (MDSS) has 2 arguments: The data address

to which the metadata is associated and a register (source operand) from which the byte, word, dword, qword or other size of metadata containing the bit is to be stored to memory. The MDSS instruction will set the correct bit in the value from its source operand.

Metadata Store and Reset Instruction (MDSR)

5 The MDSR instruction has 2 source arguments: The data address to which the metadata is associated as a source operand and a register (source operand) from which the byte, word, dword, qword or other size of metadata containing the bit is to be reset. The MDSR instruction will reset the correct bit in the value from its source operand.

10 A metadata address is determined from the referenced data address. Examples of determining a metadata address are included in the metaphysical address translation and multiple metaphysical address spaces sections above. However, note that the translation may incorporate, i.e. be based on, a compression ratio of data to metadata so as to separately store metadata for each compression ratio.

Test Metadata (CMDT)

Opcode	Instr uction	Description
0F 3A 81/3 ib	CM DT0 mem	Sets the ZF flag if the value of the metadata corresponding to the data address mem is zero, uses MDID0 from MDCR.
	CM DT1 mem	Sets the ZF flag if the value of the metadata corresponding to the data address mem is zero, uses MDID1 from MDCR.

15 **Table B: Illustrative Embodiment of Test Metadata Operation**

The CMDT instruction is to convert the memory data address to a memory metadata address with a compressed mapping function that is implementation dependent and test whether a metadata bit corresponding to the memory metadata address is set. As an example, the compression ratio CR is of 1 bit for 8 bytes. The metadata address computation incorporates one of the context IDs from the MDCR register to provide a unique set of MD for each individual context ID, addressing
 20 MDBLK[CR][MDCR.MDID[MDID number]].META. The instruction aligns the address 'mem' to the data size specified, thus enforcing alignment. The instruction tests whether Metadata is set.

25 Included below is exemplary pseudo code relating to CDMT (The ZF flag is set to represent zero metadata value. All the other flags are cleared):

```
if (TxCR.FORCE == 0) {
```

```

cr := 64    // only 1 bit for 8 bytes supported
mdid := MDCR[MDID_0] // bits 0:14
ZF := ! GetMetaDataBit(addr, cr, mdid) }

```

```

5      64MDT1 addr
      FLAGS := 0

      if (TxCR.FORCE == 0) {
cr := 64    // only 1 bit per 8 bytes supported
10     mdid := MDCR[MDID_1] // bits 15:27
      ZF := ! GetMetaDataBit(addr, cr, mdid) }

```

Compressed Metadata Store (CMDS)

Opcode	Instr uction	Description
0F 3A 81/2 ib	CM DS0 mem, imm8	Stores into the metadata corresponding to the data address mem, conversion of metadata address uses MDID0 from MDCR
	CM DS1 mem, imm8	Stores into the metadata corresponding to the data address mem, conversion of metadata address uses MDID1 from MDCR

Table C: Illustrative Embodiment of Store Metadata Operation

15 The CMDS instruction converts the memory data address to a memory metadata address with a compressed mapping function that is implementation dependent. The compression ratio is 1 bit for 8 bytes of data. The encoding of the imm8 value is as follows: 0 → MD_Value; Value to be stored into MD and 7:1 → Reserved; Not Used

Included below is exemplary pseudo code associated with CMDS:

20 Writes MDBLK[64][MDCR.MDID[MDID number]](addr).META = MD_value
Operation

```

C64MDS0 addr
cr := 64    // only 1 bit per 8 bytes supported
mdid := MDCR[MDID_0] // bits 0:14
25 StoreMetaDataBit(addr, cr, mdid, imm8[0])

```

```

C64MDS1 addr

```

cr := 64 // only 1 bit per 8 bytes supported

mdid := MDCR[MDID_1] // bits 15:27

StoreMetadataBit(addr, cr, mdid, imm8[0])

- 5 Implementation note: the instruction will perform read-set bit-write operation on metadata.

Flags affected: No

- 10 Protected Mode and Compatibility Mode Exceptions

#UD if CR4. OSTM [bit 15] = 0

No #PF

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.

15

Compressed Metadata Clear (CMDCLR)

Opcode	Instru ction	Description
0F 3A 81/4 ib	CMDCLR0 LR0 mem	Clears ranges of metadata corresponding to the data address mem, conversion of metadata address uses MDID0 from MDCR
	CMDCLR1 LR1 mem	Clears ranges of metadata corresponding to the data address mem, conversion of metadata address uses MDID1 from MDCR

Table D: Illustrative Embodiment of Clear Metadata Operation

The CMDCLR instruction resets all MDBLK[CR][MDCR.MDID[MDID number]].META that correspond to any data in the range spanning MBLK(mem).

- 20 Exemplary pseudo code related to CMDCLR is included below:

Operation

C64MDCLR0

cr := 64 // only 1 bit per 8 bytes supported

- 25 mblk := floor (addr, MBLK_SIZE)

```

        mdblklStart := mblk
        mdblklEnd := floor(mblk + MBLK_SIZE - 1, MDBLK_SIZE)

        mdid := MDCR[MDID_0] // bits 0:14
5
        for all mdblkl in Mblk DO StoreMetadataBit(addr, cr, mdid, 0)

        C64MDCLR1
        cr := 64 // only 1 bit for 8 bytes supported
10
        mblk := floor(addr, MBLK_SIZE)
        mdblklStart := mblk
        mdblklEnd := floor(mblk + MBLK_SIZE - 1, MDBLK_SIZE)

        mdid := MDCR[MDID_1] // MDCR[27:15]
15
        for all mdblkl in mblk DO StoreMetadataBit(addr, cr, mdid, 0)

        Implementation note: Will be 1 byte clear for 64:1 CR supported in 1st
20 implementation.
        Flags affected: No

        Protected Mode and Compatibility Mode Exceptions
        #UD if CR4. OSTM [bit 15] = 0
25
        64-Bit Mode Exceptions #GP(0) If the memory address is in a non-canonical
        form.

        Next, in flow 510, a metadata address is determined from the data address
        referenced in the metadata access operation based on a compression ratio, processing
30 element ID, context ID, MDID, metaphysical value, operand size, and/or other
        metaphysical address space translation related value. Any of the methods described
        above, such as combination of ID values with no translation of the data address, normal
        translation of the data address, or separate metaphysical address translation of the data

```

address, may be utilized to obtain the appropriate metadata address.

Furthermore, as stated above, in some instances a version of the test, set, clear, or other instructions are provided to allow one thread or metadata context to test, set, or clear other thread's or metadata context's metadata. As a result, the translation to a metadata
 5 address may include modification of the address, such as application of a mask, to allow the access from one thread or context ID to access another thread or context ID.

In flow **515**, the metadata referenced by the metadata address is accessed. For the normal case, the disjoint location for the metadata associated with the local requesting thread or context ID is accessed and the appropriate operations, such as test, set, and clear,
 10 are performed. However, in the second case, described above, metadata for other threads or context IDs may be accessed in this flow as well.

Abstractions

An embodiment of abstractions for software is included herein. A given CR is a power of two that indicates how many bits of data map to one bit of metadata. It is
 15 implementation defined which CRs values, if any, may be used. CR>1 denotes Compressed Metadata. CR=1 denotes Uncompressed Metadata.

MDBLK[CR][*]s are $\text{ceil}(\text{CR}/8)$ bytes in size and are naturally aligned. MDBLKs are associated with physical data, not their linear virtual addresses. All valid physical addresses A with the same value $\text{floor}(A/\text{MDBLK}[\text{CR}][*]_{\text{SIZE}})$ designate the same sets
 20 of MDBLKs.

For a given CR, there can be any number of distinct MDIDs each designating a unique instance of metadata. The metadata for a given CR and MDID is distinct from the metadata for any other CR or MDID. For example, for Thd #0, assuming addr is QWORD aligned, then the Metadata Block referred to by $\text{MDBLK}[\text{CR}=64][\text{MDID}=3](\text{addr})$ is the
 25 same as $\text{MDBLK}[\text{CR}=64][\text{MDID}=3](\text{addr}+7)$, but it is certainly distinct from $\text{MDBLK}[\text{CR}=64][\text{MDID}=4](\text{addr})$ and from $\text{MDBLK}[\text{CR}=512][\text{MDID}=3](\text{addr})$.

A given implementation may support multiple concurrent contexts, where the number of contexts will depend on the CR and certain configuration information related to the specific system of which the processor is a part. For Uncompressed Metadata, there is
 30 a QWORD of metadata for each QWORD of physical data.

Metadata is interpreted by software only. Software may set, reset, or test META for a specific $\text{MDBLK}[\text{CR}][\text{MDID}]$, or reset META for all the Thd's $\text{MDBLK}[*][*]$ s, or reset META for all the Thd's $\text{MDBLKs}[\text{CR}][\text{MDID}]$ that may intersect a given

MBLK(addr).

Metadata Loss. Any META property of the Thd may spontaneously reset to 0, generating a Metadata Loss Event.

Forced Metadata Value

5 Referring to **Figure 6**, an embodiment of providing hardware support for a forced metadata value is illustrated. STMs usually ensure consistency between memory access operations utilizing access barriers. For example, before a memory access to a data item, a metadata location or lock location associated with the data item is checked to determine if the data item is available. Other potential barrier operations include obtaining a lock, such
10 as a read lock, write lock, or other lock, on the data item in the metadata or lock location, logging/storing a version for the data item in a read or write set for a transaction, determining if a read set for a transaction to that point is still valid, buffering or backing up a value of the data item, setting monitors, updating a filter value, as well as any other transactional operations.

15 However, often within a transaction, subsequent accesses to the same data item incurs the overhead of executing an associated transactional barrier each time the access to the data item is encountered. To illustrate three writes to address A are performed within a transaction, which in this scenario, results in execution of a write barrier three separate times to acquire a write lock for address A. Yet, the lock for address A has already been
20 acquired through execution of a write barrier at the first transactional write, and the subsequent two executions of the write barriers before the last two transactional writes are superfluous – the lock on address A does not need to be re-acquired.

Therefore, in one embodiment, hardware holds a filter value to accelerate execution associated with these barriers. The filter value may be included in a cache as an
25 annotation bit, such as the read and write monitors, or held in a metadata location within a metaphysical address space, as previously described. Utilizing the example from above, when the first write barrier is encountered, it updates a write filter value from an un-accessed value to an accessed value to indicate a write barrier for address A has been encountered already within the transaction. Therefore, upon the subsequent two
30 transactional write operations within the transaction, before vectoring to the write barrier, the write filter value for address A is checked. Here, the filter value includes an accessed value, which indicates that the write barrier does not need to be executed – the write barrier was already executed within the transaction. As a result, execution is not vectored

to the write barrier for the last two write operations. In other words, the filter value accelerates transactional execution – elides or does not include execution of the write barrier for the last two accesses in comparison to the previous example without utilizing a filter.

5 Note that read filters for loads/reads, undo filters for undo operations, and miscellaneous filters for generic filter operations may be utilized in the same manner as the write filter above was utilized for write/store operations.

 Concepts also associated with transactional barriers are strong and weak atomicity, which deal with the isolation of transactional operations from non-transactional
10 operations. Here, just as a transactional writes to a memory location transactionally loaded is a potential conflict, a transactional write to a memory location non-transactionally loaded is a potential conflict that results in invalid data utilized by the non-transactional load operation. In weak atomicity systems, there are no or minimal barriers inserted at non-transactional operations, so the weak atomicity systems run the risk of
15 invalid execution. In contrast, in strong atomicity systems, transactional barriers are also inserted at non-transactional operations; this provides protection and isolation between transactional and non-transactional operations, but at a cost – the expense of executing a transactional barrier at every non-transactional operation.

 Therefore, in one embodiment, the filters described above may be leveraged in
20 combination with strong atomicity barriers at non-transactional operations to support different modes of strong and weak atomicity operation. To illustrate, a simplified exemplary embodiment is illustrated in **Figure 6**. Here, metadata **610** is held in hardware for data **605**, as discussed above. Metadata access **600** is received to access metadata **610**. In one embodiment, metadata access includes a test metadata operation to test a filter, such
25 as read filter, write filter, undo filter, or miscellaneous filter.

 A test metadata operation to test a filter may originate from a transactional or non-transactional access operation. In one embodiment, a compiler, when compiling application code, inserts the test filter operation inline in the application code as a condition to executing a call to a transactional barrier at transactional and non-
30 transactional accesses. Therefore, within a transaction, the filter operation is executed before a call to a barrier, and if it returns successful, then the call to the transactional barrier is not executed providing the acceleration discussed above.

 Yet, with non-transactional operations, in one embodiment, the hardware is

capable of operating in a weak atomicity mode, where transactional barriers at non-transactional operations are not executed, and a strong atomicity mode where transactional barriers are executed.

The mode of operation, or control **625**, may be set in metadata control register (MDCR) **615**, which may be combined with the version of MDCR described above to hold MDIDs or may be a separate control register. In another embodiment, control **625** for mode of operation may be held in a general transactional control register or status register. Here, a first mode of execution includes a strong atomicity mode where transactional barriers are to be executed at non-transactional operations. In this case, control **625** represents a first value, such as a 00, to indicate a strong atomicity and non-transactional mode of operation. In response logic **620**, which is listed as an exemplary multiplexer, selects the metadata value from hardware maintained metadata **610** associated with data address A to be provided to destination register **650** for metadata access **600**. Essentially, in a strong atomicity mode barriers are accelerated based on the actual hardware held metadata. Alternatively, during a second mode of execution, such as a weak atomicity and non-transactional mode, as indicated by control **625** representing a second value, such as 01, a fixed or forced value from MDCR is provided to destination register **650** in response to metadata access **600** instead of the hardware maintained metadata **610**.

Essentially, in a weak atomicity mode, a forced value is provided to destination register **650** in response to test filter operation **600** to ensure the test of the filter value always succeeds and the call to the transactional barrier is not executed before the non-transactional memory access. Note that this description assumes that the test filter operation, is returning a Boolean value as to indicate if the filter test succeeds (barrier is not to be executed) or fails (barrier is to be executed). As a result, the same filter software construct for accelerating transactions by eliding barriers based on the filter value is leveraged to provide one mode of operation where all barriers at non-transactional operations are elided – weak atomicity mode, and a second mode of operation where barriers at non-transactional operations are executed or accelerated based on hardware maintained metadata – strong atomicity. In another embodiment, different forced values may be provided for each mode. Here, in a strong atomicity mode, the forced value would ensure the test filter operation fails so the barrier is always executed, while in the weak atomicity mode, the forced value would ensure the test filter operation succeeds so the barrier is not executed.

Although providing a forced or fixed value from a control register, such as MDCR 615, based on control information, such as control 625, has been described in relation to providing a fixed/forced value or a metadata value based on mode operation, providing a forced or fixed value may be utilized for any generic metadata usage, such as allowing a data-invariant behavior to be utilized for debugging and generic monitoring of memory accesses capable of being enabled on-demand.

Turning to **Figure 7**, an embodiment of a flow diagram for accelerating non-transactional operations while maintaining atomicity in a transactional environment is depicted. In flow 705, a metadata (MD) access operation referencing a data address is encountered. As one specific illustrative example, the MD access operation includes a test operation previously inserted by a compiler in-line with application code to elide a transactional barrier at a non-transactional memory access if the test returns one value (successful) and to execute the barrier if the test returns a second value (failure). However, a test MD operation is not so limited, as it may include any test operation for returning a Boolean success or failure value.

In flow, 710 a mode of operation is determined. Here, examples of a mode of operation may be transactional or non-transactional in combination with strong atomicity or weak atomicity. Therefore, one, or two separate registers, may hold a first bit to indicate a transactional or non-transactional mode of operation and a second bit for strong or weak atomicity mode of operation.

If the mode of operation is transactional or non-transactional and strong atomicity, then the hardware maintained metadata value is provided to the metadata access operation – the hardware maintained value is placed in a destination register specified by the MD access operation. In contrast, if the mode of operation is non-transactional and weak atomicity, then the forced MDCR fixed value is provided to the MD access operation instead of the hardware maintained MD value. As a result, during the strong atomicity mode, barriers are accelerated or not based on a hardware maintained MD value, while in the weak atomicity mode, barriers are accelerated based on the forced MDCR value.

Efficient Transition to a Buffered and Monitored State

Turning next to **Figure 8**, an embodiment of a flow diagram for a method of efficiently transitioning a block of data to a buffered and monitored state before commit of a transaction is illustrated. As described above, blocks of memory, such as a cache line holding a data item or metadata may be buffered and/or monitored. For example,

coherency bits for a cache line include a representation of a buffered state and attribute bits for a cache line indicate if the cache line is unmonitored, read monitored, or write monitored.

