US 20070198978A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2007/0198978 A1**

Dice et al. (43) **Pub. Date: Aug. 23, 2007**

(54) **METHODS AND APPARATUS TO IMPLEMENT PARALLEL TRANSACTIONS**

(76) Inventors: **David Dice**, Foxborough, MA (US); **Nir N. Shavit**, Cambridge, MA (US)

Correspondence Address:
BARRY W. CHAPIN, ESQ.
CHAPIN INTELLECTUAL PROPERTY LAW, LLC
WESTBOROUGH OFFICE PARK
1700 WEST PARK DRIVE
WESTBOROUGH, MA 01581 (US)

(21) Appl. No.: **11/475,604**

(22) Filed: **Jun. 27, 2006**

Related U.S. Application Data

(60) Provisional application No. 60/775,564, filed on Feb. 22, 2006. Provisional application No. 60/775,580, filed on Feb. 22, 2006. Provisional application No. 60/789,483, filed on Apr. 5, 2006.

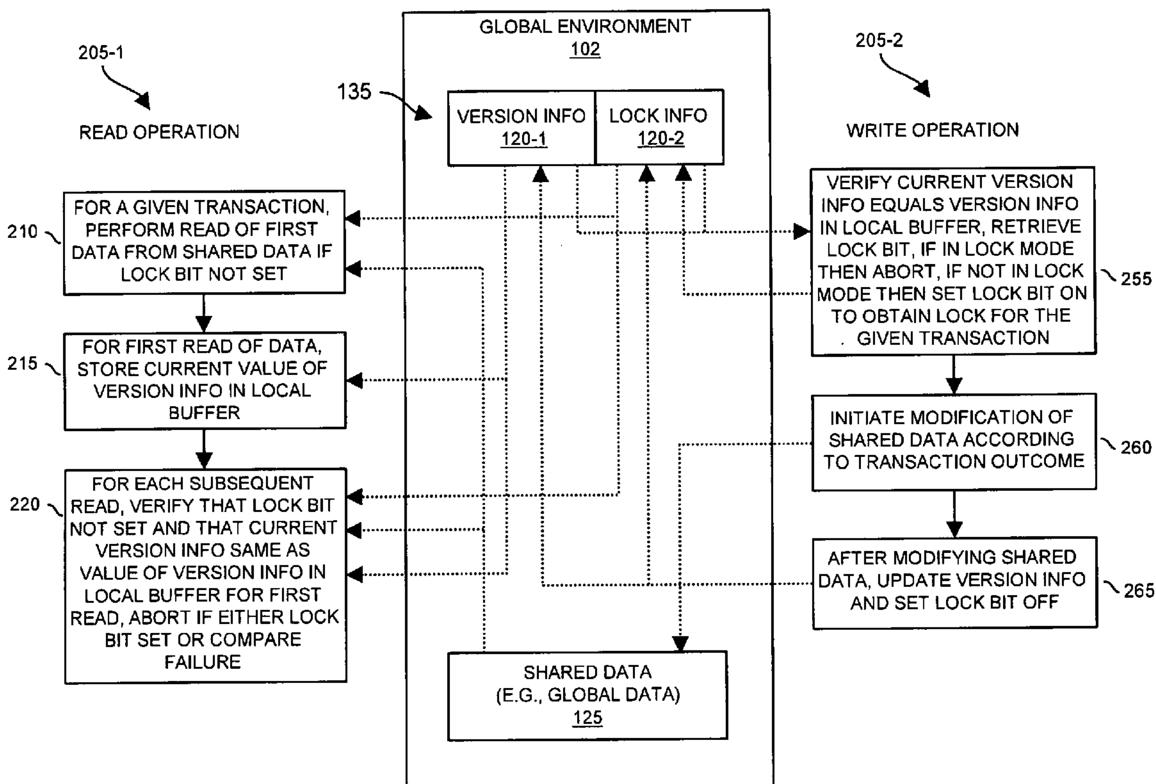
Publication Classification

(51) **Int. Cl.**
G06F 9/46 (2006.01)

(52) **U.S. Cl.** **718/100**

(57) **ABSTRACT**

A computer system includes multiple processing threads that execute in parallel. The multiple processing threads have access to a global environment including i) shared data utilized by the multiple processing threads, ii) a globally accessible register or buffer of version information that changes each time a respective one of the multiple processing threads modifies the shared data, and iii) respective lock information indicating whether one of the multiple processing threads has locked the shared data preventing other processing threads from modifying the shared data. To prevent data corruption, each of the processing threads aborts if a given processing thread detects a change in the version information or another processing thread has a lock on the shared data. This technique is well suited for use in applications such as processing threads that support a high number of reads with a corresponding number of fewer respective writes to shared data.



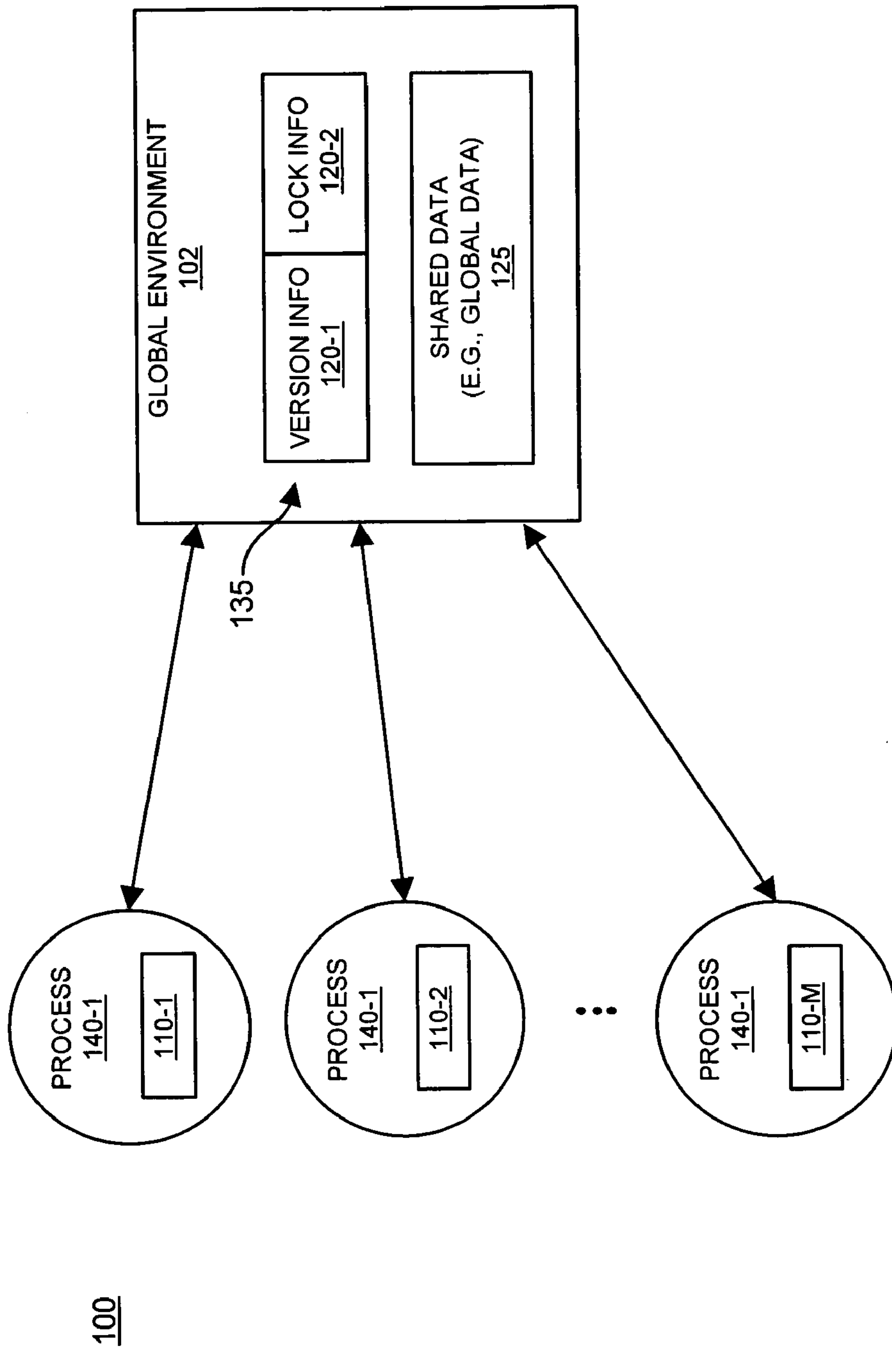


FIG. 1

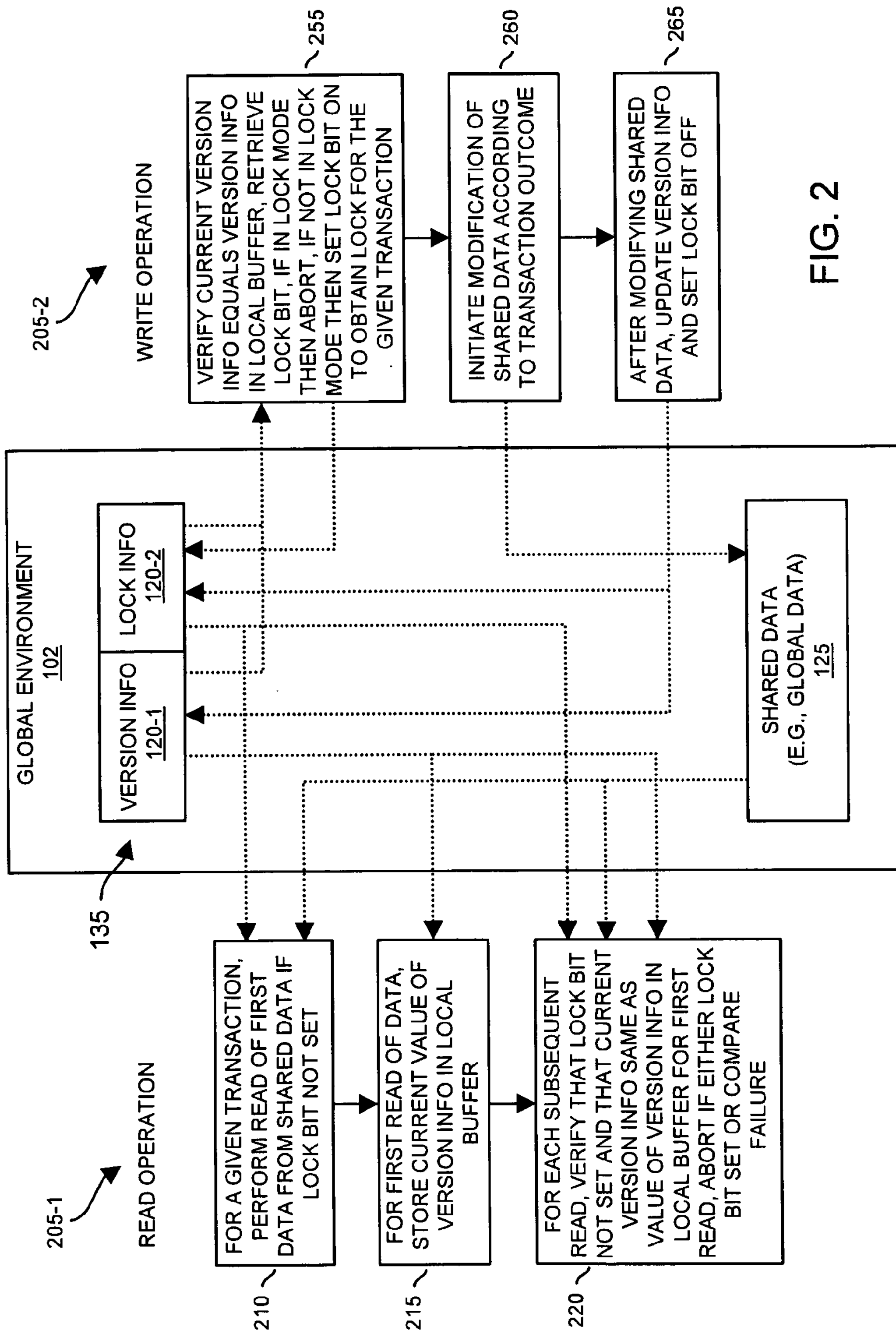


FIG. 2

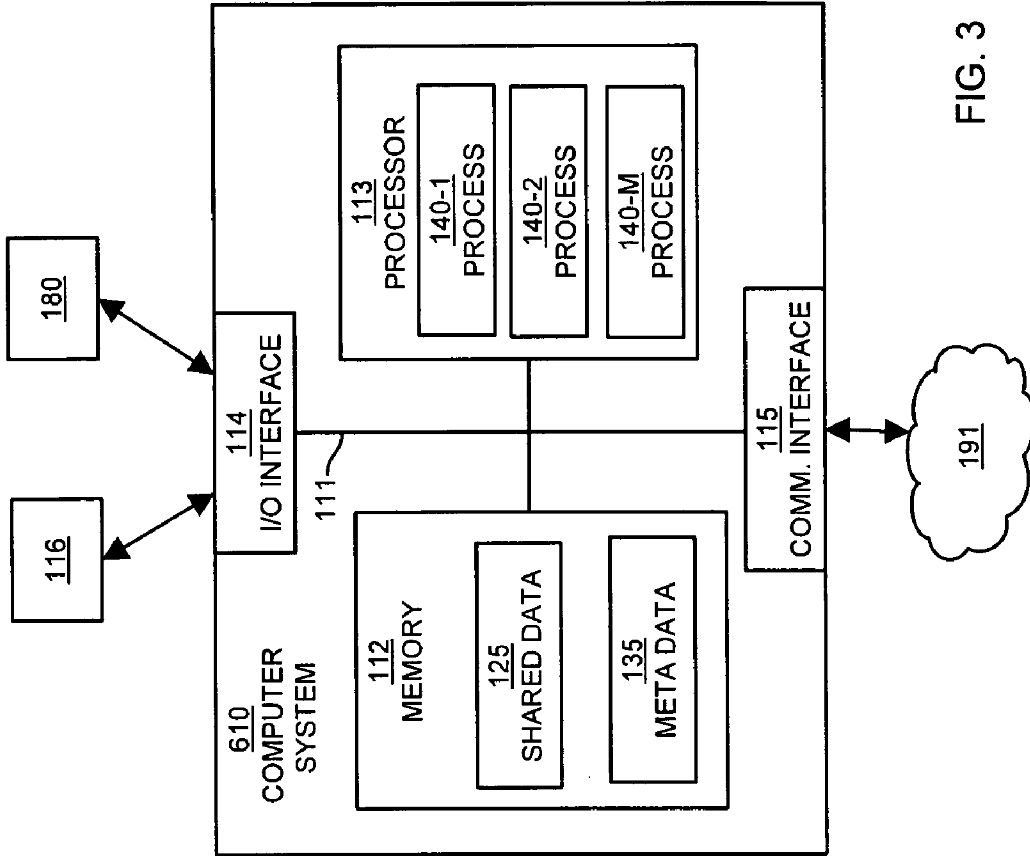


FIG. 3

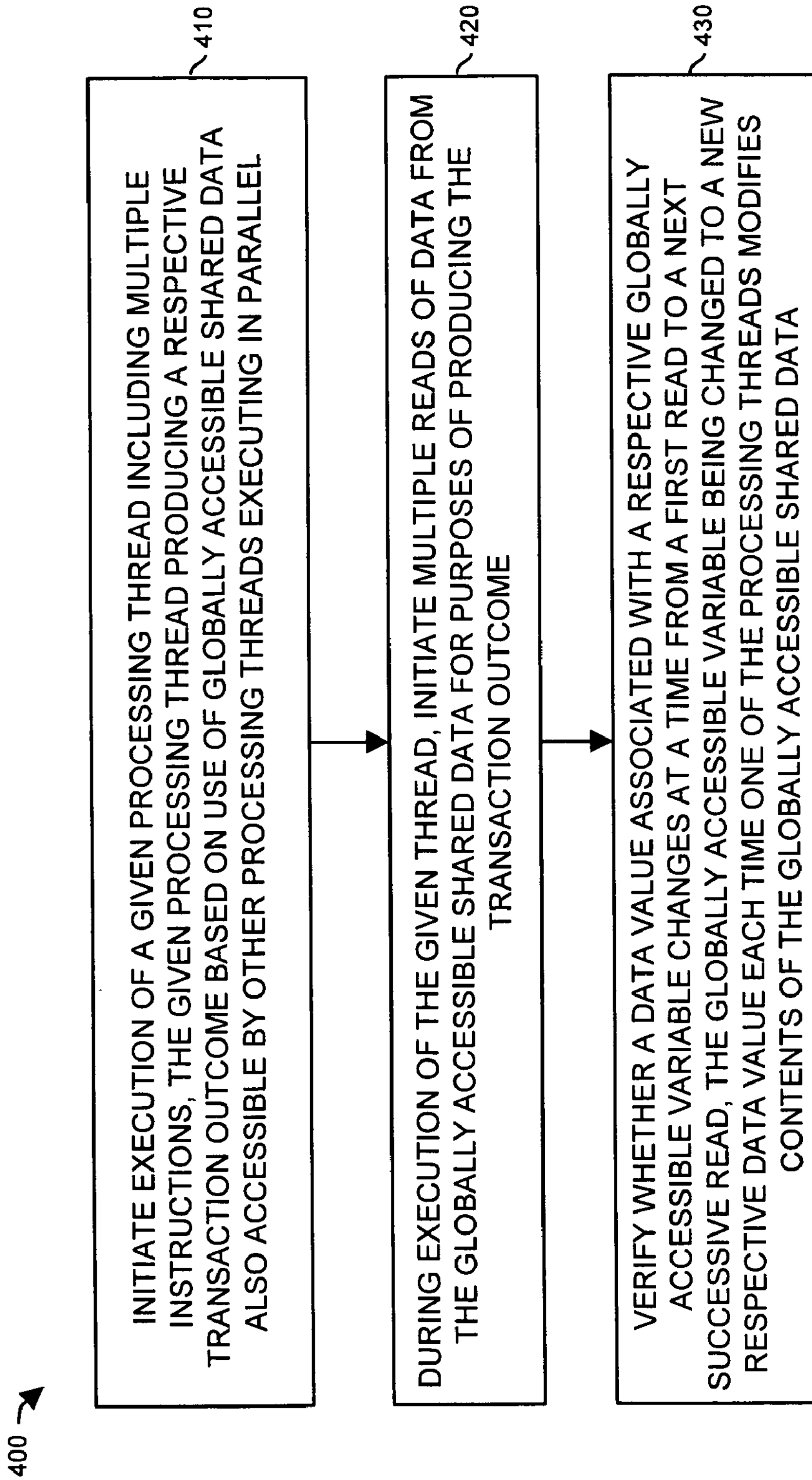


FIG. 4

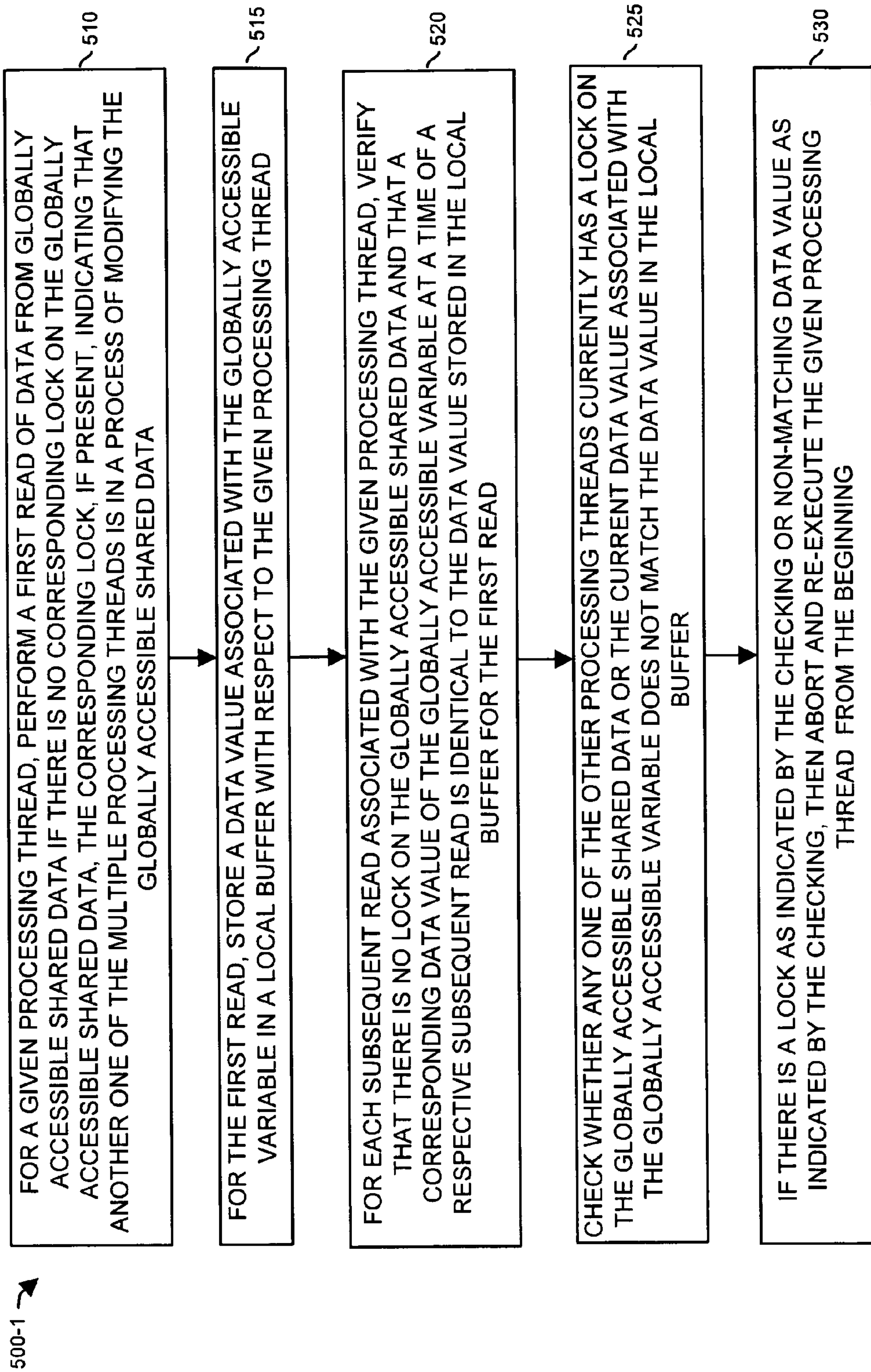


FIG. 5

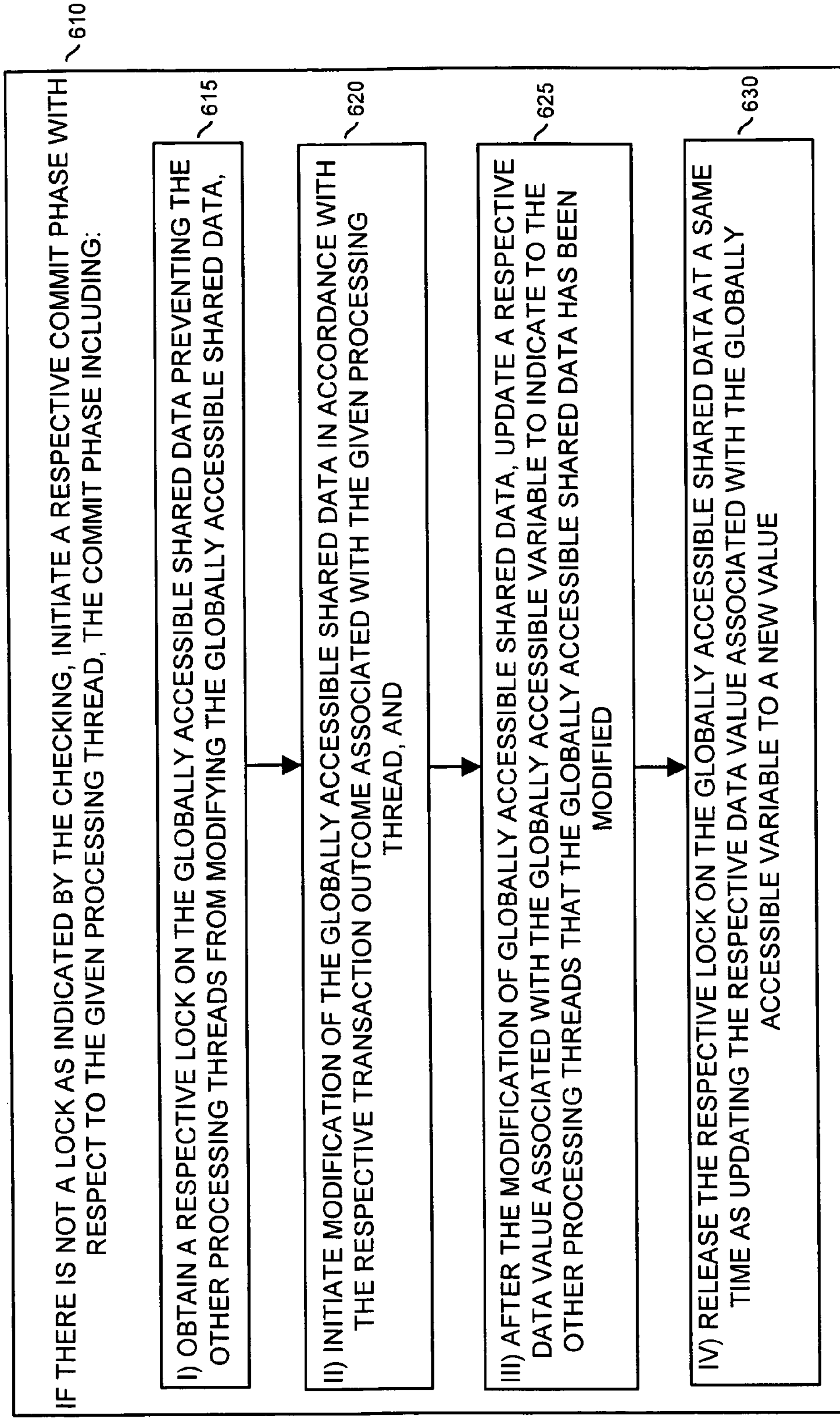


FIG. 6

METHODS AND APPARATUS TO IMPLEMENT PARALLEL TRANSACTIONS

RELATED APPLICATIONS

[0001] This application claims the benefit of and priority to U.S. Provisional Patent Application Ser. No. 60/775,564 (Attorney's docket no. SUN06-01(060711)p, filed on Feb. 22, 2006, entitled "Switching Between Read-Write Locks and Transactional Locking," the entire teachings of which are incorporated herein by this reference.

[0002] This application is related to U.S. Patent Application identified by Attorney's docket no. SUN06-04(060720), filed on same date as the present application, entitled "METHODS AND APPARATUS TO IMPLEMENT PARALLEL TRANSACTIONS," which itself claims the benefit of and priority to U.S. Provisional Patent Application Ser. No. 60/775,580 (Attorney's docket no. SUN06-02p(060720), filed on Feb. 22, 2006, entitled "Transactional Locking," the entire teachings of which are incorporated herein by this reference.

[0003] This application is related to U.S. Patent Application identified by Attorney's docket no. SUN06-06(060908), filed on same date as the present application, entitled "METHODS AND APPARATUS TO IMPLEMENT PARALLEL TRANSACTIONS," which itself claims the benefit of and priority