5 In some embodiments, a cache line is buffered, but unmonitored, which means the data held in the cache line is lossy and that conflicts to the cache line are not detected, since there is no monitoring applied. For example, data that is local to a transaction and is not to be committed, such as metadata, may be held in a buffered and unmonitored state.

10 When conflicts between buffered data and writes to the same address are to be detected, read monitoring is applied to the data. The cache line is then moved to a buffered and read monitored state; however, to get to that state, a read request is sent to external processing elements forcing all other copies to transition to a shared state. These external read requests may result in a conflict with another processing element maintaining a write monitor on the same block/cache line.

15 Similarly, when conflicts between the buffered data and reads to the same memory blocks are to be detected, write monitoring is applied to the cache line. The line is then moved to a buffered and write monitored state, which is achieved by a sending read for ownership request to other processing element forcing all other copies to transition to an invalid state. Similarly, a conflict is detected with any processing element maintaining either a read or write monitor on the same memory block.

20 Yet, to minimize transactional conflicts, a memory block that the transaction needs to update but not eventually commit may be maintained in the buffered but unmonitored state, as described above. However, if a block held in the buffered but unmonitored state is determined to be committed, then in one embodiment an efficient path from the buffered and unmonitored state to a committable state is provided as illustrated in **Figure 8**.

25 As an example, a buffered update to a memory block – a cache line to hold the block – is received in flow **805**. Either before the buffered update, or simultaneously therewith, read monitoring is applied to the block. For example, a read attribute for the cache line is set to a read monitor value to indicate the block is read monitored. However, to apply read monitoring, a read request is first sent out to other processing elements in flow **815**. In response to receiving the read request, the other processing elements either
30 detect a conflict due to maintaining the line in a write monitoring state already, or transition their copies to a shared state in flow **820**. In flow **825**, if there are no conflicts, then the cache line is transitioned to a buffered and read monitored state – cache line

coherency bits are updated to a buffered coherency state and the read monitor attribute is set.

In flow **830**, conflicting writes are detected to the cache line based on the read monitoring. In one embodiment, the read attributes are coupled to snoop logic, such that an external read for ownership request to the cache line will detect a conflict with the read monitor being set on the cache line.

Later, when the block is to be committed as part of a state of a transaction in flow **835**, then write monitoring is applied in flow **840**. Here, a read for ownership request is sent to the other processing elements in flow **845**, which either detects a conflict in response to holding the cache line in a read or write monitored state, or transitions their copy to an invalid state in flow **850**. As a result, the detection of the conflicts at the read for ownership request allows for any conflicts to be detected at that point, which essentially places the line in a committable state.

Consequently, transitioning the buffered and unmonitored block to a committable state in two stages – flow **810** and flow **840** – is potentially advantageous. Deferring the acquisition of ownership via the staged acquisition of read and write monitors allows multiple concurrent transactions to update the same block, while reducing the conflicts between these transactions. If a transaction does not get to the commit stage for any reason, updating the block in a buffered and read monitored way will not cause another transaction that will get to the commit stage to needlessly abort. In addition, deferring acquiring sole ownership of the block until the commit stage is therefore a way to obtain higher concurrency among threads without sacrificing validity of data.

Table E below illustrates an embodiment of conflicting states between two processing elements: P0 and P1. For example, a line held by P1 in a buffered read monitored state, as indicated by the R-B column, and any state of P0 with the cache line maintained with a write monitor, as indicated by the –W–, RW–, WB, RWB, is conflicting, as represented by the x in the intersecting cells.

P0 State	P1 state							
	--	--	W-	W-	-B	-B	B	B
--								
R			x	X			x	X

W-	-			x	X		x	x	X
W-	R			x	X		x	x	X
-B	-			x	X			x	X
-B	R			x	X			x	X
B	W			x	X		x	x	X
WB	R			x	X		x	x	X

Table E: An embodiment of conflicting states between two processing elements

Additionally, Table F below illustrates a loss of an associated property in processing element P1 in response to the operation listed under P0. For example, if P1 holds a line in a buffered read monitored state, as indicated by the R-B column, and either a store or set write monitor operation occurs on P0, then P1 loses both read monitoring and buffering of the line as indicated by the **x-x** in the intersection of the store/set WM rows and the R-B column.

P0 Instruction	P1 state							
	--	--	W-	W-	-B	-B	B	B
Load		--	x-	x-	-B	-B	xB	RxB
Set RM		--	x-	x-	-B	-B	xB	RxB
Buffered Str		--	x-	x-	-B	-B	xB	RxB
Store		--	x-	-	-x	-x	xx	xxx
Set WM		--	x-	-	-x	-x	xx	xxx

Table E: An embodiment of loss of attributes as result of an operation

10 Branch Instruction (JLOSS) for conflict or loss of transactional data

Turning to **Figure 9**, an embodiment of hardware to support a loss instruction to jump to a destination label based upon a status value in a transaction status register is

illustrated. In one embodiment, hardware provides an accelerated way to check a transaction's consistency. As examples, hardware may support consistency checking by providing mechanisms that track loss of monitored or buffered data from the cache – eviction of buffered or monitored lines, or track potential conflicting accesses to such data – monitors to detect conflicting snoops, such as a read request for ownership to a monitored line.

In addition, in one embodiment, hardware provides architectural interfaces to allow software to access these mechanisms based on the status of monitored or buffered data. Two such interfaces include the following: (1) Instructions to read or write a status register that allow the software to poll the register explicitly during execution; (2) an interface that allows software to setup a handler that is invoked whenever the status register indicates a potential loss of consistency.

In another embodiment, hardware supports a new instruction called JLOSS that performs a conditional branch based on the status of HW monitored or buffered data. The JLOSS instruction branches to a label if the hardware detects potential loss of any monitored or buffered data from the cache, or it detects potential conflicts to any such data. A label includes any destination, such as an address of a handler or other code to be executed as a result of a loss of data or detection of a conflict.

As an illustrative embodiment, **Figure 9** depicts decoders **910**, which recognize JLOSS as part of a processor ISA and decodes the instruction to allow logic of the processor to perform the conditional branch based on the status of a transaction. As an example, the status of a transaction is held in transaction status register **915**. Transaction status register may represent the status of transactions, such as when hardware detects a conflict or a loss of data—herein referred to as a loss event. To illustrate, a conflict flag in TSR **915** is set upon a monitor indicating an address is monitored in combination with a snoop to the monitored address, the conflict flag in TSR **912** indicating a conflict was detected. Similarly, a loss flag is set upon a loss of data, such as an eviction of a line including transactional data or metadata.

Therefore here, JLOSS when decoded and executed, tests the status register flags, and if there is a loss event—loss and/or conflict, then logic **925** provides the label referenced by JLOSS to execution resources **930** as a jump destination address. As a result, with a single instruction, software is able to discern the status of a transaction, and based on that status is capable of vectoring execution to a label specified by the single

instruction. Because JLOSS checks consistency, reporting of false conflicts is acceptable – JLOSS may conservatively report that a conflict has occurred.

In one embodiment, software, such as a compiler, inserts JLOSS instructions into the program code to poll for consistency. Although, JLOSS may be utilized inline with
 5 main application code, often JLOSS instructions are utilized within read and write barriers to determine consistency on demand, which are often provided within libraries; therefore, execution of program code may include a compiler to insert JLOSS in code, or execution of JLOSS from the program code, any other form of inserting or executing an instruction. It's expected that polling by JLOSS is much faster than an explicit read of the status
 10 register, because the JLOSS instruction does not require additional registers – there is no need for a destination register to receive the status information for an explicit read. Several embodiments of this instruction exist in which the conditions to check for consistency are provided either explicitly in the instruction or implicitly in a separate control register.

As an example, transaction status register **915**, or other storage element, holds
 15 specific conflict and loss status information, such as if a read monitored location has been written by another agent – read conflict, a write monitored location has been read or written by another agent – write conflict, a loss of physical transactional data, or a loss of metadata. Therefore, different versions of the JLOSS instruction may be utilized. For example, a JLOSS.rm <label> instruction will branch to its label if any read monitored
 20 location may have been written by another agent. A hardware-accelerated STM (HASTM) is able to use this JLOSS.rm instruction to accelerate consistency checking – quickly check for conflicting updates to a read set by using JLOSS.rm wherever it is to ensure read-set consistency, such as after each transactional load in a native code TM system. In this case, a read set may be verified utilizing JLOSS in a read barrier, so the
 25 JLOSS instruction is inserted in the barrier within a library or after the load operation inline with main application code. Similar to the JLOSS.rm instruction for detecting writes to read monitored locations, a JLOSS.wm instruction may be utilized to detect any reads or writes to a write monitored locations. As yet another example, in a processor that is able to buffer locations, a JLOSS.buf instruction may be used to determine if buffered
 30 data has been lost and jump to a specified label as a result.

The following Pseudo Code, labeled pseudo code A, shows a native code STM read barrier that provides a consistent read set and uses JLOSS. The setrm(void* address) function sets the read monitor on the given address and the jloss_rm() function is an

intrinsic function for the JLOSS instruction that returns true if any conflicting accesses to read monitored locations may have occurred. This pseudo-code monitors the loaded data, but it's also possible to monitor the transaction records (ownership records) instead. It's possible to use an instruction that combines setting of the read monitor with loading of the data – e.g. a movxm instruction that both loads and monitors the data. It's also possible to use this in a read barrier that performs filtering in addition to monitoring, as well as to use this in an STM system that only uses hardware monitoring for read-set validation – a STM system that performs no software read logging and no SW validation.

Pseudo Code A: An in-place update STM, optimistic read, native code read barrier

```

Type tmRd<Type>(TxnDesc* txnDesc, Type* addr) {
    setrm(addr); /* set the read monitor on loaded address */
    TxnRec* txnRecPtr = getTxnRecPtr(addr);
    TxnRec txnRec = *txnRecPtr;
    val = *addr;
    if (txnRec != txnDesc) {
        while (!validateAndLogUtm(txnDesc, txnRecPtr, txnRec)) {
            /* retry */
            txnRec = *txnRecPtr;
            val = *addr;
        }
    }
    return val;
}

bool validateAndLog(TxnDesc* txnDesc, TxnRec* txnRecPtr, TxnRec txnRec) {
    if (isWriteLocked(txnRec) ||
        !checkReadConsistency(txnDesc, txnRecPtr, txnRec)) {
        handleContention(...);
        return false;
    }
    logRead(txnDesc, txnRecPtr);
}

```

```

    return true;
}

bool checkReadConsistency(TxnDesc* txnDesc, TxnRec* txnRecPtr, TxnRec txnRec)
{
    if (txnRec > txnDesc->timestamp) {
        TxnRec timestamp = GlobalTimestamp;
jloss_rm Lost:
        txnDesc->timestamp = timestamp;
        return true;
    Lost:
        if (validateReadSet(txnDesc) == false)
            abort();
        txnDesc->timestamp = timestamp;
    }
    TSR.status_bits = 0;
    return (txnRec == *txnRecPtr); /* check if txnrec changed */
}

```

Similarly, an STM system that does not maintain read-set consistency, such as an STM for managed code, may avoid infinite loops, or other incorrect control flow -- exceptions, due to inconsistency by inserting JLOSS.rm instruction at loop back edges , or
5 other critical control flow points, such as instructions that may raise exceptions.

The following Pseudo Code, labeled pseudo code B, shows another native code read barrier that provides consistency. This version TM system uses cache-resident write sets using buffered updates for writes inside transactions. A read from a location that was previously buffered and then lost causes inconsistency, so to maintain consistency, this
10 read barrier avoids reading from any lost buffered location. The COMMIT_LOCKING flag is true if the STM is using commit time locking for buffered locations. The jloss_buf() check is utilized on reads from a previously locked location when not using commit-time locking; otherwise, it is utilized on all reads.

Pseudo Code B: In-place update, native code STM read barrier

```

Type tmRd<Type>(TxnDesc* txnDesc, Type* addr) {
setrm(addr); /* set the read monitor on loaded address */
TxnRec* txnRecPtr = getTxnRecPtr(addr);
TxnRec txnRec = *txnRecPtr;
val = *addr;
if (txnRec != txnDesc) {
while (!validateAndLogUtm(txnDesc, txnRecPtr, txnRec)) {
/* retry */
txnRec = *txnRecPtr;
val = *addr;
}
} else if
jloss_buf Abort;
return val;
}

Abort
Abort();

bool checkReadConsistency(TxnDesc* txnDesc, TxnRec* txnRecPtr, TxnRec
txnRec) {
if (COMMIT_LOCKING && jloss_buf() == false)
abort(); /* abort if we lost buffered data */
if (txnRec > txnDesc->timestamp) {
TxnRec timestamp = GlobalTimestamp;
jloss_rm Lost:
txnDesc->timestamp = timestamp;
return true;
Lost:
if (validateReadSet(txnDesc) == false)
abort();
txnDesc->timestamp = timestamp;
}
}

```

```

    }
    return (txnRec == *txnRecPtr); /* check if txnrec changed */
}

```

TM systems may combine read monitoring with buffering and write monitoring, as discussed above, and thus also include checking for conflicts to either monitored or buffered lines to maintain consistency. To accommodate such systems, different embodiments may also provide JLOSS flavors that branch on logical combinations of

5 different monitoring and buffering events such as JLOSS.rm.buf (conflict on read monitored or buffered locations), JLOSS.rm.wm, (conflict on read or write monitored locations), or JLOSS.* (conflict on read monitored, write monitored, or buffered location).

In an alternate embodiment, the architectural interface decouples the JLOSS instruction from the conditions under which it branches by allowing software to setup the

10 conditions – conflict on read/write monitored lines or buffered lines – in a separate control register. This embodiment requires only a single JLOSS instruction encoding and can support future extensions to the set of events on which the JLOSS should branch.

Turning to **Figure 10**, an embodiment of a flow diagram for a method of executing a loss instruction to jump to a destination label based upon a conflict or loss of specific

15 information is depicted. In one embodiment, a JLOSS instruction is received in flow **1005**. As stated above, the JLOSS instruction may be inserted by a programmer or compiler within either main codes, such as after a load operation to ensure read set consistency, or within a barrier, such as within a read or write barrier. The JLOSS instruction, and its variants discussed above, are in one embodiment recognizable as part

20 of a processor's ISA. Here, decoders are able to decode the opcodes for the JLOSS instructions.

In flow **1010**, it is determined if a conflict or loss of information has occurred. In one embodiment, the type of conflict or loss is dependent on the variant of the JLOSS instruction. For example, if the received JLOSS instruction is a JLOSS.rm instruction,

25 then it is determined if a read monitored line has been conflictingly accessed by an external write. However, as stated above, any variant on JLOSS may be received, including a JLOSS instruction that allows the user to specify conditions in a control register.

Therefore, once the conditions are established, either from the control register or

type of JLOSS instruction, then it is determined if those conditions have been met. As a first example, information in a transaction status register, such as TSR **915** is utilized to determine if the conditions are satisfied. Here, TSR **915** may include a read monitor status flag, which by default is set to a no conflict value and is updated to a conflict value to indicate a conflict has occurred. Yet, a status register is not the only way for determining if a conflict has occurred, and in fact, any known method for determining a loss or conflict may be utilized.

In response to no conflict being detected, such as when a read monitor conflict flag is still set to a default value in TSR **915**, then a false value is returned in flow **1025** and execution continues normally. However, if a conflict or loss is detected, such as the read monitor conflict flag being set, then JLOSS returns true in flow **1015** and directs execution to jump to a label defined by the received JLOSS instruction in flow **1020**.

Hardware Support for Transactional Memory Commit

As previously discussed, hardware supported transactions may accelerate software's version management by buffering transactional writes in the cache without making them globally visible. In this case, a simple commit instruction may be utilized, which makes the buffered values visible to all processors, but fails if any buffered lines are lost. Yet, the ability of hardware to also hold metadata that software is able to use for acceleration, such as a filter to eliminate/filter redundant barriers, may want a commit instruction to fail if hardware detected any conflicts. In addition, upon commit, it may be desirable to clear different combinations of information held in hardware for a transaction, such as metadata, monitors, and buffered lines.

Therefore, in one embodiment, hardware supports multiple forms of a commit instruction to allow the commit instruction to specify both the conditions for commit and the information to clear upon commit. Referring to Figure 11, an embodiment of a general case for hardware to support definition of commit conditions and clear controls in a commit instruction is depicted.

As illustrated, commit instruction **1105** includes an opcode **1110**, which is recognizable as part of a processor's ISA – decoders **1115** are able to decode opcode **1110**. In the illustrated example, opcode **1110** includes two portions: commit conditions **1111** and clear control **1112**. Commit conditions **1111** are to specify the conditions for a transaction to commit, while commit clear control **1112** specifies the information to clear upon commit of a transaction.

In one embodiment, both portions includes four values: read monitoring (RM), write monitoring (WM), Buffering (Buf), and metadata (MD). Essentially, if any of the four values are set in portion **1111** – include a value to indicate that the associated attribute/property is a commit condition, then the corresponding property is a condition for commit. In other words, if the first bit of conditions **1111** corresponding to read monitor information is set, then the loss of any read monitoring data from monitors **1135** associated with the transaction results in an abort – no commit as a specified condition of the commit instruction failed. Similarly, if a value in **1112** is set, then the corresponding property is cleared upon the commit. Continuing the example, if RM in portion **1112** is set, then the read monitor information in monitors **1135** for the transaction are cleared when the transaction is committed. Therefore, in this example, there is a possibility of four conditions for commit and four clear controls, which results in 256 possible combinations as variations on a commit instruction. In one embodiment, by allowing the commit conditions to be specified in the opcode, the hardware is able to support all of the variations. However, a few variations are discussed below to further understanding of the different styles of commit instructions and how they may be utilized.

TXCOMWM

As a first example, a Txcomwm instruction is discussed. This instruction ends the transaction and makes all write-monitored buffered data globally visible if no write monitored data has been lost (success); otherwise, it fails if write monitored data has been lost. Txcomwm sets (or resets) a flag to indicate success (or failure). On success, Txcomwm clears the buffered state of all write monitored data. Txcomwm does not affect read or write monitoring state, allowing software to re-use such state in subsequent transactions; it also does not affect the state of locations that are buffered but not write monitored, allowing software to persist information kept in such locations. The pseudo code below labeled pseudo code C illustrates an algorithmic description of Txcomwm. When TSR.LOSS_WM is 0, the BF property of all *write monitored* buffered BBLKs is atomically cleared and all such buffered data becomes visible to other agents. TCR.IN_TX is cleared. Buffered blocks that lack WM are not affected and remain buffered. The CF flag is set upon completion. When TSR.LOSS_WM is 1, the CF flag is cleared and TCR.IN_TX is cleared. CF flag is set to 1 if the operation succeeded and set to 0 for failure. The OF, SF, ZF, AF, and PF flags are set to 0.

Pseudo Code C: Embodiment of algorithm for Txcomwm Operation

```

        atomically
        {
        if (TSR.LOSS_WM == 1)
        {
5         CF := 0;
        }
        else
        {
        for (all mblk)
10         {
            CommitAllInMblk(mblk)
        }

        CF := 1;
15         }
        }

        TCR.IN_TX := 0;
        OF := 0;
20         SF := 0;
        ZF := 0;
        AF := 0;
        PF := 0;

```

25 The pseudo code below labeled Pseudo Code D shows how a HASTM system is able to use the Txcomwm instruction to commit a transaction that uses hardware write buffering to avoid undo logging in an in-place update STM. The CACHE_RESIDENT_WRITES flag indicates this execution mode. illustrates an embodiment of how a HASTM

Pseudo Code D: Embodiment of pseudo code for use of Txcomwm instruction

```

Void tmCommitUtm(TxnDesc* txnDesc) {
    if (CACHE_RESIDENT_WRITES) {

```

```

if (LAZY_LOCKING) {
  if (EAGER_MONITORING == false) {
    /* Lazy locking & lazy monitoring */
    setWriteMonitors(txnDesc);
    /* abort if any buffered lines lost during setting of write monitors */
    if (EJECTOR_ENABLED == false && checkTsrLoss(LOSS_BF))
      abort(txnDesc);
  }
  /* end the transaction to disable ejectors while we acquire locks */
  if (EJEcTOR_ENABLED)
    tx();
  acquireWriteLocks(txnDesc);
}
/* commit write monitored lines and end the transaction */
if (txcomwm() == false) {
  tx(); /* clear buffering & monitoring */
  abort(txnDesc);
}
} else {
  /* unbounded writes */
  tx(); /* end the transaction */
}
TxnRec myCommitTimestamp = lockedIncrement(&GlobalTimestamp);
if (myCommitTimestamp == txnDesc->timestamp-1 &&
    validateReadSet(txnDesc) == false)
  tmRollbackAndAbort(txnDesc, myCommitTimestamp);
releaseWriteLocks(txnDesc, myCommitTimestamp);
quiesce(txnDesc);
}

```

TXCOMWMRM

One variant, txcomwmrm, extends the Txcomwm instruction so that it fails if any read monitored locations have also been lost. This variant is useful for transactions that

use only hardware to detect read-set conflicts. The pseudo code below labeled pseudo code E illustrates an algorithmic description of Txcomwmm. When TSR.LOSS_WM and TSR.LOSS_RM are 0, the BF property of all *write monitored* buffered BBLKs is atomically cleared and all such buffered data becomes visible to other agents. TCR.IN_TX is cleared. Buffered blocks that lack WM are not affected and remain buffered. The CF flag is set upon completion. When TSR.LOSS_WM or TSR.LOSS_RM are 1, the CF flag is cleared and TCR.IN_TX is cleared. The CF flag is set to 1 if the operation succeeded and cleared to 0 for failure. The OF, SF, ZF, AF, and PF flags are set to 0.

Pseudo Code E: An embodiment of an algorithmic description of Txcomwmm

```

10         atomically
           {
           if ((TSR.LOSS_RM == 1) || (TSR.LOSS_WM == 1))
           {
15         CF := 0;
           }
           else
           {
           for (all mblk)
20         {
           CommitAllInMblk(mblk)
           }

           CF := 1;
25         }
           }

           TCR.IN_TX := 0;
           OF := 0;
30         SF := 0;
           ZF := 0;
           AF := 0;
           PF := 0;

```

- The next pseudo-code, Pseudo Code F, shows the commit algorithm utilizing txcomwrm instruction for an STM system that uses hardware both to buffer transactional writes and to detect read-set conflicts. The HW_READ_MONITORING flag indicates
- 5 whether the algorithm uses only hardware for read-set conflict detection.

Pseudo Code F: An embodiment of pseudo code utilizing txcomwrm instruction

```

Void tmCommitUtm(TxnDesc* txnDesc) {
    if (CACHE_RESIDENT_WRITES) {
        if (LAZY_LOCKING) {
            if (EAGER_MONITORING == false) {
                /* Lazy locking & lazy monitoring */
                setWriteMonitors(txnDesc);
                /* abort if any buffered lines lost during setting of write monitors */
                if (EJECTOR_ENABLED == false && checkTsrLoss(LOSS_BF))
                    abort(txnDesc);
            }
            /* end the transaction to disable ejectors while we acquire locks */
            if (EJECTOR_ENABLED)
                tx();
            acquireWriteLocks(txnDesc);
        }
        /* commit write monitored lines and end the transaction */
        if (HW_READ_MONITORING) {
            if (txcomwrm() == false) {
                txa(); /* clear buffering & monitoring */
                abort(txnDesc);
            }
            } else if (txcomwm() == false) {
                txa(); /* clear buffering & monitoring */
                abort(txnDesc);
            }
        } else {

```

```

/* unbounded writes */
tx(); /* end the transaction */
}
TxnRec myCommitTimestamp = lockedIncrement(&GlobalTimestamp);
if (HW_READ_MONITORING == false &&
myCommitTimestamp == txnDesc->timestamp-1 &&
validateReadSetUtm(txnDesc) == false)
tmRollbackAndAbort(txnDesc, myCommitTimestamp);
releaseWriteLocks(txnDesc, myCommitTimestamp);
quiesce(txnDesc);
}

```

TXCOMWMIRMC

A third discussed variant is illustrated in the algorithm description of Pseudo Code F below. When TSR.LOSS_WM and TSR.LOSS_IRM are 0, the BF property of all *write* 5 *monitored* buffered BBLKs is atomically cleared and all such buffered data becomes visible to other agents. RM, WM and IRM, as well as TCR.IN_TX is cleared. Buffered blocks that lack WM are not affected and remain buffered. The CF flag is set upon completion. When TSR.LOSS_WM or TSR.LOSS_IRM are 1, the CF flag is cleared and TCR.IN_TX is cleared. The CF flag is set to 1 if the operation succeeded and cleared to 0 10 for failure. The OF, SF, ZF, AF, and PF flags are set to 0.

Pseudo Code F: An embodiment of an algorithmic description for Txcomwmirmc

instruction

```

atomically
{
15   if ((TSR.LOSS_IRM == 1) || (TSR.LOSS_WM == 1))
   {
   CF := 0;
   }
   else
20   {
   for (all mb1k)
   {

```

```

CommitAllInMblk(mblk)

    mblk.RM := 0;
    mblk.WM := 0;
5    mblk.IRM := 0;
    }
    CF := 1;
    }
    }

10    TCR.IN_TX := 0;
    OF := 0;
    SF := 0;
    ZF := 0;
15    AF := 0;
    PF := 0;

```

Referring to **Figure 12**, an embodiment of a flow diagram for a method of executing a commit instruction, which defines commit conditions and clear controls is illustrated. In flow **1205** a commit instruction is received. As stated above, a compiler may insert a commit instruction in program code. As a specific illustrative example, a call to a commit function is inserted in main code and the commit function, such as those included above in pseudo code, are provided in a library; a compiler may also insert the commit instruction into the commit function within the library.

After the commit instruction is received, decoders are capable of decoding the commit instruction. From the decoded information, the conditions specified by the opcode of the commit instruction are determined in flow **1210**. As described above, the opcode may set some flags and reset others to indicate what conditions are to be utilized for commit. If the conditions are not satisfied, then false is returned and the transaction may be separately aborted. However, if the conditions for commit, such as any combination of no loss of read monitors, write monitors, metadata, and/or buffering, then in flow **1215** the clear conditions/control are determined. As an example, any combination of read monitors, write monitors, metadata, and/or buffering for the transaction is determined to be cleared. As a result, the information determined to be cleared is cleared in flow **1225**.

Optimized Memory Management for UTM

As discussed above, Unbounded transactional memory (UTM) architecture and its hardware implementation extend the processor architecture by introducing the following properties: monitoring, buffering and metadata. These combined provide software the means necessary to implement a variety of sophisticated algorithms, including a wide spectrum of transactional memory designs. Each property may be implemented in hardware by either extending the existing cache protocols in the cache implementation or allocating independent new hardware resources.

With UTM properties implemented by HW, UTM architecture and its hardware implementations potentially provide performance boost over the software only solution (STM) on transactions if it is able to effectively avoid and minimize the incidents such as UTM transaction aborts and subsequent transaction retry-operations. One of the major causes of the hardware transaction aborts was due to frequent ring transition caused by external interrupts, system call events and page faults.

A current privilege level (CPL) based suspension mechanism makes hardware transaction active (enabling hardware accelerated transaction with UTM properties such as buffering and monitoring and enabling the ejection mechanism), while the processor is operating at the privilege level 3 (user mode). Any ring transitions from the ring 3 causes the currently active transactions to automatically suspend (stopping to generate UTM properties and disabling the ejection mechanism). Similarly, any ring transitions back to the ring 3 automatically resumes the previously suspended hardware transaction if it were active. The potential downside of this approach is that use of the hardware transactional memory resources in the kernel code or at any other ring levels, except at the ring 3, are mostly precluded.

Another approach is introducing duplicated TM control resources such as a transaction control register (TxCR) for the ring 0 so that we can still enable the hardware transactions for the ring 0 code with these separate TM resources. However, this approach potentially lacks an efficient solution for handling nested interrupts and exceptions during ring 0 transaction operations.

As a result, **Figure 13** illustrates an embodiment of hardware to support handling privilege level transitions during execution of transactions, which enables ring 0 transactions on top of the user mode (ring 3) transactions, but also provides for the OS and a hypervisor, such as a Virtual Machine Monitor (VMM) to handle infinite levels of nested interrupts and NMI cases with presence of ring 0 transactions.

A storage element, such as EFLAGS register **1310**, includes transaction enable field (TEF) **1311**. When TEF **1311** holds an active value it indicates that a transaction is currently active and enabled, while when TEF **1311** holds an inactive value it indicates that a transaction is suspended.

5 In one embodiment, a transaction begin operation, or other operation at a start of a transaction, sets the TEF field **1311** to the active value. Upon a ring level transition event at flow **1300**, such as an interrupt, exception, system call, exit of a virtual machine, or enter of a virtual machine, the state of PE 0 Eflags register **1310** is pushed on kernel stack **1320** in flow **1301**. At flow **302**, the TEF field **1311** is cleared/updated to the inactive
10 value to suspend the transaction. The ring level transition event is handled or serviced appropriately while the transaction is suspended. Upon detecting a return event at flow **1303**, the state of Eflags register **1310**, which was pushed onto the stack at flow **1301**, is popped at flow **1304** to restore Eflags **1310** with the previous state. The restore of the previous state returns TEF **1311** to the active value and resumes the transaction as active
15 and enabled.

 Specific examples of the process for illustrative ring level transition events are listed below. Upon interrupts and exceptions, the processor pushes the EFLAGS register into the kernel stack and clears “Transaction Enable” bit if it is set, suspending the
20 previously enabled transaction. Upon IRET, the processor restores the entire EFLAGS register state for the interrupted thread including the “Transaction Enable” bit from the kernel stack, un-suspending the transaction if it was previously enabled.

 Upon SYSCALL, the processor pushes the EFLAGS register and clears “Transaction Enable” if it is set, suspending the previously enabled transaction. Upon
25 SYSRET, the processor restores the entire EFLAGS register state for the interrupted thread including the “Transaction Enable” bit from the kernel stack, un-suspending the transaction if it was previously enabled.

 Upon VM-Exit, the processor saves the EFLAGS register of the guest including the Transaction Enable” bit state into the Virtual Machine Control Structure (VMCS) and loads up the EFLAGS register state of the host which “Transaction Enable” bit state is
30 clear, suspending the previously enabled transaction of the guest if enabled.

 Upon VM-Enter, the processor restores the EFLAGS register of the guest including the Transaction Enable” bit state from the VMCS, un-suspending the previously enabled transaction of the guest if it was enabled.

This enables the kernel mode (ring 0) hardware accelerated UTM transactions on top of the user mode (ring 3) hardware accelerated UTM transactions but also provides ways for both the OS and VMM to handle infinite levels of nested interrupts and NMI cases with presence of ring 0 transactions. None of the prior arts provided such mechanisms.

A module as used herein refers to any hardware, software, firmware, or a combination thereof. Often module boundaries that are illustrated as separate commonly vary and potentially overlap. For example, a first and a second module may share hardware, software, firmware, or a combination thereof, while potentially retaining some independent hardware, software, or firmware. In one embodiment, use of the term logic includes hardware, such as transistors, registers, or other hardware, such as programmable logic devices. However, in another embodiment, logic also includes software or code integrated with hardware, such as firmware or micro-code.

A value, as used herein, includes any known representation of a number, a state, a logical state, or a binary logical state. Often, the use of logic levels, logic values, or logical values is also referred to as 1's and 0's, which simply represents binary logic states. For example, a 1 refers to a high logic level and 0 refers to a low logic level. In one embodiment, a storage cell, such as a transistor or flash cell, may be capable of holding a single logical value or multiple logical values. However, other representations of values in computer systems have been used. For example the decimal number ten may also be represented as a binary value of 1010 and a hexadecimal letter A. Therefore, a value includes any representation of information capable of being held in a computer system.

Moreover, states may be represented by values or portions of values. As an example, a first value, such as a logical one, may represent a default or initial state, while a second value, such as a logical zero, may represent a non-default state. In addition, the terms reset and set, in one embodiment, refer to a default and an updated value or state, respectively. For example, a default value potentially includes a high logical value, i.e. reset, while an updated value potentially includes a low logical value, i.e. set. Note that any combination of values may be utilized to represent any number of states.

The embodiments of methods, hardware, software, firmware or code set forth above may be implemented via instructions or code stored on a machine-accessible or machine readable medium which are executable by a processing element. A machine-

accessible/readable medium includes any mechanism that provides (i.e., stores and/or transmits) information in a form readable by a machine, such as a computer or electronic system. For example, a machine-accessible medium includes random-access memory (RAM), such as static RAM (SRAM) or dynamic RAM (DRAM); ROM; magnetic or
5 optical storage medium; flash memory devices; electrical storage device, optical storage devices, acoustical storage devices or other form of propagated signal (e.g., carrier waves, infrared signals, digital signals) storage device; etc. For example, a machine may access a storage device through receiving a propagated signal, such as a carrier wave, from a medium capable of holding the information to be transmitted on the propagated signal.