to U.S. Provisional Patent Application Ser. No. 60/789,843 (Attorney's docket no. SUN06-05(060908)p, filed on Apr. 5, 2006, entitled "Globally Versioned Transactional Locking," the entire teachings of which are incorporated herein by this reference.

[0004] This application is related to U.S. Patent Application identified by Attorney's docket no. SUN06-08(061191), filed on same date as the present application, entitled "METHODS AND APPARATUS TO IMPLEMENT PARALLEL TRANSACTIONS," which itself claims the benefit of and priority to U.S. Provisional Patent Application Ser. No. 60/775,564 (Attorney's docket no. SUN06-01(060711)p, filed on Feb. 22, 2006, entitled "Switching Between Read-Write Locks and Transactional Locking," the entire teachings of which are incorporated herein by this reference.

BACKGROUND

[0005] There has been an ongoing trend in the information technology industry to execute software programs more quickly. For example, there are various conventional advancements that provide for increased execution speed of software programs.

[0006] One technique for increasing execution speed of a program is called parallelism. Parallelism is the practice of executing or performing multiple things simultaneously. Parallelism can be possible on multiple levels, from executing multiple instructions at the same time, to executing multiple threads at the same time, to executing multiple programs at the same time, and so on. Instruction Level Parallelism or ILP is parallelism at the lowest level and involves executing multiple instructions simultaneously. Processors that exploit ILP are typically called multiple-issue processors, meaning they can issue multiple instructions in a single clock cycle to the various functional units on the processor chip.

[0007] There are different types of conventional multiple-issue processors. One type of multiple-issue processor is a superscalar processor in which a sequential list of program instructions are dynamically scheduled. A respective processor determines which instructions can be executed on the same clock cycle, and sends them out to their respective functional units to be executed. This type of multi-issue processor is called an in-order-issue processor since issuance of instructions is performed in the same sequential order as the program sequence, but issued instructions may complete at different times (e.g., short instructions requiring fewer cycles may complete before longer ones requiring more cycles).

[0008] Another type of multi-issue processor is called a VLIW (Very Large Instruction Width) processor. A VLIW processor depends on a compiler to do all the work of instruction reordering and the processor executes the instructions that the compiler provides as fast as possible according to the compiler-determined order. Other types of multi-issue processors issue out of order instructions, meaning the instruction issue order is not be the same order as the order of instructions as they appear in the program.