10 Reference throughout this specification to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrases “in one embodiment” or “in an embodiment” in various places throughout this specification are not necessarily all referring to the same embodiment.
15 Furthermore, the particular features, structures, or characteristics may be combined in any suitable manner in one or more embodiments.

 In the foregoing specification, a detailed description has been given with reference to specific exemplary embodiments. It will, however, be evident that various
modifications and changes may be made thereto without departing from the broader spirit
20 and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense. Furthermore, the foregoing use of embodiment and other exemplarily language does not necessarily refer to the same embodiment or the same example, but may refer to different and distinct embodiments, as well as potentially the same embodiment.

25

CLAIMS

What is claimed is:

1. An apparatus comprising:
5 a plurality of processing elements, wherein a processing element of the plurality of processing elements is to be associated with a plurality of software subsystems;
metaphysical logic to associate a metadata access operation, which is to be associated with a current software subsystem of the plurality of software
10 subsystems and is to reference a data address, with a metaphysical address space associated with the current software subsystem based at least on the data address and a metadata identifier (MDID) associated with the current software subsystem.
- 15 2. The apparatus of claim 1, wherein the metaphysical address space associated with the current software subsystem is to be orthogonal to a data address space including the data address and at least one other metaphysical address space associated with a second software subsystem of the plurality of software subsystems.
- 20 3. The apparatus of claim 2, wherein each of the plurality of software subsystems are individually selected from a group consisting of a transactional runtime subsystem, a garbage collection runtime subsystem, a memory protection subsystem, a software translation subsystem, an outer transaction of a nested group of
25 transactions, and an inner transaction of a nested group of transactions.
4. The apparatus of claim 1, further comprising decoding logic to decode the metadata access operation, wherein the metadata access operation includes an operation code (opcode) recognized as one of a plurality of supported operations
30 within the decoding logic.
5. The apparatus of claim 1, wherein the metaphysical logic includes metaphysical translation logic to translate the data address to a metadata address within the

metaphysical address space associated with the current software subsystem based on at least the MDID.

- 5 6. The apparatus of claim 5, wherein the metaphysical translation logic to translate the data address to a metadata address within the metaphysical address space associated with the current software subsystem is further based on a processing element identifier (PEID) associated with the processing element.
- 10 7. The apparatus of claim 6, wherein the metaphysical translation logic to translate the data address to a metadata address within the metaphysical address space associated with the current software subsystem is further based on a compression ratio of data to metadata.
- 15 8. The apparatus of claim 6, further comprising a register, which is modifiable by the current software subsystem, wherein the register is to hold the MDID in response to a write from the current software subsystem to indicate the current software subsystem is currently executing on the processing element, and wherein the metaphysical translation logic to translate the data address to a metadata address within the metaphysical address space associated with the current software
20 subsystem is based on the PEID and the MDID comprises the metaphysical translation logic to combine a representation of the data address with the PEID and the MDID.
- 25 9. The apparatus of claim 8, wherein the metaphysical translation logic to combine a representation of the data address with the PEID and the MDID is based on a combination algorithm selected from a group consisting of an algorithm to add the PEID and the MDID to the data address to form the metadata address, an algorithm to translate the data address to a translated data address utilizing normal data translation tables and adding the PEID and MDID to the translated address to form
30 the metadata address, and an algorithm to translate the data address to a translated metadata address utilizing metaphysical translation tables separate from normal data translation tables and adding the PEID and MDID to the translated metadata address to form the metadata address.

10. A method comprising:
encountering a metadata operation referencing a data address, which is within a
data address space and is associated with a data item held in a data entry of
5 a cache memory;
determining a metadata address within a metaphysical address space disjoint from
the data address space based on the data address, a processing element
identifier (PEID) of a processing element associated with the metadata
operation, and a metadata identifier (MDID) for a software subsystem
10 associated with the processing element;
accessing a metadata entry of the cache memory based on the metadata address.
11. The method of claim 10, wherein the metaphysical address space is also disjoint
from an additional metaphysical address space, which is associated with an
15 additional software subsystem also associated with the processing element.
12. The method of claim 10, wherein the software subsystem is selected from a group
consisting of a transactional runtime subsystem, a garbage collection runtime
subsystem, a memory protection subsystem, a software translation subsystem, an
20 outer transaction of a nested group of transactions, and an inner transaction of a
nested group of transactions.
13. The method of claim 10, further comprising:
writing the MDID to a control register associated with the processing element in
25 response to encountering a write operation to the control register from the
software subsystem responsive to the software subsystem currently
executing on the processing element; and
determining the MDID from the control register.
- 30 14. The method of claim 13, further comprising: determining the PEID from a portion
of an opcode for the metadata operation.
15. The method of claim 13, wherein determining the metadata address from the data

address, the PEID, and the MDID comprises: combining the data address, the PEID, and the MDID with an algorithm selected from a group consisting of an algorithm to add the PEID and the MDID to the data address to form the metadata address, an algorithm to translate the data address to a translated data address
5 utilizing normal data translation tables and adding the PEID and MDID to the translated address to form the metadata address, and an algorithm to translate the data address to a translated metadata address utilizing metaphysical translation tables separate from normal data translation tables and adding the PEID and MDID to the translated metadata address to form the metadata address

10

16. An apparatus comprising:

decode logic to decode a metadata access instruction, which is to reference a data address of a data item, the metadata access instruction to include an opcode recognizable as part of an instruction set capable of being properly decoded
15 by the decoding logic; and
metadata logic to translate the data address to a distinct metadata address transparently to software and to access metadata referenced by the distinct metadata address in response to the decoding logic decoding the metadata access instruction.

20

17. The apparatus of claim 16, wherein the metadata access instruction is selected from a group of instructions consisting of a metadata bit test and set (MDLT) instruction, a metadata store and set (MSS) instruction, and a metadata store and reset instruction (MDSR).

25

18. The apparatus of claim 16, wherein the metadata access instruction is selected from a group of instructions consisting of a compressed metadata test (CMDT) instruction, a compressed metadata store (CMS) instruction, and a compresses metadata clear (CMDCLR) instruction.

30

19. The apparatus of claim 16, wherein the metadata logic to translate the data address to a distinct metadata address transparently to software comprises translating the data address based at least on a metadata identifier (MDID) specified in a control

register by a software subsystem, which is associated with the metadata access instruction.

- 5 20. The apparatus of claim 16, wherein the metadata access instruction is also to include a reference to a destination register, and wherein the metadata logic to access metadata referenced by the distinct metadata address comprises the metadata logic to load the metadata at the referenced distinct metadata address into the destination register.
- 10 21. The apparatus of claim 20, wherein the opcode includes a thread identifier field to identify the thread the metadata access instruction originated from.
- 15 22. The apparatus of claim 20, wherein the metadata logic to access metadata referenced by the distinct metadata address further comprises the metadata logic to set the metadata at the referenced distinct metadata address to a set value in response to the metadata loaded into the destination register being an unset value.
- 20 23. The apparatus of claim 22, wherein the set and unset values are specified in the metadata access instruction.
- 25 24. A machine readable medium holding program code, which when executed by a machine, causes the machine to perform the operations of:
responsive to a data access operation, which references a data address: generating a metadata access operation to reference the data address at the data access operation, the metadata access operation, when executed by the machine, to cause the machine to:
translate the data address to a metadata address, which is disjoint from the data address, and
access metadata for a data item at the data address based on the metadata address.
- 30 25. The machine readable medium of claim 24, wherein the metadata access operation is selected from a group of instructions consisting of a metadata bit test and set

(MDLT) instruction, a metadata store and set (MSS) instruction, and a metadata store and reset instruction (MDSR).

- 5 26. The machine readable medium of claim 24, wherein the metadata access operation is selected from a group of compression instructions consisting of a compressed metadata test (CMDT) instruction, a compressed metadata store (CMS) instruction, and a compresses metadata clear (CMDCLR) instruction.
- 10 27. The machine readable medium of claim 26, wherein the metadata access operation, when executed by the machine, to cause the machine to translate the data address to a metadata address comprises the metadata access operation, when executed by the machine, to cause the machine to combine the data address with a processing element identifier (PEID) associated with the metadata access operation and a metadata data identifier (MDID) associated with the metadata access operation
15 based on a compression ratio of data to metadata.
- 20 28. The machine readable medium of claim 27, wherein the data address is also capable of being translated by virtual to physical address translation logic in the machine to reference the data item.
- 25 29. The machine readable medium of claim 24, wherein the metadata access operation also references an operand register, and wherein the metadata access operation, when executed by the machine, to cause the machine to access metadata for the data item comprises the metadata access operation, when executed by the machine,
30 to cause the machine to update the metadata for the data item with a value held in the operand register.
- 30 30. The machine readable medium of claim 24, wherein the program code includes compiler code, and wherein the compiler code is to compile application code including the data access operation, and wherein generating the metadata access operation at the data access operation includes generating the metadata access operation within a compiled version of the application code.

31. A machine readable medium holding program code, which when executed by a machine, causes the machine to perform the operations of:
translating a data address referenced by a metadata access instruction within the
program code to a metadata address based on a metadata identifier (MDID)
5 associated with a software subsystem currently active on a processing
element associated with the metadata access instruction; and
accessing metadata based on the metadata address.
32. The machine readable medium of claim 31, wherein the metadata access
10 instruction is selected from a group of instructions consisting of a metadata load
instruction to load the metadata, a metadata store instruction to store to the
metadata, and a metadata clear instruction to reset the metadata.
33. The machine readable medium of claim 31, wherein the software subsystem is
15 selected from a group consisting of a transactional runtime subsystem, a garbage
collection runtime subsystem, a memory protection subsystem, a software
translation subsystem, an outer transaction of a nested group of transactions, and
an inner transaction of a nested group of transactions.
34. The machine readable medium of claim 31, wherein translating a data address
20 referenced by a metadata access instruction within the program code to a metadata
address based on a metadata identifier (MDID) associated with a software
subsystem currently active on a processing element associated with the metadata
access instruction comprises combining the data address with the MDID based on
a combination algorithm selected from a group consisting of an algorithm to add
25 the MDID to the data address to form the metadata address, an algorithm to
translate the data address to a translated data address utilizing normal data
transactional tables and adding the MDID to the translated address to form the
metadata address, and an algorithm to translate the data address to a translated
metadata address utilizing metaphysical translation tables separate from normal
30 data translation tables and adding the MDID to the translated metadata address to
form the metadata address.
35. The machine readable medium of claim 34, wherein adding the MDID comprises

an algorithm of adding the MDID selected from a group consisting of an algorithm to append the MDID in a MSB position, an algorithm to append the MDID in a LSB position, and an algorithm to replace address bits with the MDID.

- 5 36. The machine readable medium of claim 34, wherein the program code, which when executed by the machine, further causes the machine to perform the operations of: determining the MDID from a control register for the processing element, which is to represent the current software subsystem is currently active on the processing element.
- 10 37. A system comprising:
a memory to hold program code including a metadata access instruction, which is to reference a data memory address associated with a data item;
a processor associated with the memory, the processor including a processing element of a plurality of processing elements to be associated with
15 execution of the metadata access instruction, fetch logic to fetch the metadata access instruction from the memory, decode logic to decode the metadata access instruction into at least a metadata access operation, a control register to hold a metadata identifier (MDID) associated with an active context on the processing element, a data cache memory to include a
20 data entry to hold the data item, and execution logic to execute the metadata access operation, wherein the execution logic to execute the metadata access operation includes metaphysical address translation logic in the processor to translate the data memory address to a metadata memory address based on the MDID held in the control register and cache control
25 logic coupled to the data cache memory to perform the metadata access operation to a separate entry of the data cache memory based on the metadata memory address.
- 30 38. The system of claim 37, wherein the metadata access instruction is selected from a group of instructions consisting of a metadata load instruction to load the metadata, a metadata store instruction to store to the metadata, and a metadata clear instruction to reset the metadata.

39. The system of claim 37, wherein the active context is selected from a group consisting of a transactional runtime subsystem, a garbage collection runtime subsystem, a memory protection subsystem, a software translation subsystem, an outer transaction of a nested group of transactions, and an inner transaction of a nested group of transactions.
40. The system of claim 37, wherein the metaphysical address translation logic in the processor to translate the data memory address to a metadata memory address is further based on a processing element identifier (PEID) for the processing element, and wherein the metaphysical address translation logic in the processor to translate the data memory address to a metadata memory address based on the MDID held in the control register and the PEID comprises combining the data address with the MDID and the PEID based on a combination algorithm selected from a group consisting of an algorithm to add the MDID and the PEID to the data address to form the metadata address, an algorithm to translate the data address to a translated data address utilizing normal data transactional tables and adding the MDID and the PEID to the translated address to form the metadata address, and an algorithm to translate the data address to a translated metadata address utilizing metaphysical translation tables separate from normal data translation tables and adding the MDID and the PEID to the translated metadata address to form the metadata address.
41. A processor comprising:
an execution module to execute a metadata load operation, which is to reference an address;
a force module, in response to the metadata load operation, to provide a metadata value associated with address responsive to the processor operating in a first mode and to provide a fixed value responsive to the processor operating in a second mode.
42. The processor of claim 41, wherein the first mode includes a strong atomicity mode and the second mode includes a weak atomicity mode.

43. The processor of claim 42, further comprising a first register to hold the fixed value.
44. The processor of claim 43, further comprising a second register to hold a mode value, wherein when the mode value is to represent a first value to indicate the processor is operation in the strong atomicity mode and the mode value is to represent a second value to indicate the processor is operation in the weak atomicity mode.
45. The processor of claim 44, wherein the first and the second registers are the same metadata control register.
46. The processor of claim 44, wherein the force module to provide a metadata value associated with address responsive to the processor operating in the strong atomicity mode and to provide a fixed value responsive to the processor operating in the weak atomicity mode comprises the force module to load the metadata value into a destination register to be specified by the metadata load operation responsive to the mode value to be held in the second register representing the first value to indicate the processor is operating in the strong atomicity mode and to load the fixed value from the first register into the destination register responsive to the mode value to be held in the second register representing the second value to indicate the processor is operating in the weak atomicity mode.
47. A method comprising:
- encountering a metadata access operation referencing an address;
 - determining a mode of processor execution;
 - providing a metadata value associated with the address for the metadata access operation in response to determining the mode of processor execution is a first mode of execution; and
 - providing a fixed value from a register for the metadata access operation in response to determining the mode of processor execution is a second mode of execution.

48. The method of claim 47, wherein determining a mode of processor execution comprises reading a mode flag from a first control register, the mode flag to hold a first value to indicate the mode of processor execution is a first mode of execution and to hold a second value to indicate the mode of processor execution is a second mode of execution.
49. The method of claim 47, wherein providing a metadata value associated with the address for the metadata access operation comprises loading the metadata value from a memory location associated with the address into a destination register referenced by the metadata access operation.
50. The method of claim 49, wherein providing a fixed value from a register for the metadata access operation comprises loading the fixed value from the register into the destination register.
51. A system comprising:
- a memory to hold a metadata load operation to reference an address and a destination register;
 - a processor associated with the memory, the processor including execution logic to execute the metadata load operation, a metadata register to hold a forced value, a cache memory to hold a metadata value associated with the address, and force logic, in response to execution logic to execute the metadata load operation, to provide the metadata value to the destination register responsive to the processor operating in a first mode and to provide the forced value from the metadata register to the destination register responsive to the processor operating in a second mode of operation.
52. The system of claim 51, wherein the force logic is further to determine if the processor is operating in the first mode or the second mode.
53. The system of claim 52, wherein the first mode includes a strong atomicity mode and the second mode includes a weak atomicity mode.

54. The system of claim 52, wherein the metadata register is further to hold a mode value, the mode value to represent a first value when the processor is operating in the first mode and to represent a second value when the processor is operating in the second mode, wherein the force logic is further to determine if the processor is operating in the first mode or the second mode comprises the force logic to interpret the mode value from the metadata register.
55. The system of claim 52, further comprising a control register to hold a mode value, the mode value to represent a first value when the processor is operating in the first mode and to represent a second value when the processor is operating in the second mode, and wherein the force logic is further to determine if the processor is operating in the first mode or the second mode comprises the force logic to interpret the mode value from the control register.
56. The system of claim 51, wherein the memory is selected from a group consisting of a dynamic random access memory (DRAM), a static random access memory (SRAM), and a non-volatile memory.
57. A apparatus comprising:
a data cache array to hold a cache entry;
cache control logic coupled to the data cache array, the cache control logic to transition the cache entry from an unmonitored state to a buffered coherency and read monitored state upon a buffered update to the cache entry, and to subsequently transition the cache entry to a buffered coherency and write monitored state before a transition of the cache entry to a modified state to commit the buffered update.
58. The apparatus of claim 57, wherein the buffered update to the cache entry includes an update selected from a group consisting of a transactional memory access to a data address for a data item to be held in the cache entry, a metadata access to a data address associated with metadata to be held in the cache entry, and a local update to the cache entry.

59. The apparatus of claim 57, wherein the cache control logic to transition the cache entry from an unmonitored state to a buffered coherency and read monitored state comprises the cache control logic to update coherency bits associated with the cache entry to a buffered value to represent the buffered coherency state and to
5 update a read monitor attribute bit associated with the cache entry to a read monitored value to represent the read monitored state.
60. The apparatus of claim 59, wherein the cache control logic to subsequently
10 transition the cache entry to a buffered coherency and write monitored state before a transition of the cache entry to a modified state to commit the buffered update comprises the cache control logic to maintain the coherency bits associated with the cache entry at the buffered value to represent the buffered coherency state and to update a write monitor attribute bit associated with the cache entry to a write
15 monitored value to represent the write monitored state.
61. The apparatus of claim 60, wherein the cache control logic to transition the cache entry to the modified state comprises the cache control logic to update the coherency bits associated with the cache entry to a modified value to represent the modified coherency state.
20
62. The apparatus of claim 57, further comprising execution logic to execute the buffered update and to subsequently execute a commit operation, wherein the cache control logic to subsequently transition the cache entry to a buffered coherency and write monitored state before a transition of the cache entry to a
25 modified state to commit the buffered update is in response to the execution logic executing the commit operation.
63. A method comprising:
30 encountering a buffered update to a block of a cache memory;
applying read monitoring to the block upon encountering the buffered update to the block of the cache memory; and
subsequently applying write monitoring to the block before committing the block.

64. The method of claim 63, wherein the buffered update to the block of the cache memory includes a transactional write to the block of the cache memory.
- 5 65. The method of claim 63, further comprising performing the buffered update to the block of the cache memory simultaneously with applying read monitoring, wherein the block is held in a buffered coherency state after performing the buffered update.
- 10 66. The method of claim 63, further comprising performing the buffered update to the block of the cache memory after applying read monitoring, wherein the block is held in a buffered coherency state after performing the buffered upate.
- 15 67. The method of claim 63, wherein applying read monitoring to the block upon encountering the buffered update to the block of the cache memory comprises:
generating a read request for the block to processing elements external to a cache domain of the cache memory; and
updating a read monitor attribute associated with the block of the cache memory to a read monitor value to apply read monitoring to the block in response to detecting no conflicts from the processing elements external to the cache domain responsive to the read request for the block.
- 20 68. The method of claim 67, wherein subsequently applying write monitoring to the block before committing the comprises:
generating a read for ownership request for the block to the processing elements external to the cache domain of the cache memory; and
25 updating a write monitor attribute associated with the block of the cache memory to a write monitor value to apply write monitoring to the block in response to detecting no conflicts from the processing elements external to the cache domain responsive to the read for ownership request for the block.
- 30 69. The method of claim 68, wherein committing the block comprises: transitioning a cache coherency state of the block from a buffered coherency state to a modified coherency state.

70. A machine accessible medium holding program code, which when executed by a machine, causes the machine to perform the operations of:
applying read monitoring to a block of a cache memory upon a buffered write to
5 the block;
performing the buffered write to the block; and
applying write monitoring to the block subsequent to applying the read monitoring
and before committing the block.
- 10 71. The machine accessible medium of claim 70, wherein applying read monitoring to the block upon a buffered write to the block of the cache memory comprises:
generating a read request for the block to processing elements external to a cache
domain of the cache memory; and
updating a read monitor attribute associated with the block of the cache memory to
15 a read monitor value to apply read monitoring to the block in response to
detecting no conflicts from the processing elements external to the cache
domain responsive to the read request for the block..
- 20 72. The machine accessible medium of claim 71, wherein applying write monitoring to the block subsequent to applying the read monitoring and before committing the
block comprises:
generating a read for ownership request for the block to the processing elements
external to the cache domain of the cache memory; and
updating a write monitor attribute associated with the block of the cache memory
25 to a write monitor value to apply write monitoring to the block in response
to detecting no conflicts from the processing elements external to the cache
domain responsive to the read for ownership request for the block.
- 30 73. The machine accessible medium of claim 70, wherein applying write monitoring to the block subsequent to applying the read monitoring and before committing the
block is in response to encountering a commit operation:
74. The machine accessible medium of claim 70, wherein committing the block

comprises: transitioning a cache coherency state of the block to a modified coherency state.

75. A system comprising:

- 5 a system memory to hold a transactional write referencing a memory address and a commit operation;
- a processor associated with the system memory, the processor including a cache memory to generate a read request for a cache line associated with the memory address in response to receiving the transactional write; transition
- 10 the cache line to a buffered and read monitored state in response to no conflicts being detected based on the read request; generate a read for ownership request in response to receiving the commit operation; transition the cache line to a buffered and write monitored state in response to no conflicts being detected based on the read for ownership request; and
- 15 transition the cache line to a modified state in response to transitioning the line to the buffered and write monitored state.

76. The system of claim 75, wherein the cache memory to transition the cache line to a buffered and read monitored state comprises the cache memory to update
- 20 coherency bits associated with the cache line to a buffered value to represent the buffered part of the buffered and read monitored state and to update a read monitor attribute bit associated with the cache line to a read monitored value to represent the read monitored portion of the buffered and read monitored state.

- 25 77. The system of claim 76, wherein the cache memory to transition the cache line to a buffered and write monitored state comprises the cache memory to maintain the coherency bits associated with the cache line at the buffered value to represent the buffered part of the buffered and write monitored state and to update a write monitor attribute bit associated with the cache line to a write monitored value to
- 30 represent the write monitored portion of the buffered and write monitored state.

78. The system of claim 77, wherein the cache memory to transition the cache line to a modified state in response to transitioning the line to the buffered and write

monitored state comprises: updating the coherency bits to a modified value to represent the modified state.

5 79. The system of claim 75, wherein the memory is selected from a group consisting of a dynamic random access memory (DRAM), a static random access memory (SRAM), and a non-volatile memory.

80. An apparatus comprising:
10 decoding logic to decode a loss instruction to provide a decoded element, the loss instruction to reference a label and to include an operation code (opcode), which is to be part of an instruction set recognizable by the decoding logic; a status storage element to include a loss field to hold a loss value, the loss value to indicate a loss event was detected; and
15 jump logic coupled to the status storage element to transfer control to the label based on the decoded element and the loss value to indicate the loss event was detected.

20 81. The apparatus of claim 80, wherein the label includes a jump destination address, and wherein a loss event is selected from a group consisting of a read monitor conflict indicating a write to a read monitored cache line may have occurred, a write monitor conflict indicating an access to a write monitored cache line may have occurred, and a loss of a buffered cache line.

25 82. The apparatus of claim 80, wherein the status storage element includes a register, and wherein the loss field to hold a loss value comprises a first bit to be set if a read monitor conflict was detected, a second bit to be set if a write monitor conflict was detected, a third bit to be set if a loss of buffered physical data was detected, and a fourth bit to be set if a loss of buffered metadata was detected.

30 83. The apparatus of claim 80, wherein the loss instruction includes a read monitor loss instruction and the opcode is to specify a read monitor loss event type, and wherein the jump logic to transfer control to the label based on the decoded element and the loss value to indicate the loss event was detected comprises the

jump logic to jump execution to the label in response to the loss field holding the loss value indicating the loss event that occurred was of the read monitor loss event type specified by the opcode of the read monitor loss instruction.

5 84. The apparatus of claim 80, wherein the loss instruction includes a write monitor loss instruction and the opcode is to specify a write monitor loss event type, and wherein the jump logic to transfer control to the label based on the decoded element and the loss field holding the loss value to indicate the loss event was detected comprises the jump logic to jump execution to the label in response to the
10 loss field holding the loss value indicating the loss event that occurred was of the write monitor loss event type specified by the opcode of the write monitor loss instruction.

15 85. The apparatus of claim 80, wherein the loss instruction includes a buffered loss instruction and the opcode is to specify a buffered loss event type, and wherein the jump logic to transfer control to the label based on the decoded element and the loss field holding the loss value to indicate the loss event was detected comprises the jump logic to jump execution to the label in response to the loss field holding the loss value indicating the loss event that occurred was of the buffered loss event
20 type specified by the opcode of the buffered loss instruction.

86. A machine accessible medium holding program code, which when executed by a machine, causes the machine to perform the operations of:
responsive to a loss instruction:

25 determining a status of a transaction held in a transactional status register, which is specified by the loss instruction and resides in the machine; and
 vectoring execution to a label specified by the loss instruction in response to the status of the transaction indicating a loss event associated
30 with the loss instruction was detected.

87. The machine accessible medium of claim 86, wherein the label includes a jump destination address, and wherein a loss event is selected from a group consisting of

a read monitor conflict indicating a write to a read monitored cache line may have occurred, a write monitor conflict indicating an access to a write monitored cache line may have occurred, and a loss of a buffered cache line.

- 5 88. The machine accessible medium of claim 86, wherein the loss instruction includes
a read monitor jump loss(JLOSS) instruction, which is to specify the loss event to
be a read monitor conflict, determining a status of a transaction held in the
transactional status register comprises determining a status of a read monitor
10 conflict bit held in the transaction status register, and vectoring execution to a label
specified by the loss instruction in response to the status of the transaction
indicating a loss event associated with the loss instruction was detected comprises
vectoring execution to a label specified by the loss instruction in response to the
status of the read monitor conflict bit held in the transaction status register
indicating a read monitor conflict was detected.
- 15 89. The machine accessible medium of claim 86, wherein the loss instruction includes
a write monitor jump loss(JLOSS) instruction, which is to specify the loss event to
be a write monitor conflict, determining a status of a transaction held in the
transactional status register comprises determining a status of a write monitor
20 conflict bit held in the transaction status register, and vectoring execution to a label
specified by the loss instruction in response to the status of the transaction
indicating a loss event associated with the loss instruction was detected comprises
vectoring execution to a label specified by the loss instruction in response to the
status of the write monitor conflict bit held in the transaction status register
25 indicating a write monitor conflict was detected.
- 30 90. The machine accessible medium of claim 86, wherein the loss instruction includes
a buffered monitor jump loss(JLOSS) instruction, which is to specify the loss event
to be a buffered monitor conflict, determining a status of a transaction held in the
transactional status register comprises determining a status of a buffered monitor
conflict bit held in the transaction status register, and vectoring execution to a label
specified by the loss instruction in response to the status of the transaction
indicating a loss event associated with the loss instruction was detected comprises

vectoring execution to a label specified by the loss instruction in response to the status of the buffered monitor conflict bit held in the transaction status register indicating a buffered monitor conflict was detected.

5 91. A method comprising:
 encountering a loss instruction in a processor;
 determining if a loss event associated with the loss instruction has been detected in
 the processor in response to encountering the loss instruction; and
 branching to a label referenced by the loss instruction in response to encountering
10 the loss instruction and determining the loss event associated with the loss
 instruction has been detected in the processor.

92. The method of claim 91, wherein the label includes a jump address.

15 93. The method of claim 91, wherein the loss instruction includes a read monitor loss
 instruction, and wherein the loss event associated with the read monitor loss
 instruction includes a write to a read monitored cache line.

20 94. The method of claim 91, wherein the loss instruction includes a write monitor loss
 instruction, and wherein the loss event associated with the write monitor loss
 instruction includes an access to a write monitored cache line.

25 95. The method of claim 91, wherein the loss instruction includes a buffered loss
 instruction, and wherein the loss event associated with the buffered loss instruction
 includes an eviction of a buffered cache line.

30 96. The method of claim 95, wherein determining if an eviction of a buffered cache
 line has been detected in the processor comprises checking a buffered loss status
 bit in a transaction status register and determining an eviction of a buffered cache
 line has been detected in response to the buffered loss status bit being set to a loss
 value.

97. An apparatus comprising:

- decoding logic to decode a commit instruction for a transaction to provide a
decoded element, the commit instruction to specify a commit condition and
to include an operation code (opcode), which is to be part of an instruction
set recognizable by the decoding logic; and
5 commit logic to determine if the commit condition to be specified by the commit
instruction is satisfied for the transaction in response to the decoded
element.
- 10 98. The apparatus of claim 97, wherein the commit condition includes any specified
combination of no loss of read monitored data, no loss of write monitored data, no
loss of buffered data, and no loss of metadata, and wherein the commit logic to
determine the commit condition is satisfied includes determining that the specified
combination of no loss of read monitored data, no loss of write monitored data, no
loss of buffered data, and no loss of metadata occurred.
- 15 99. The apparatus of claim 97, wherein the commit instruction to specify a commit
condition comprises the commit instruction to hold four bits: a first bit, when set,
to indicate any loss of read monitored data is a condition to commit, a second bit,
when set, to indicate any loss of write monitored data is a condition to commit, a
20 third bit, when set, to indicate any loss of buffered data is a condition to commit,
and a fourth bit, when set, to indicate any loss of metadata is a condition to
commit.
- 25 100. The apparatus of claim 99, wherein the four bits are to be included in the
opcode.
- 30 101. The apparatus of claim 99, wherein the commit logic to determine if the
commit condition to be specified by the commit instruction is satisfied for the
transaction comprises the commit logic to check corresponding status bits in a
transaction status register for each of the four bits that are set in the commit
instruction and to determine the commit condition is satisfied if none of the
corresponding status bits in the transaction register checked are set to indicate an
associated loss.

102. The apparatus of claim 97, wherein the commit instruction is further to specify clear controls to indicate a combination of read monitored data, write monitored data, buffered data, and metadata to clear upon commit, and wherein the
5 commit logic is to clear the specified combination of read monitored data, write monitored data, buffered data, and metadata after committing the transaction in response to determine the commit condition to be specified by the commit instruction is satisfied for the transaction.
- 10 103. A machine readable medium holding program code, which when executed by a machine, causes the machine to perform the operations of:
encountering a commit instruction for a transaction from the program code, the
commit instruction specifying at least one commit failure condition;
determining if the at least one commit failure condition specified by the commit
15 instruction has been detected during a pendency of the transaction;
providing a value to indicate the at least one commit failure condition specified by
the commit instruction has been detected during the pendency of the
transaction in response to determining least one commit failure condition
specified by the commit instruction has been detected during a pendency of
20 the transaction.
104. The machine readable medium of claim 103, wherein the at least one
commit failure condition is selected from a group consisting of a loss of read
monitored data, a loss of write monitored data, a loss of buffered data, and a loss of
25 metadata.
105. The machine readable medium of claim 103, wherein providing a value to
indicate the at least one commit failure condition specified by the commit
instruction has been detected during the pendency of the transaction comprises
30 loading the value to a destination register to indicate the at least one commit failure
condition specified by the commit instruction has been detected during the
pendency of the transaction.

106. The machine readable medium of claim 103, wherein determining if the at least one commit failure condition specified by the commit instruction has been detected during a pendency of the transaction comprises checking a status bit in a transaction status register associated with the at least one commit failure condition; determining the at least one commit failure condition specified by the commit instruction has been detected during a pendency of the transaction in response to the status bit associated with the at least one commit failure condition is set to indicate the at least one commit failure condition has been detected during the pendency of the transaction; determining the at least one commit failure condition specified by the commit instruction has not been detected during a pendency of the transaction in response to the status bit associated with the at least one commit failure condition is reset to indicate the at least one commit failure condition has not been detected during the pendency of the transaction.
107. The machine readable medium of claim 106, further comprising committing the transaction in response to determining the at least one commit failure condition specified by the commit instruction has not been detected during a pendency of the transaction.
108. A method comprising:
encountering a commit instruction within a transaction, the commit instruction including an operation code (opcode) specifying commit failure conditions for the transaction;
determining no commit failure conditions for the transaction specified within the opcode of the commit instruction were detected during a pendency of the transaction; and
committing the transaction in response to determining no commit failure conditions for the transaction specified within the opcode of the commit instruction were detected during the pendency of the transaction.
109. The method of claim 108, wherein the opcode specifying commit failure conditions for the transaction comprises a first bit of the opcode, when set, specifies a loss of read monitored data is a commit failure condition, a second bit

of the opcode, when set, specifies a loss of write monitored data is a commit failure condition, a third bit of the opcode, when set, specifies a loss of buffered data is a commit failure condition, and a fourth bit of the opcode, when set, specifies a loss of metadata is a commit failure condition.