[0009] Conventional techniques for executing instructions using ILP can utilize look-ahead techniques to find a larger amount of instructions that can execute in parallel within an instruction window. Looking-ahead often involves determining which instructions might depend upon others during execution for such things as shared variables, shared memory, interference conditions, and the like. When scheduling, a handler associated with the processor detects a group of instructions that do not interfere or depend on each other. The processor can then issue execution of these instructions in parallel thus conserving processor cycles and resulting in faster execution of the program.

[0010] One type of conventional parallel processing involves a use of coarse-grained locking. As its name suggests, coarse-grained locking prevents conflicting groups of code from operating on different processes at the same time based on use of lockouts. Accordingly, this technique enables non-conflicting transactions or sets of instructions to execute in parallel.

[0011] Another type of conventional parallel processing involves a use of fine-grain locking. As its name suggests, fine-grain locking prevents conflicting instructions from being simultaneously executed in parallel based on use of lockouts. This technique therefore enables non-conflicting instructions to execute in parallel.

SUMMARY

[0012] Conventional applications that support parallel processing can suffer from a number of deficiencies. For example, although easy to implement from the perspective of a software developer, coarse-grained locking techniques provide very poor performance because of limitations on parallelism. Although fine-grain lock-based concurrent software can perform exceptionally well during run-time, developing such code can be a very difficult task for a respective one or more software developers.

[0013] Techniques discussed herein deviate with respect to conventional applications such as those discussed above as well as other techniques known in the prior art. For

example, embodiments herein include novel techniques for enhancing performance associated with transactions executing in parallel.

[0014] In general, a technique according to embodiments herein provides a unique way for each of multiple processes to operate in parallel using (e.g., based on reading, modifying, and writing to) a same set of shared data without causing corruption to the shared data. For example, according to one embodiment herein, a computer system includes multiple processing threads that execute in parallel. The multiple processing threads have access to a global environment including i) shared data utilized by the multiple processing threads, ii) a globally accessible register or buffer of version information that changes each time a respective one of the multiple processing threads modifies the shared data, and iii) respective lock information indicating whether one of the multiple processing threads has locked the shared data preventing other processing threads from modifying the shared data.

[0015] Each of the multiple processing threads accesses the version information (associated with the globally accessible shared data) and/or respective lock information at multiple times during execution of a respective processing thread to determine whether to abort execution of the respective processing thread as a result of another processing thread modifying the shared data. For example, if a given processing thread identifies (during a course of executing each of multiple successive reads) that another one of multiple processing threads modified the shared data or is currently modifying the shared data, the given processing thread aborts further operations and re-executes from the beginning again.

[0016] One way that respective processing threads notify each other of changes to the shared data is to update the respective version information associated with the shared data. For example, a respective processing thread can increment the version information after modifying contents of the shared data. If the given processing thread identifies that the version information is a first value at a respective time of a first read of shared data and a second value at a time of a second or subsequent read of shared data, then the given processing thread can immediately abort to end a respective transaction of multiple instructions.

[0017] The lock information associated with the shared data also can be used to identify whether to abort a respective transaction. In one embodiment, the lock information is a single lock bit value set and reset by the respective processing threads. Only one processing thread can obtain a lock over the shared data at a time.

[0018] Accordingly, each of the multiple processing threads is initially optimistic that the corresponding processing thread will be able to complete a respective transaction prior to having to self-abort as a result of another processing thread contemporaneously modifying the shared data. If another processing thread modifies the shared data during the execution of a given processing thread, it can't be certain that the given processing thread and its results have not been corrupted. Self-aborting of a processing thread thus prevents potential corruption.

[0019] Techniques herein are well suited for use in applications such as processing threads that support a high number of reads and fewer respective writes to shared data because, in such an instance, one processing thread will rarely or less frequently (if at all) modify contents of the shared data and cause abortion of other processing threads. However, it should be noted that configurations herein are not limited to such use and thus configurations herein and deviations thereof are well suited for use in other environments as well.

[0020] In addition to the embodiments discussed above, other embodiments herein include a computerized device (e.g., a host computer, workstation, etc.) configured to support the techniques disclosed herein such as use of a globally accessible version information enabling parallel execution of transaction performed by different processes. In such embodiments, a computer environment can include a memory system, a processor (e.g., a processing device), a respective display, and an interconnect connecting the processor and the memory system. The interconnect can also support communications with the respective display (e.g., display screen or display medium). The memory system can be encoded with one or more applications that, when executed on a respective processor, supports parallel processing according to techniques herein.

[0021] Yet other embodiments of the present disclosure include software programs to perform the method embodiment and operations summarized above and disclosed in detail below in the Detailed Description section of this disclosure. More specifically, one embodiment herein includes a computer program product (e.g., a computer-readable medium). The computer program product includes computer program logic (e.g., software instructions) encoded thereon. Such computer instructions can be executed on a computerized device to support parallel processing according to embodiments herein.

[0022] For example, the computer program logic, when executed on at least one processor associated with a computing system, causes the processor to perform the operations (e.g., the methods) indicated herein as embodiments of the present disclosure. Such arrangements as further disclosed herein can be provided as software, code and/or other data structures arranged or encoded on a computer readable medium such as an optical medium (e.g., CD-ROM), floppy or hard disk, or other medium such as firmware or microcode in one or more ROM or RAM or PROM chips or as an Application Specific Integrated Circuit (ASIC). The software or firmware or other such configurations can be installed on a computerized device to cause one or more processors in the computerized device to perform the techniques explained herein.

[0023] Yet another more particular technique of the present disclosure is directed to a computer program product or computer environment that includes a computer readable medium having instructions stored thereon to facilitate use of shared information among multiple processes. Each of the multiple processes can support a technique of: i) initiating execution of a given processing thread including multiple instructions, the given processing thread producing a respective transaction outcome based on use of globally accessible shared data also accessible by other processing threads executing in parallel; ii) during execution of the given thread, initiate multiple reads of data from the globally accessible shared data for purposes of producing the trans-

action outcome; and iii) for each of the multiple reads, verify whether a respective globally accessible version information associated with the globally accessible shared data remains set to a constant value during execution of the given processing thread or portion thereof (such as from a first read to a next successive read), the globally accessible version information being changed to a new respective data value each time one of the processing threads modifies the globally accessible shared data. Other embodiments of the present disclosure include hardware and/or software programs to perform any of the method embodiment steps and operations summarized above and disclosed in detail below.

[0024] It is to be understood that the system of the invention can be embodied as a software program, as software and hardware, and/or as hardware alone. Example embodiments of the invention may be implemented within computer systems, processors, and computer program products and/or software applications manufactured by Sun Microsystems Inc. of Palo Alto, Calif., USA.

BRIEF DESCRIPTION OF THE DRAWINGS

[0025] The foregoing and other objects, features, and advantages of the present application will be apparent from the following more particular description of preferred embodiments of the present disclosure, as illustrated in the accompanying drawings in which like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, with emphasis instead being placed upon illustrating the embodiments, principles and concepts.

[0026] FIG. **1** is a diagram illustrating a computer environment including multiple parallel processes that access shared data according to embodiments herein.

[0027] FIG. **2** is a diagram of one of multiple processing threads utilizing a global accessible environment according to embodiments herein.

[0028] FIG. **3** is a diagram of a sample architecture supporting shared use of data according to embodiments herein.

[0029] FIG. **4** is a diagram of a flowchart illustrating a technique supporting simultaneous execution of multiple processing threads according to an embodiment herein.

[0030] FIGS. **5** and **6** combine to form a flowchart illustrating a technique supporting simultaneous execution of multiple processing threads according to an embodiment herein.