5

110. The method of claim 109, wherein determining no commit failure conditions for the transaction specified within the opcode of the commit instruction were detected during a pendency of the transaction comprises determining a read monitor bit of a transaction status register is not set to indicate no loss of read monitored data in response to the first bit of the opcode being set, determining a write monitor bit of the transaction status register is not set to indicate no loss of write monitored data in response to the second bit of the opcode being set, determining a buffered bit of the transaction status register is not set to indicate no loss of buffered data in response to the third bit of the opcode being set, and determining a metadata bit of the transaction status register is not set to indicate no loss of metadata in response to the fourth bit of the opcode being set.

10

15

20

25

111. The method of claim 109, wherein the opcode is further to specify clear controls, and the opcode specifying clear controls comprises a fifth bit of the opcode, when set, specifies read monitored data is to be cleared upon commit, a sixth bit of the opcode, when set, specifies write monitored data is to be cleared upon commit, a seventh bit of the opcode, when set, specifies buffered is to be cleared upon commit, and an eighth bit of the opcode, when set, specifies metadata is to be cleared upon commit.

30

112. The method of claim 111, committing the transaction comprises clearing read monitored data if the fifth bit is set, clearing write monitored data if the sixth bit is set, clearing buffered data if the seventh bit is set, and clearing metadata if the eighth bit is set.

113. A system comprising:
a memory to hold program code including a commit instruction for a transaction;
the commit instruction to include an operation code (opcode), which is to

specify failure to commit conditions for the transaction and clear control information;

a processor including decode logic to decode the opcode of the commit instruction;
and commit logic to determine if none of the failure to commit condition to
5 be specified in the opcode were detected during a pendency of the
transaction and to commit the transaction in response to the commit logic
determining the failure to commit condition was not detected during the
pendency of the transaction, wherein commit logic to commit the
transaction includes the commit logic to clear transactional information
10 based on the clear control information to be specified in the opcode of the
commit instruction.

114. The system of claim 113, wherein the failure to commit condition is based
on a combination of a loss of read monitored data, a loss of write monitored data, a
15 loss of buffered data, and a loss of metadata.

115. The system of claim 114, wherein the failure to commit condition is
selected from a group consisting of a loss of write monitored data; a loss of read
monitored data or a loss of write monitored data; loss of write monitored data or
20 loss of buffered data; loss of write monitored data or loss of metadata, and loss of
write monitored data, loss of read monitored data, loss of buffered data, or loss of
metadata.

116. The system of claim 113, wherein the opcode to specify clear control
25 information comprises the opcode to specify which of read monitors, write
monitors, buffered coherency, and metadata is to be cleared upon commit, and
wherein commit logic to clear transactional information based on the clear control
information to be specified in the opcode of the commit instruction comprises the
commit logic clearing the read monitors, write monitors, buffered coherency, and
30 metadata which is specified to be cleared in by the opcode.

117. An apparatus comprising:
a storage element to include a transaction enable field (TEF), the TEF when

holding an active value to indicate that an associated transaction is active and enabled and when holding an inactive value to indicate an associated transaction is suspended; and

logic to save a state of at least the TEF in a storage structure in response to a ring level transition event and to restore the state of at least the TEF from the storage structure to the storage element in response to a return event.

5

10

118. The apparatus of claim 117, wherein the ring level transition event includes an event selected from a group consisting of an interrupt, an exception, a system call, a virtual machine enter, and a virtual machine exit.

15

119. The apparatus of claim 117, wherein the return event includes an event selected from a group consisting of an interrupt return (IRET), a system return (SYSRET), a virtual machine (VM) enter, and a virtual machine (VM) exit.

20

120. The apparatus of claim 117, wherein the storage element includes a flags register, and wherein the TEF includes a transaction enable flag.

121. The apparatus of claim 117, wherein the storage structure includes a stack, the logic to save the state of least the TEF in the stack includes push logic to push the state of least the TEF on the stack, and the logic to restore the state of at least the TEF from the stack to the storage element comprises pop logic to pop the state of at least the TEF off the stack and restore the TEF in the storage element.

25

122. A system comprising:
a memory to hold code, when executed, to cause a ring level transition event; and
a processor comprising a register to include a transaction enable field (TEF) to hold an active value to indicate that an associated transaction is active; and stack logic to push a previous state of the register onto a stack in response to the ring level transition event, clear the TEF to an inactive value to indicate that the associated transaction is suspended, and to restore the previous state of the register from the stack to the register in response to a return event.

30

123. The system of claim 122, wherein the ring level transition event includes an event selected from a group consisting of an interrupt, an exception, a system call, a virtual machine enter, and a virtual machine exit.
- 5
124. The system of claim 122, wherein the return event includes an event selected from a group consisting of an interrupt return (IRET), a system return (SYSRET), a virtual machine (VM) enter, and a virtual machine (VM) exit.
- 10
125. The system of claim 122, wherein the register includes a flags register, the TEF includes a transaction enable flag, the active value includes a high logical value of the flag, and the inactive value includes a low logical value of the flag.
126. A method comprising:
- 15
- detecting a ring level transition event from a current ring level;
- saving a previous state of a register including a transaction enable field in a storage structure;
- clearing the transaction enable field to indicate an associated transaction is suspended;
- 20
- detecting a return to the current ring level event;
- restoring the previous state of the register from the storage structure in response to detecting the return to the current ring level event.
127. The method of claim 126, wherein the storage structure includes a kernel stack, saving the previous state of the register in the kernel stack includes pushing the previous state of the register onto the kernel stack, and restoring the previous state of the register from the kernel stack includes popping the previous state of the register from the kernel stack and restoring the previous state to the register.
- 25
128. The method of claim 126, wherein the current ring level includes a user ring level.
- 30
129. The method of claim 128, wherein the ring level transition event includes

an event selected from a group consisting of an interrupt, an exception, a system call, and a virtual machine enter.

- 5 130. The method of claim 129, wherein the return to the current privilege level event includes an event selected from a group consisting of an interrupt return (IRET), a system return (SYSRET), and a virtual machine (VM) exit.