DETAILED DESCRIPTION

[0031] The present disclosure describes a unique way for each of multiple processes to operate in parallel using a common set of data (e.g., shared data) without causing corruption to the shared data. For example, as will be discussed in this specification, a computer system includes multiple processing threads that execute in parallel. The multiple processing threads have access to a global environment including i) shared data utilized by the multiple processing threads, ii) a globally accessible register or buffer of version information that changes each time a respective one of the multiple processing threads modifies the shared data, and iii) respective lock information indicating whether one of the multiple processing threads has locked the shared data preventing other processing threads from modifying the shared data.

[0032] To prevent data corruption, each of the processing threads monitors a status of the version information and the lock information during execution. If a given one of the multiple processing threads detects a change in the version information or another processing thread has a lock on the shared data, the given processing thread aborts and restarts execution again from the beginning. In one embodiment, each of the processes checks whether version information associated with shared content changes upon initiation of each of one or more successive read operations.

[0033] This technique of aborting a given processing thread is well suited for use in parallel processing applications that support a high number of reads relative to a corresponding number of fewer respective writes to shared data. For example, when there are a fewer number of processing threads changing the shared data, there will be a fewer number of processing threads will need to abort a respective transaction as a result of changes to the shared data.

[0034] FIG. **1** is a block diagram of a computer environment **100** according to an embodiment herein. As shown, computer environment **100** includes shared data **125** and corresponding metadata **135** in global environment **102** (e.g., a respective repository or global work environment) that is globally accessible by multiple processes **140** such as process **140-1**, process **140-2**, . . . process **140**-M. In one embodiment, each of processes **140** (e.g., processing threads) has a corresponding local repository **110** for storing information such as an instantaneous value of version information **120-1** at different times during execution of a corresponding set of multiple instructions.

[0035] In the context of a general embodiment herein, metadata **135** enables each of processes **140** to identify whether content associated with shared data **125** has been "locked" (e.g., via checking lock information **120-2**). Additionally, metadata **135** enables the processes **140** to identify whether any portions of shared data **125** have changed during execution of a respective transaction (e.g., via use of version information **120-1**).

[0036] In one embodiment, version information **120-2** and version information **120-1** form lock-word. A lock-word is a shared global variables that contain version information **120-1** and a lock-bit information **120-2**. A given globally accessible shared data structure (e.g., segment) is associated with only one lock-word. We say that lock-word protects or guards that data structure. The lock-word is meta-data that moderates access to the shared data structure.

[0037] Each of processes **140** competes to obtain a respective lock on shared data **125**. To obtain a lock preventing other processes from modifying shared data **125**, a given process writes a logic one to a respective lock bit of lock information **120-2**. When set to a logic one, processes **140** other than a respective process holding the lock are, by protocol, unable to modify contents of shared data **125**. Critically, M4 is a cooperative protocol. Threads use M4 to avoid races or undesirable interference. It's cooperative in the sense that all threads accessing the shared data will, by convention, use appropriate access protocol. (Nothing

would stop an errant from accessing the data. Upon release of a lock based on a respective process setting the lock bit to a logic zero, the processes **140** can again compete to obtain a lock on shared data **125**.

[**0038**] In addition to lock information **120-2**, metadata **135** includes version information **120-1** that is updated each time a respective process modifies contents of shared data **125**. Accordingly, the processes **140** can monitor the version information to identify whether shared data **125** has been modified during a respective execution of a processing transaction (e.g., multiple software instructions).

[**0039**] In one embodiment, each of processes **140** stores an instantaneous data value (e.g., sampled version value or sampled version number) of current version information **120-1** in a respective local repository **110** at least some time during a respective execution of a transaction. Throughout this specific, the instantaneous data value is defined as a current value of the version information **120-1** at a time of reading. As discussed throughout this specific, this value changes over time as each of the processes **140** modifies the respective shared data **125** and updates this version information **120-1** to a new value.

[**0040**] Some time after an initial load of version information **120-1** associated with shared data **125**, a respective process compares a data value associated with version information **120-1** in global environment **102** to the data value previously stored in its local repository **110** to identify whether another process **140** modified contents of shared data **125**. As will be discussed further in this specification, a respective process can abort itself and try again if version information **120-1** changes during respective execution.

[**0041**] Accordingly, the above algorithm includes a single lock that protects all transactions. Conceptually, the technique herein (e.g., a so-called M4 algorithm) is a read-write lock that provides automatic and transparent upgrade from read state to read-write state at the time of the 1st store in a transaction. The lock consists of a version field with the low-order bit serving as a write-lock. We describe M4 here as a specific example of a locking algorithm it may be beneficial to switch with, but anyone skilled in the art will know how to make the described switching mechanism work for another mechanism that implements transactions by having a single lock protect all executed transactions.

[**0042**] One embodiment of the M4 algorithm works as follows. We assume a read-write lock implemented using a single word, which we call the "lockword" (e.g., lock information **120-2**) with a version field (e.g., version information **120-1**).

1. According to One Embodiment, on a Transactional Load:

[**0043**] (a) if the loading thread holds the write-lock the load simply executes the load as normal.

[**0044**] (b) If the thread doesn't hold the write-lock it loads the lockword.

[**0045**] i. If the lockword indicates that some other thread holds the write-lock, the reading thread aborts immediately and retries.

[**0046**] ii. Otherwise:

[**0047**] A. if this is the 1st transactional load, the thread saves the fetched lockword which contains the version subfield) into a thread-local variable and then executes the equested load.

[**0048**] B. If this is not the 1st transactional load the thread checks the just-loaded version against the version saved in the thread's thread-local variable. If they disagree then the thread aborts immediately, otherwise the load executes as normal.

[**0049**] Note that the last two steps A and B can be modified to carry yet another embodiment herein. For example, in one embodiment, a transactional load can be carried out by first fetching data from memory and thereafter fetching (& validating) the version information. However, if this is a first load transaction, then the order of operation can include fetching the version information first, then fetching the shared data, and then fetching the version information again.

2. According to One Embodiment, on a Transactional Store:

[**0050**] (a) if the storing thread already holds the write-lock, the store executes normally.

[**0051**] (b) if the storing thread does not hold the write-lock:

[**0052**] i. If there was a previous transactional load, the thread attempts to CAS (atomic Compare and Swap instruction) the write-lock bit with the previously observed version. If the CAS is successful the thread now holds the write-lock and the store executes normal. If the CAS fails then the transaction aborts immediately.

[**0053**] ii. If there was no prior transactional load in this transaction the thread loops, trying to toggle the low-order write-lock bit in the lockword from 0 to 1. Once the thread has acquired the write-lock it executes the store or write operation.

[**0054**] According to the M4 technique, there's no need to track read-sets or write-sets so the overhead is quite low. A read set is a set of data locations read during a transaction. A write set is a set of data locations written to during a transaction as discussed in related applications. After a thread acquires the write-lock, it can simply store directly into the global fields without the need to save stores in a log file (e.g., a log file used for undo puporses). Likewise, loads don't need to look-aside into the store-log. Once the write-lock is acquired the operation can't abort.

[**0055**] In a related manner (since there's only one lock) there's no possibility of livelock or deadlock. A classic read-write lock can't normally be upgraded or promoted from read to read-write state without the application having been explicitly and intentionally written to refresh any cached values seen during the read phase. That's because the read-write lock implementation might need to drop the lock to avoid deadlock during the upgrade to a read-write state. The M4 technique as described herein provides for automatic and transparent upgrade from a read-only lock to a read-write lock. M4 tends to be more profitable (e.g., more efficient) in circumstances where read-parallelism exists.

[**0056**] Consider, for example, a hash table where most operations are queries. According to M4, pure readers may operate completely in parallel. Furthermore, the readers do not need to store into a shared location to acquire a lock. Note that traditional SMP systems use snoop- or directory-based cache coherency protocols, concurrent sharing of or access to cache lines tend to incur high latency and consume

precious coherency bandwidth if stores to the shared line are frequent. Pure read sharing, however, is typically inexpensive.

[0057] Furthermore, A number of variations are possible and it'll pay to enumerate them in the application. To keep things straight I'll call them Validate-after-each-ld (VAL) and validate-at-commit-time (VAC).

```
Transactional Load in VAL-mode:
    if (the loading thread holds the write lock) then
        execute the load of the global variable as normal
    else
        if (this is 1st load in the thread's txn) then
            load lock-word
            if (lock-word lock-bit is set) then
                optionally spin, waiting for lock-bit to clear
                abort current txn
            else
                save previously fetched lock-word version into
                thread-local txn version variable
        execute the load of the global variable placing result in
            thread-local temporary variable
        load the lock-word
        if (the lock-word's lock-bit is set OR
            the lock-word's version # differs from the thread's
            txn version variable) then abort the txn.
        return the temporary variable (previously fetched value)
            as the result of the txn load
```

[0058]

```
Transactional load in VAC-mode:
if (the loading thread holds the write lock) then
    execute the load of the global variable as normal
else
    if (this is 1st load in the thread's txn) then load lock-word
        if (lock-word lock-bit is set) then
            optionally spin, waiting for lock-bit to clear
            abort current txn
        else
            save lock-word version into thread-local txn version
            variable
            execute the load of the global variable as normal
```

[0059] In VAL-mode we don't need to re-check the version at commit-time whereas we must validate the version at commit-time in VAC-mode. The differences between VAC and VAL boil down to when and where do we validate that the saved txn version remains unchanged. VAC mode moves validation from the txn LD operation to the commit operator and defers validation until commit-time. Recall that validation tells us if the previously read global variables are mutually consistent. VAL mode is the preferred embodiment as it doesn't admit so-called "zombie" transactions. A zombie is a transaction that has read inconsistent data and is doomed, but has yet to validate and abort. Zombies can misbehave, dereferencing illegal pointers, generating divide-by-zero traps, or entering infinite loops. VAC-mode is slightly more efficient, but admits zombies. Zombies require "containment" to prevent their misbehavior from effecting the rest of the system.

[0060] Note that VAL- and VAC-mode define two ends of a spectrum. Another valid scheme is to validate periodically during the txn—less frequently than at every txn load, but more frequently than just at commit-time. Critically, VAL mode is still more efficient than the read-set validation performed by previous software transactional memory implementations (STMs).

[0061] FIG. **2** is a diagram more particularly illustrating details associated with a respective process according to embodiments herein. For clarity sake, FIG. **2** illustrates read and write operations that can be performed by any one of the multiple processes **140**. For example, each of the multiple processes **140** can simultaneously perform read operations. In general, the processes **140** do not compete with each other to perform read type of operations. However, a respective process must obtain the lock before modifying or writing to contents of shared data **125**.

[0062] Each of the multiple processes **140** can perform a respective read operation **205-1** to retrieve one or more data values associated with shared data **125**. Assume in this example that process **140-1** carries out read operation **205-1**. Note that each process **140** can operate in a similar manner.

[0063] When carrying out such a read operation as illustrated in step **210**, process **140-1** first checks whether a respective lock bit associated with lock information **120-2** has been set by another process currently modifying shared data **125**. If the lock bit is not set, the respective process performs a first read of a portion (e.g., one or more locations) of shared data **125**.

[0064] In step **215**, the respective process **140-1** initiates retrieval of respective version information **120-1** for storage in its corresponding local repository **110-1** (e.g., a buffer, register, allotted memory, etc.) For example, the version information at a time of first read may be 000145. The process **140-1** stores this instantaneous data value of 000145 in its local repository **110-1**. Note that the local repository can be a specific location of shared data assigned for use by only a respective process.

[0065] The process eventually performs subsequent reads in step **220**. For each subsequent read of shared data **125**, the process verifies that the lock bit is not set by another one of the multiple processes **140**. Additionally, the process **140-1** checks whether a current value of the version information **120-1** has been modified since the last read. For example, the process **140-1** compares an instantaneous data value of the version information **120-1** at a time of the subsequent read and compares it to the data value stored in local repository **110-1**. If they match (e.g., the instantaneous data value of the version information **120-2** is still 000145), the process **140-1** can continue executing a respective transaction. If not, such as when the instantaneous data value of the version information **120-1** is 000146 indicating that another process modified shared data **125**, then the process **140-1** aborts the transaction and retries from the start.

[0066] Note that in one embodiment, each of the processes **140** increments a respective value of the version information **120-1** upon committing corresponding results of a transaction to shared data **125**. In this latter case resulting in abortion, any intermediate data values generated by a respective processing thread prior to abortion are disregarded.

[0067] Note that the steps **210** through **220** can be modified to carry yet another embodiment herein. For example, in one embodiment, a transactional load can be carried out by

first fetching data from memory and thereafter fetching (& validating) the version information. However, for a respective first load transaction, the order of operation can include first fetching the version information, then fetching the shared data, and then fetching the version information again.

[0068] Each of the processes **140** can also perform a write operation **205-2**. A write operation involves modifying contents of shared data **125** during a so-called write phase. At the write-time in M4, if there were any prior writes in the thread's transaction, then the current thread must hold the write-lock. In this case the commit operation is a no-op. An M4 operation is guaranteed to commit normally (no aborts) if the thread manages to acquire (and validate) the M4 lock at the time of the 1st transactional write. If there are no writes in the transaction (e.g., it is a pure read transaction) then in VAL mode (as discussed above in FIG. **1**) the respective process doesn't need to do anything, in VAC-mode process validates that the saved thread-local transaction version information still matches the version in the global lock-word.

[0069] For example, in step **255**, a respective process **140-1** can initially verify that current version information associated equals the instantaneous data value of version information **120-1** stored in its local repository **110-1**. As discussed above, the process can abort itself if the version information changes during a course of executing a respective processing thread or transaction.

[0070] The process **140-1** also can retrieve the lock information **120-2** to identify whether another process currently has a lock on shared data **125**. If the lock bit is set, process **140-1** aborts a respective transaction. If the lock bit is not set, the process **140-1** sets the lock bit to an active state to obtain the lock over shared data **125**. Once a lock is obtained, no other processes can modify contents of shared data **125**.

[0071] After obtaining a respective lock in step **260**, the process **140-1** initiates modification of shared data **125** according to a respective transaction outcome in step **260**.

[0072] In step **265**, after modifying shared data **125** the process **140-1** updates the version information **120-1** with a new value (e.g., to a data value of 000146) and sets the lock bit (e.g., lock information **120-2**) to an inactive state (e.g., the process releases a respective lock where the lock bit and version information are co-located in a single lock-word). In one embodiment, the process **140-1** simultaneously sets a lock bit low at a same time of updating the version information **120-1**.

[0073] FIG. **3** is a block diagram illustrating an example computer system **610** (e.g., an architecture associated with computer environment **100**) for executing parallel processes **140** and other related processes according to embodiments herein. Computer system **610** can be a computerized device such as a personal computer, workstation, portable computing device, console, network terminal, processing device, etc.