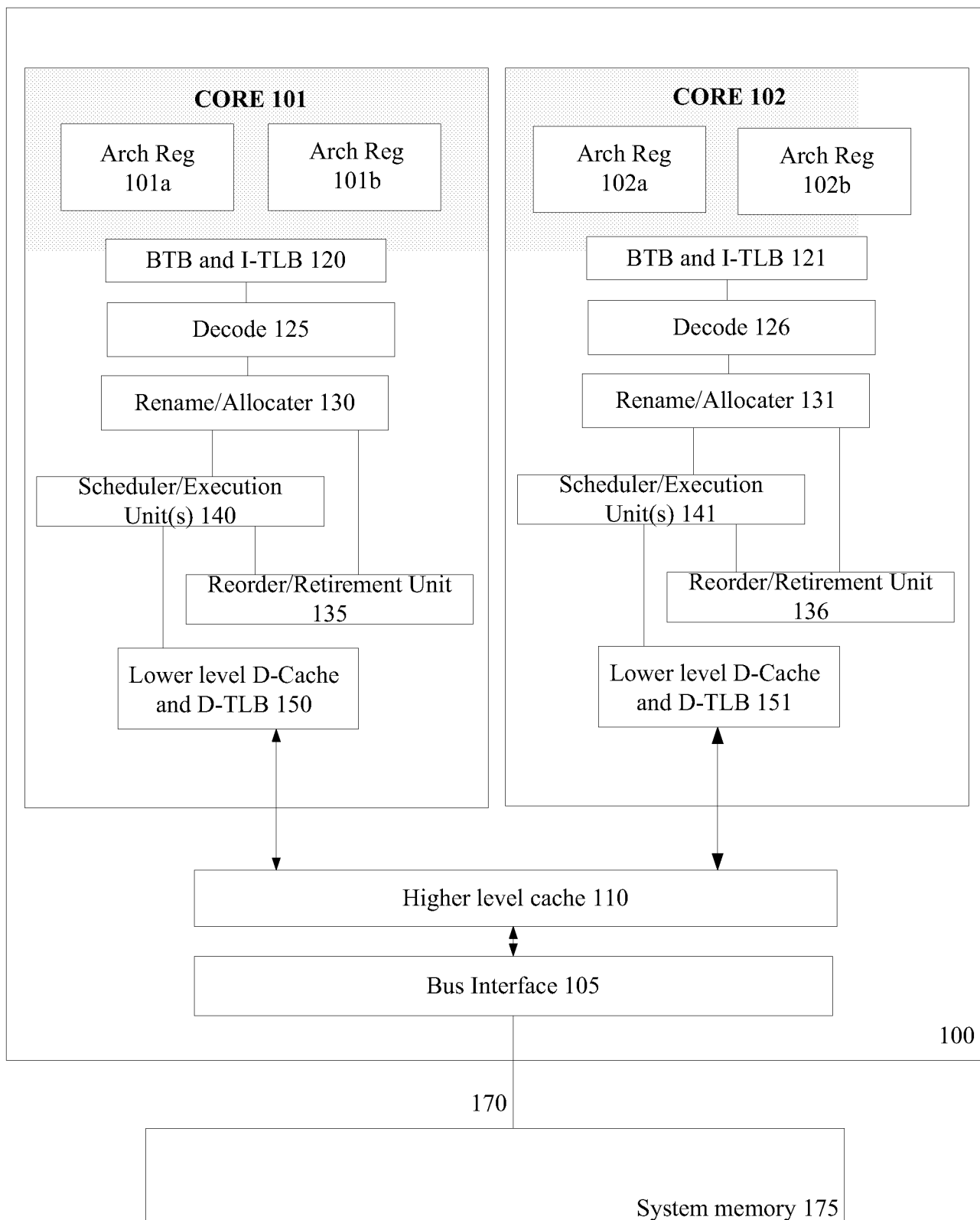


FIG. 1

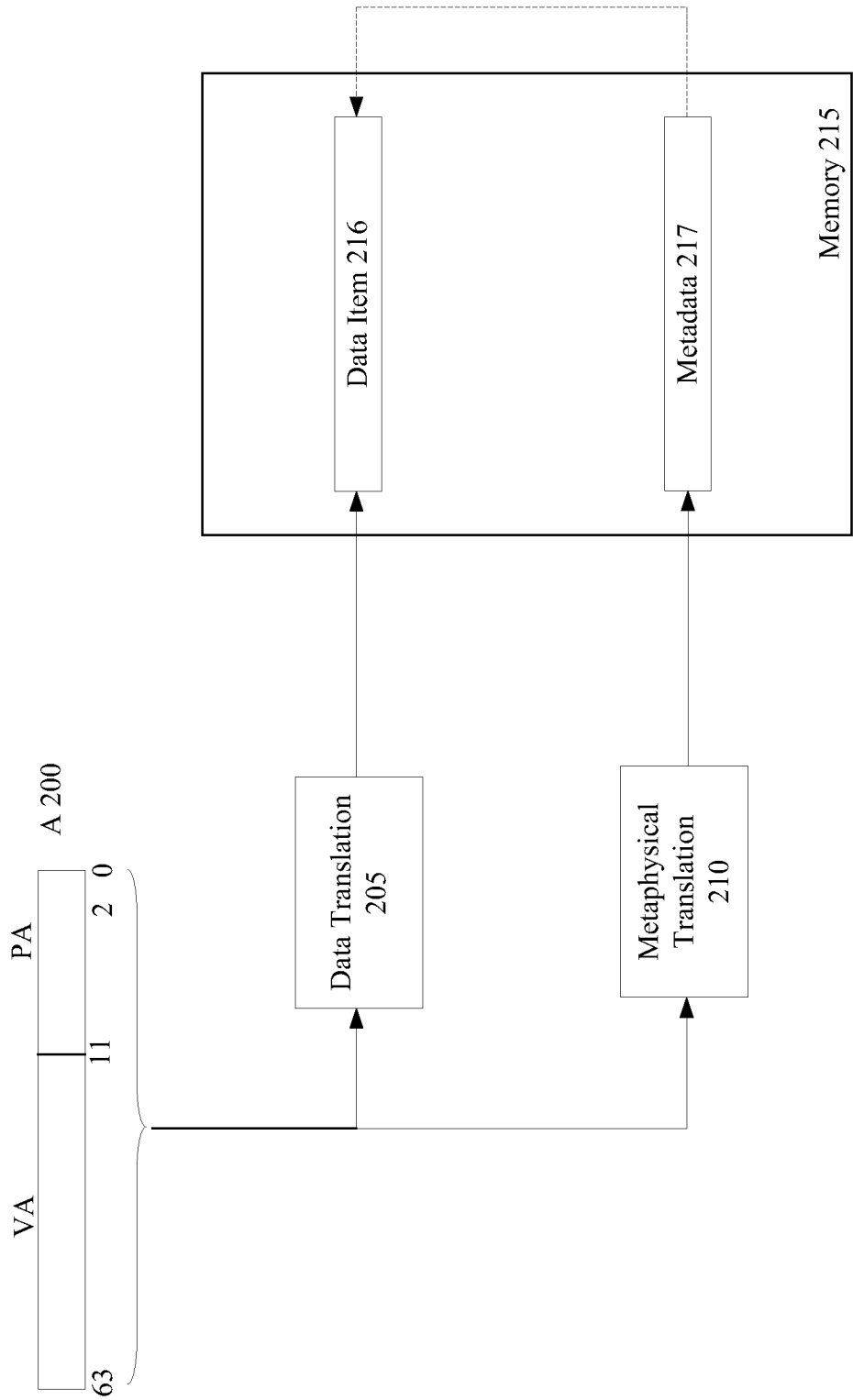


FIG. 2

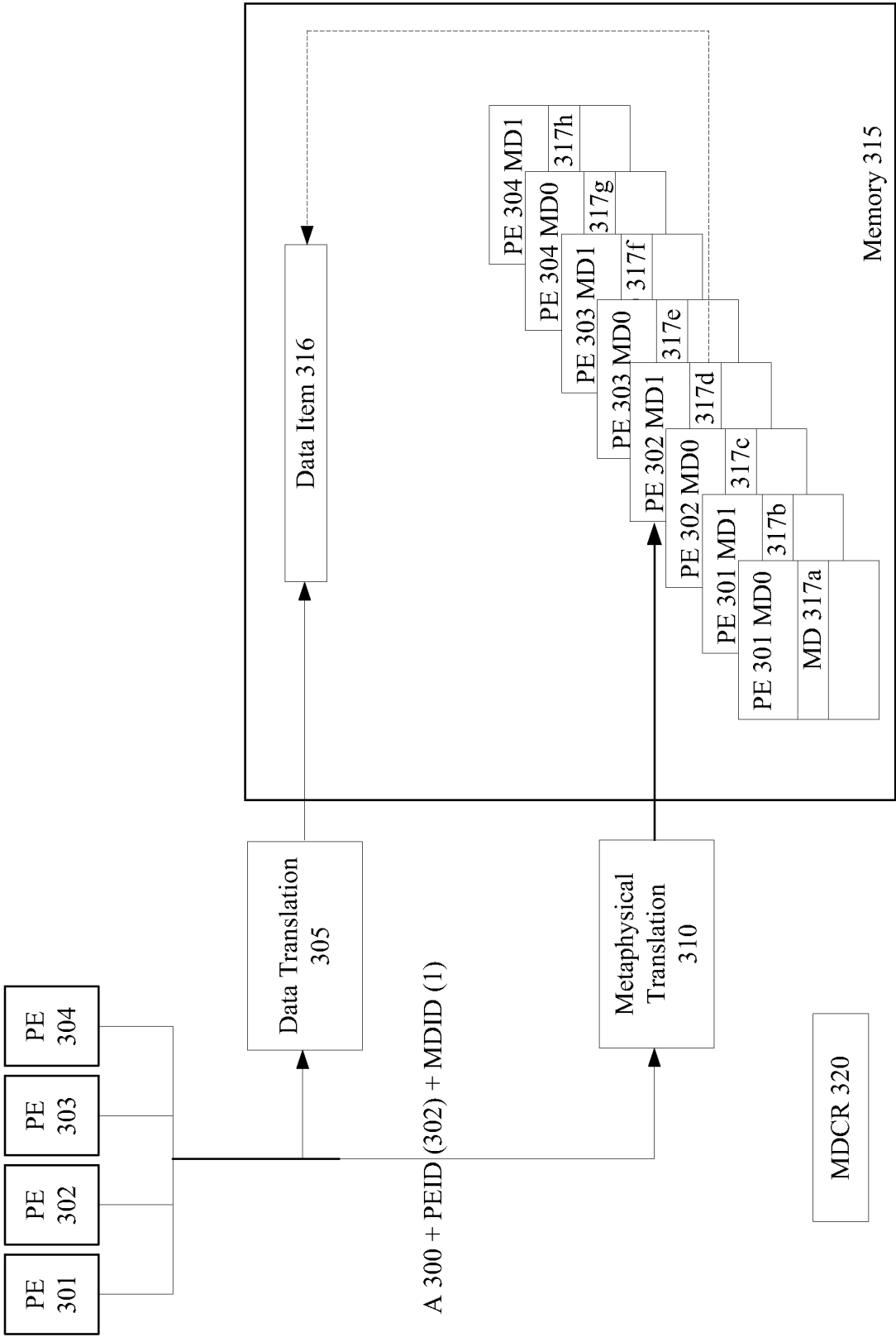


FIG. 3

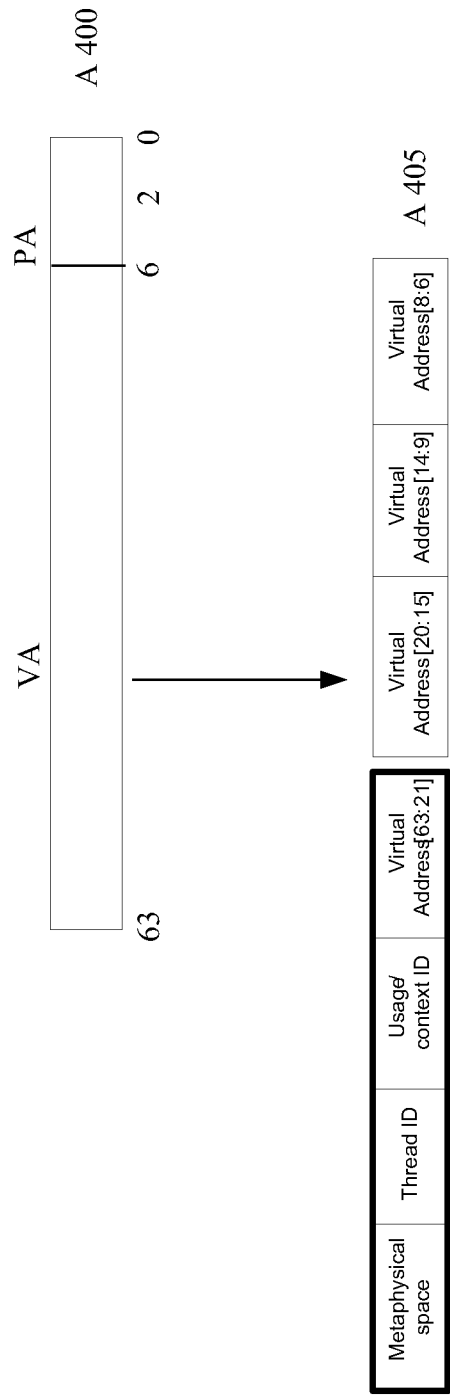
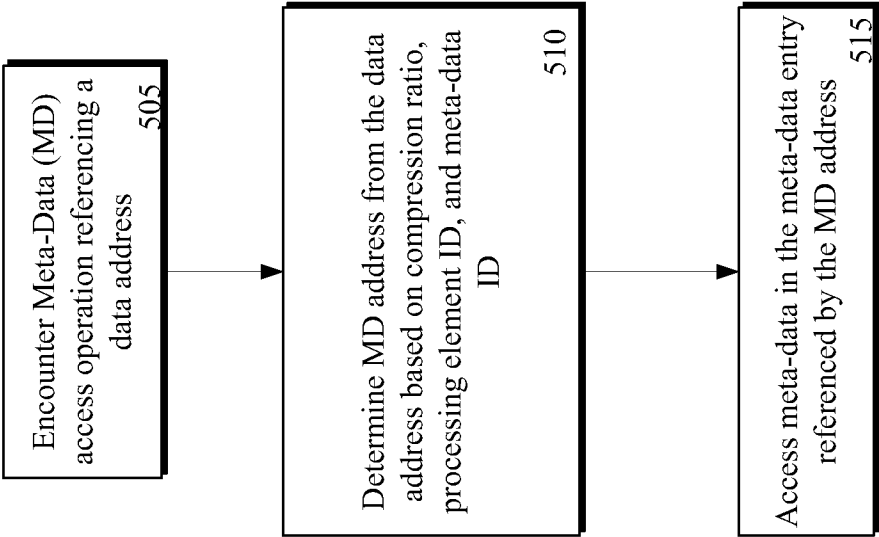


FIG. 4



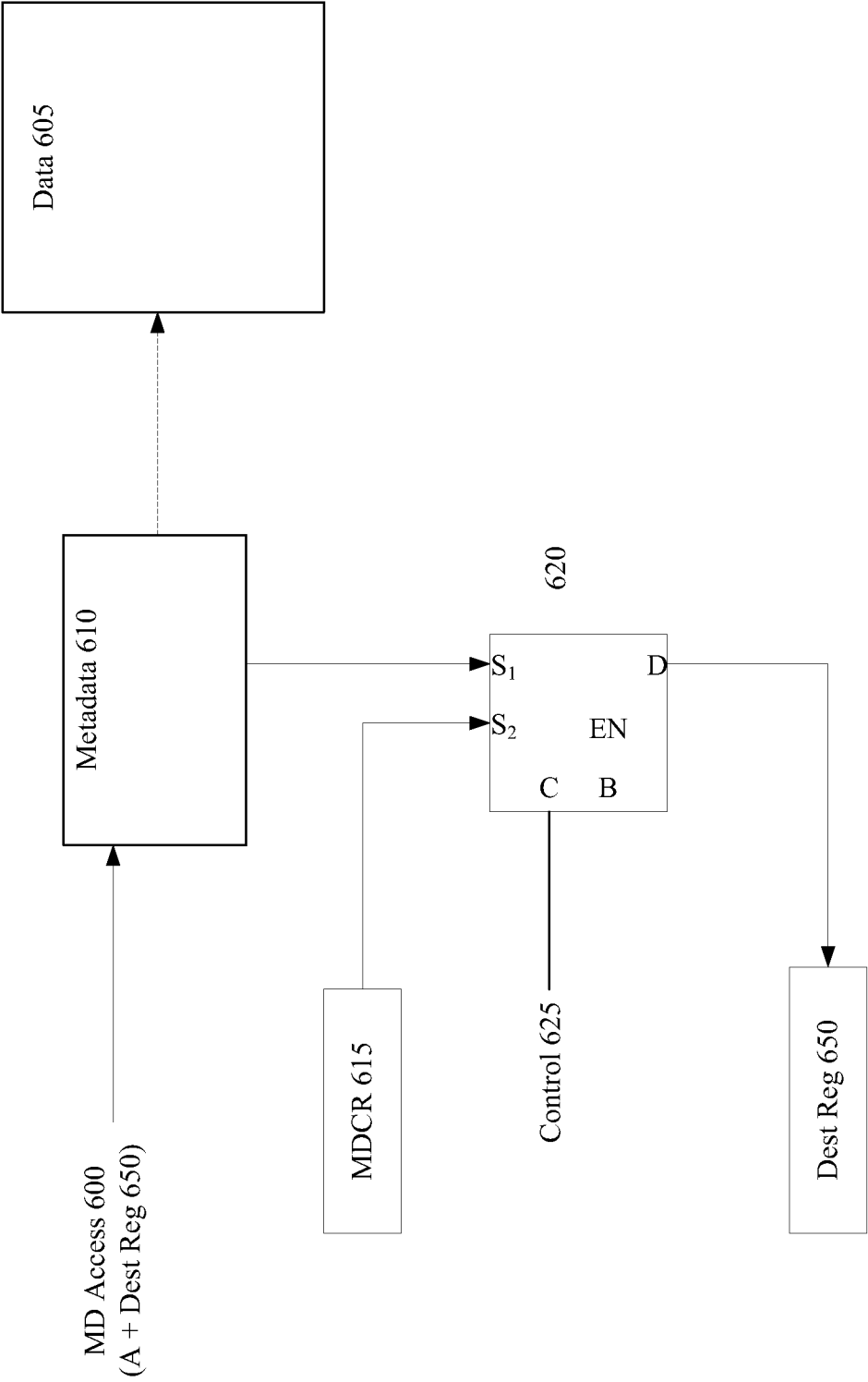
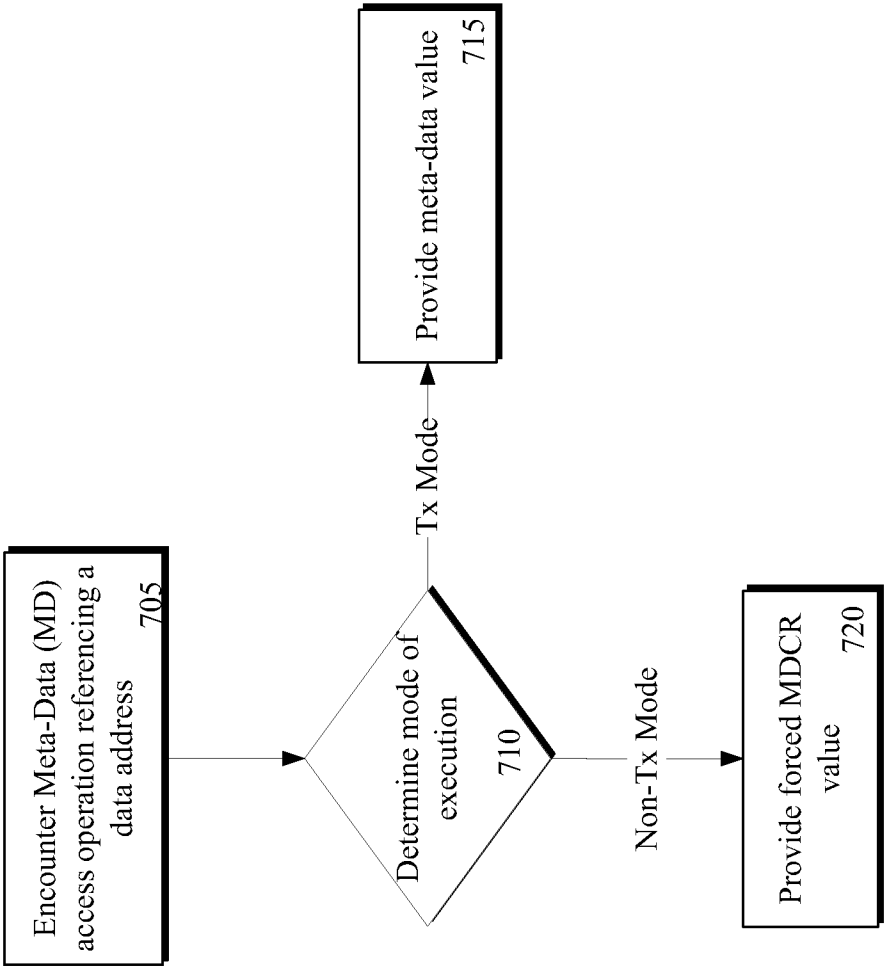
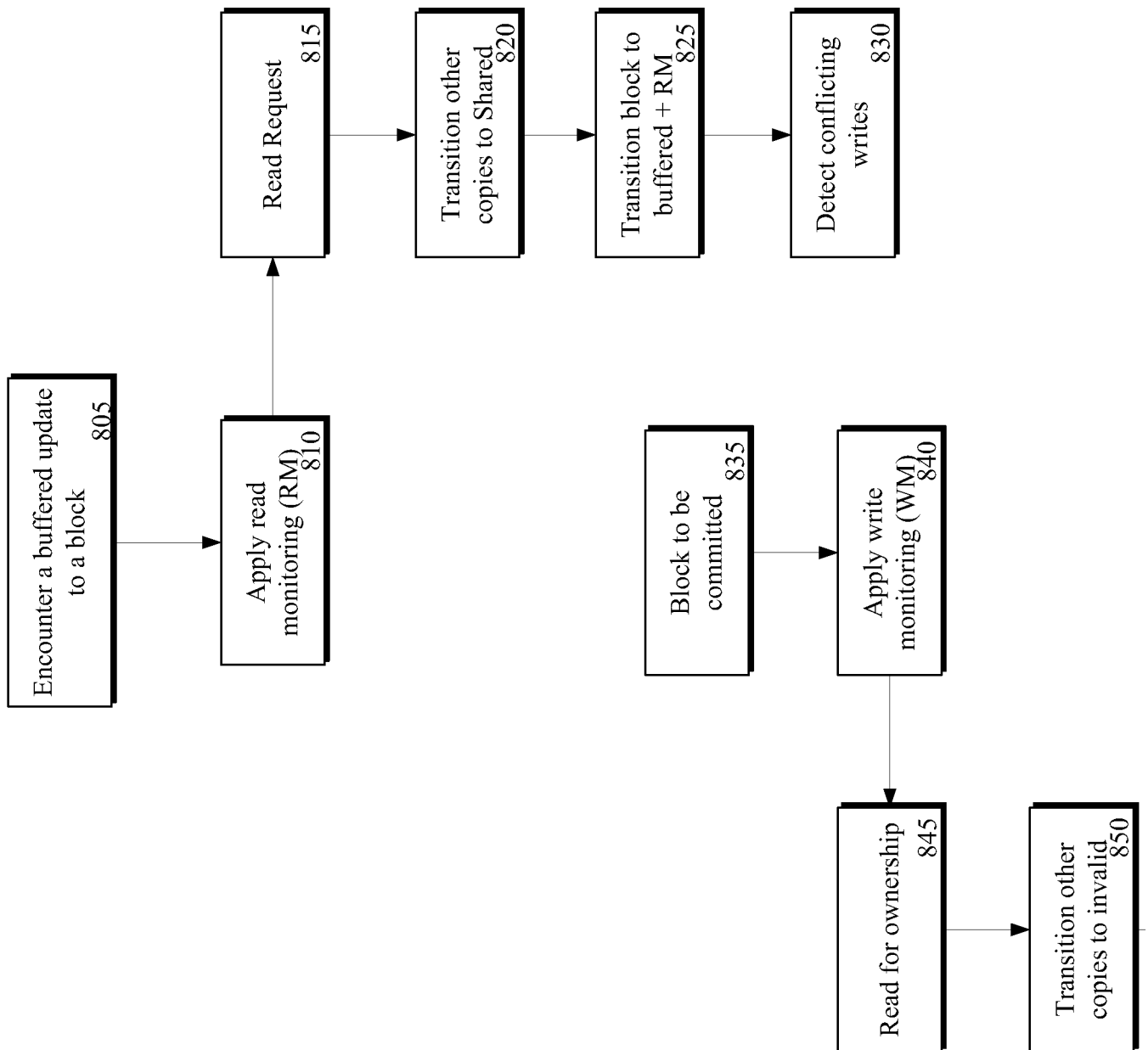
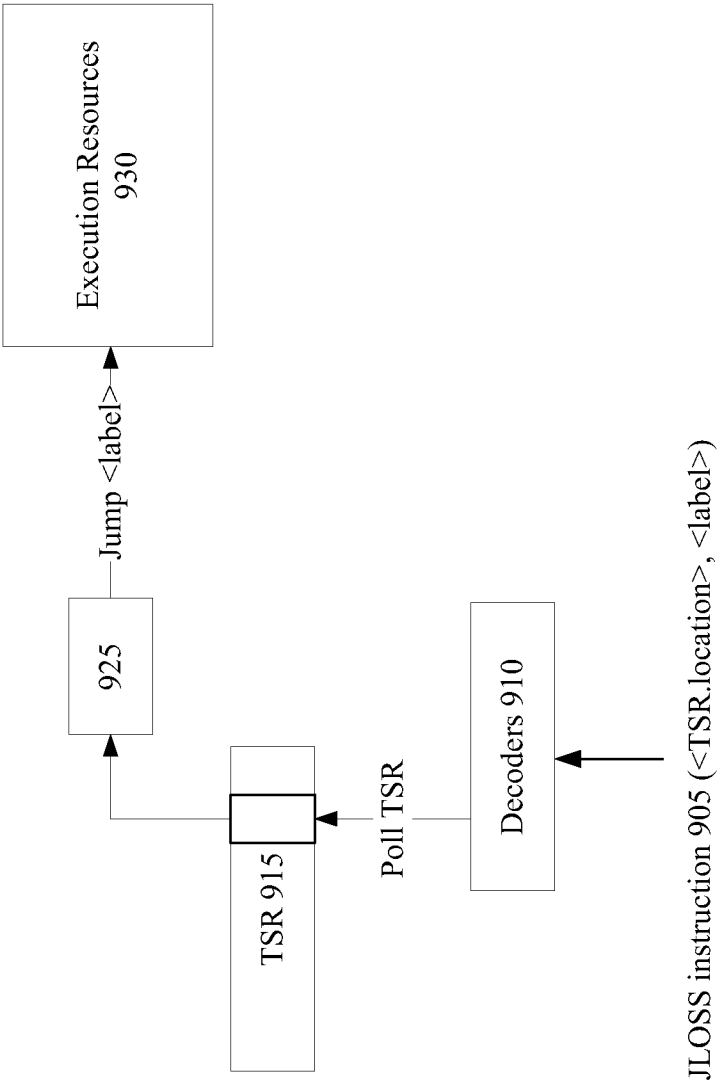
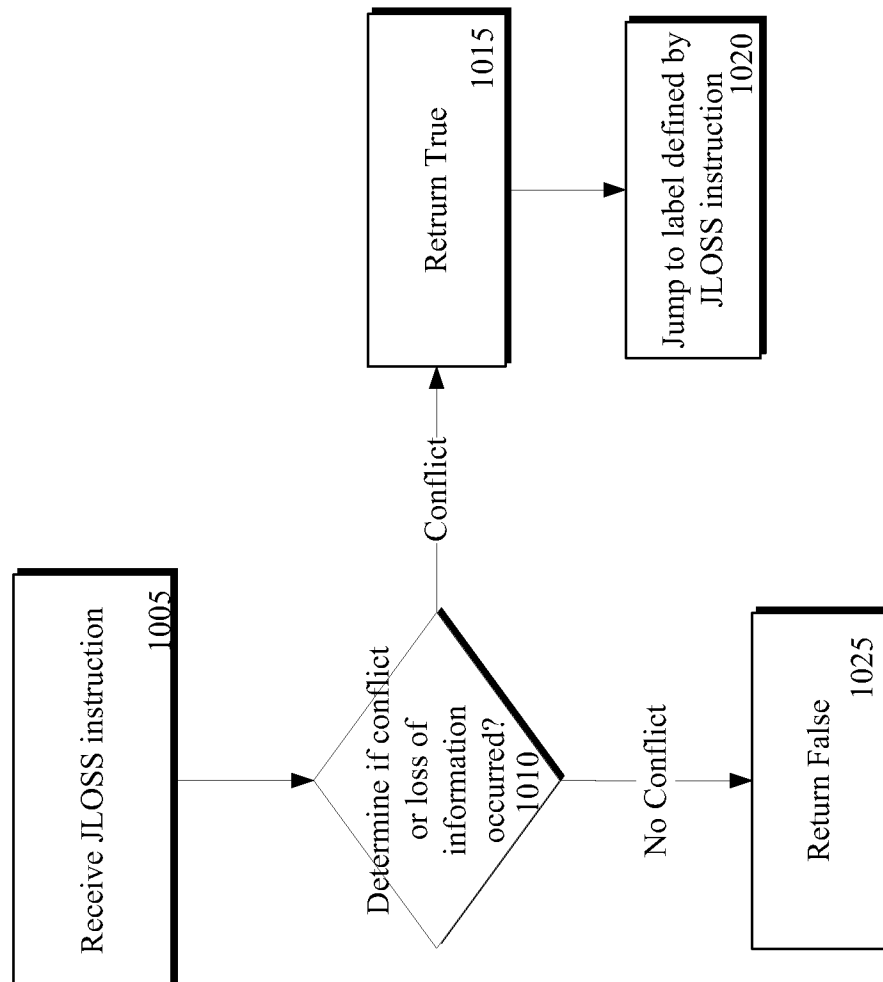


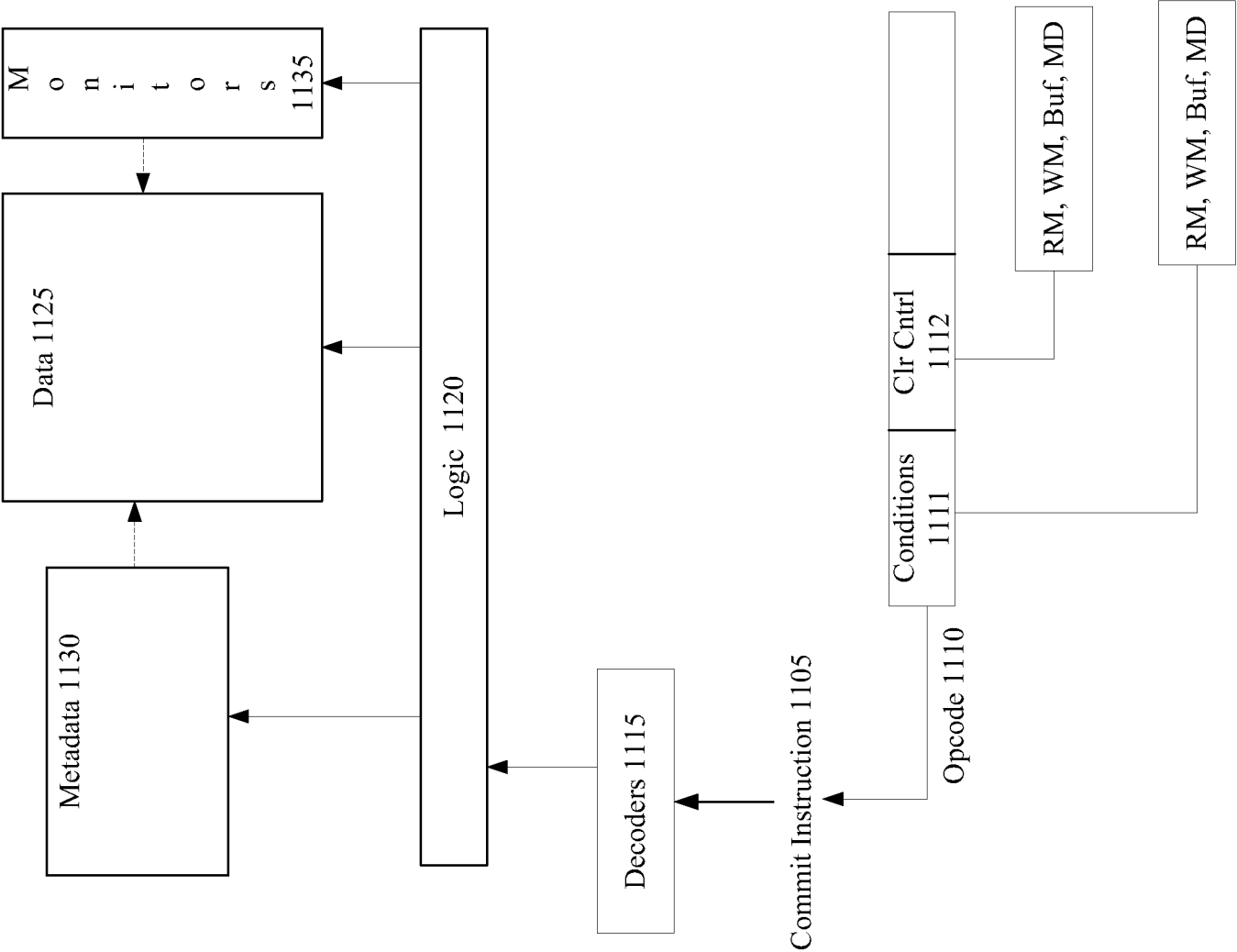
FIG. 6

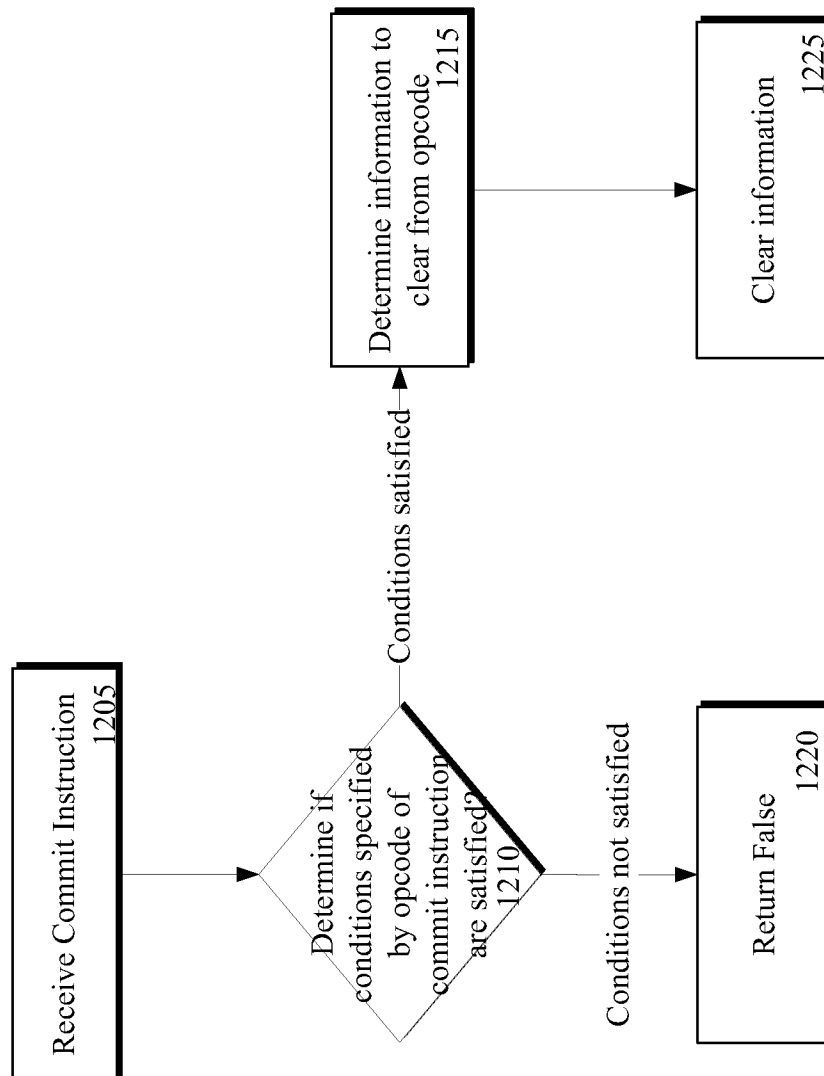


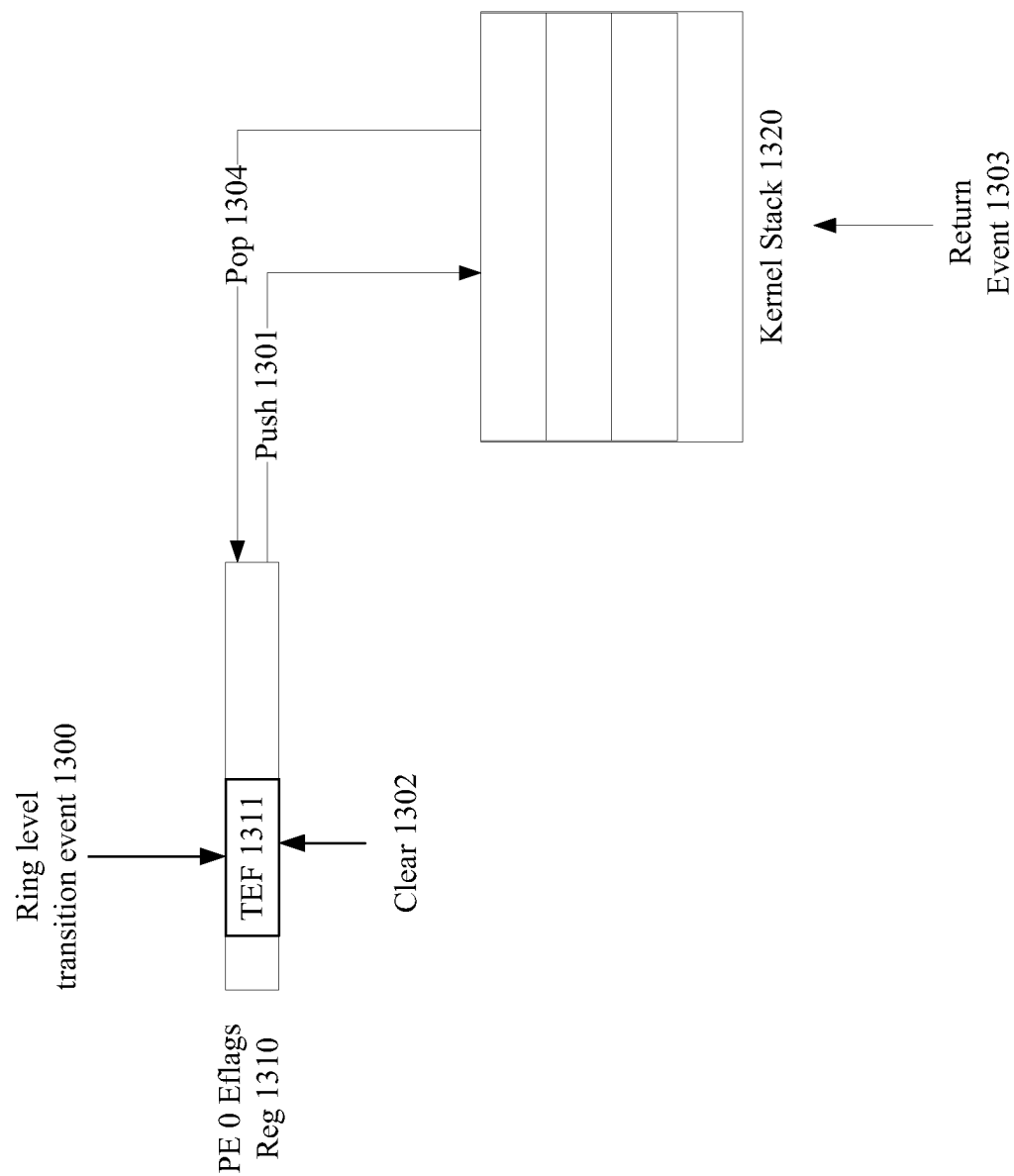












INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2009/048947**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/06(2006.01)i, G06F 9/44(2006.01)i, G06F 9/46(2006.01)i, G06F 15/00(2006.01)i, G06F 13/00(2006.01)i**

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC 8: G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean Utility models applications for Utility Models since 1975, IPC as above

Japanese Utility Models and application for Utility Models since 1975, IPC as above

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

eKOMPASS, IEEEExpl, Google; transactional memory, metadata, metaphysical;

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2008-0040511 A1 (JAN GRAY., et al.) 14 Feb. 2008 See the abstract, claims 1-20 and figures 1-35.	1-130
A	US 2007-0156994 A1 (HAITHAM H. AKKARY., et al.) 5 Jul. 2007 See the abstract, claims 1-30 and figures 1-7.	1-130
A	US 2007-0198519 A1 (DAVID DICE., et al.) 23 Aug. 2007 See the abstract, claims 1-20 and figures 1-8.	1-130

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

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

24 FEBRUARY 2010 (24.02.2010)

Date of mailing of the international search report

25 FEBRUARY 2010 (25.02.2010)

Name and mailing address of the ISA/KR

Korean Intellectual Property Office
Government Complex-Daejeon, 139 Seonsa-ro, Seo-
gu, Daejeon 302-701, Republic of Korea

Facsimile No. 82-42-472-7140

Authorized officer

BOK, Jin Yo

Telephone No. 82-42-481-5113



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2009/048947

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008-0040551 A1	14.02.2008	AU 2006-322227 A1	14.06.2007
		CN 101542437 A	23.09.2009
		EP 1958063 A2	20.08.2008
		JP 2009-523271 A	18.06.2009
		KR 10-2008-0071135 A	01.08.2008
		US 2007-0169030 A1	19.07.2007
		US 2007-0245309 A1	18.10.2007
		WO 2007-067390 A3	14.06.2007
		WO 2008-154193 A3	18.12.2008
US 2007-0156994 A1	05.07.2007	CN 101322103 A	10.12.2008
		DE 112006003597 T5	13.11.2008
		GB 2447575 A	17.09.2008
		JP 2009-521767 A	04.06.2009
		KR 10-2008-0076981 A	20.08.2008
		WO 2007-078883 A1	12.07.2007
US 2007-0198519 A1	23.08.2007	US 2007-0239943 A1	11.10.2007
		US 07640402 B2	29.12.2009