[0074] As shown, computer system **610** of the present example includes an interconnect **111** that couples a memory system **112** storing shared data **125** (e.g., globally accessible shared data), metadata **135**, one or more processors **113** executing processes **140** (e.g., process **140-1**, process **140-2**, . . . , process **140**-M), an I/O interface **114**, and a communications interface **115**. Peripheral devices **116** (e.g., one or more optional user controlled devices such as a keyboard, mouse, display screens, etc.) can couple to processor **113** through I/O interface **114**. I/O interface **114** also enables computer system **610** to access repository **180** (that also potentially stores shared data **125** and/or metadata **135**). Communications interface **115** enables computer system **610** to communicate over network **191** to transmit and receive information from different remote resources.

[0075] Note that each of processes **140** can be executed by the same processor **113** supporting parallel processing of processes **140** or executed by a different set of respective processors executing one or more of processes **140**. In one embodiment, each of processes **140** executes a respective transaction associated with the same overall application. However, the processes **140** can be associated with different respective applications.

[0076] Further note that functionality associated with processes **140** and/or computer environment **100** can be embodied as software code such as data and/or logic instructions (e.g., code stored in the memory or on another computer readable medium such as a disk) that support functionality according to different embodiments described herein. Alternatively, the functionality associated with processes **140** can be implemented via hardware or a combination of hardware and software code. Processor **113** can be one or multiple separate processors executing multiple processing threads.

[0077] It should be noted that, in addition to the processes **140** themselves, embodiments herein include a respective application and/or set of instructions to carry out processes **140**. Such a set of instructions associated with processes **140** can be stored on a computer readable medium such as a floppy disk, hard disk, optical medium, etc. The set of instruction can also be stored in a memory type system such as in firmware, RAM (Random Access Memory), read only memory (ROM), etc. or, as in this example, as executable code.

[0078] Collective attributes associated with processes **140**, global environment **102**, and computer environment **100**, etc. will now be discussed with respect to flowcharts in FIGS. **4-6**. For purposes of this discussion, global environment **102**, processes **140** and/or computer environment **100** can execute or carry out the steps described in the respective flowcharts. Note that the steps in the below flowcharts need not always be executed in the order shown.

[0079] Now, more particularly, FIG. **4** is a flowchart **400** illustrating a technique supporting execution of parallel transactions in computer environment **100** according to an embodiment herein. Note that techniques discussed in flowchart **400** overlap and summarize some of the techniques discussed above.

[0080] In step **410**, a user or other source initiates execution of a given process (e.g., processing thread such as any one of processes **140**) including multiple corresponding instructions. The given process produces a respective transaction outcome based on use of globally accessible shared data **125** also accessible by other processes **140** executing in parallel.

[0081] In step **420**, during execution of the given process (e.g., processing thread), the given process initiates multiple

reads of data from the globally accessible shared data **125** for purposes of producing the transaction outcome.

[**0082**] In step **430**, the given process **140-1** verifies whether an instantaneous data value associated with a respective globally accessible version information (e.g., version information **120-1**) changes at a time from a first read to a next successive read. As previously discussed, the globally accessible version information (version information **120-1**) changes each time one of the process **140** modifies contents of the globally accessible shared data **125**.

[**0083**] FIGS. **5** and **6** combine to form a flowchart **500** (e.g., flowchart **500-1** and flowchart **500-2**) illustrating a technique supporting parallel execution of processes **140** according to embodiments herein. Note that techniques discussed in flowchart **500** overlap with the techniques discussed above in the previous figures.

[**0084**] In step **510**, a given process (e.g., one of the multiple processing threads executing in parallel) performs a first read of data from globally accessible shared data **125** if there is no corresponding lock on the globally accessible shared data **125**. A corresponding lock, if present, indicates that another one of the multiple processing threads is currently modifying the globally accessible shared data **125**.

[**0085**] In step **515**, the given process stores a data value associated with the version information **120-1** (e.g., globally accessible version information) in a local buffer (e.g., a respective local repository **110**) with respect to the given processing thread for the first read.

[**0086**] In step **520**, for each subsequent read issued by the given process, the given process verifies that there is no lock on the globally accessible shared data **125** and that a corresponding data value of the version information **120-1** at a time of a respective subsequent read is identical to the data value stored in the local buffer for the first read the given processing thread is aborted if the version information **120-1** changes over time.

[**0087**] In step **525**, after completing one or more subsequent reads and during a respective commit phase associated with the given process, the given process checks whether any one of the other processing threads currently has a lock on the globally accessible shared data **125**. In one embodiment, the given process can further check whether a current data value associated with the version information **120-1** match the data value in its the local buffer. Recall that the data value in the local repository associated with the given process stores a respective data value of the version information for the first READ. At commit time, the current value of version information **120-1** should be the same as the data value in the local repository.

[**0088**] In step **530**, if there is a lock as indicated by the checking or non-matching data value as indicated by the checking, the given process aborts the commit phase and re-executes the given process (e.g., transaction) again from the beginning. If few of the other processes **140** rarely initiate modification of shared data **125**, the given process will have a good chance of completing if re-executed again at a later time.

[**0089**] In step **610** of flowchart **500-2** of FIG. **6**, if there is not a lock as indicated by the checking, the given process initiates a respective commit phase with respect to the given process. In one embodiment, the given process carries out the following steps during a respective commit phase.

[**0090**] For example, in step **615** of the write phase (e.g., a period in a transaction from the first write until either abort or commit), the given process obtains a respective lock on the globally accessible shared data **125** preventing the other processing threads from modifying the globally accessible shared data. As previously discussed, the lock can be obtained by writing a logic one to a lock bit in lock information **120-2**.

[**0091**] In step **620**, after obtaining the lock, the given process initiates modification of the globally accessible shared data **125** in accordance with the respective transaction outcome associated with the given process.

[**0092**] In step **625**, after modification of globally accessible shared data **125**, the given process updates a respective data value associated with the version information to indicate to the other processing threads that the globally accessible shared data **125** has been modified. For example, if the data value of the version information **120-1** was 000145 before the modification, the process changes the version information to a data value of 000146.

[**0093**] In step **630**, the given process releases the respective lock on the globally accessible shared data **125**. For example, the given process resets the lock bit of lock information **120-2** to a logic zero. In one embodiment, the given process updates the version information and releases the lock at the same time via a single operation (e.g., store operation).

[**0094**] As discussed above, techniques herein are well suited for use in applications such as those that support parallel processing threads in the same processor or in different processors. However, it should be noted that configurations herein are not limited to such use and thus configurations herein and deviations thereof are well suited for use in other environments as well.

[**0095**] While this invention has been particularly shown and described with references to preferred embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the present application as defined by the appended claims. Such variations are covered by the scope of this present disclosure. As such, the foregoing description of embodiments of the present application is not intended to be limiting. Rather, any limitations to the invention are presented in the following claims. Note that the different embodiments disclosed herein can be combined or utilized individually with respect to each other.

We claim:

1. A method comprising:

initiating execution of a given processing thread including multiple instructions, the given processing thread producing a respective transaction outcome based on use of globally accessible shared data also accessible by other processing threads executing in parallel;

during execution of the given thread, initiating multiple reads of data from the globally accessible shared data for purposes of producing the transaction outcome; and

verifying whether a data value associated with a respective globally accessible version information variable changes at a time from a first read to a next successive read during a course of a transaction, the globally accessible version information variable being changed to a new respective data value each time one of the processing threads modifies contents of the globally accessible shared data.

2. A method as in claim 1, wherein verifying whether the respective globally accessible version information variable changes over successive reads enables the given processing thread to efficiently detect whether any of the other processing threads modified the contents of the globally accessible shared data during execution of the given thread from the first read to the next successive read.

3. A method as in claim 2 further comprising:

detecting that another processing thread modified contents of the globally accessible shared data during a respective time between the first read and the next successive read based on a change with respect to the globally accessible version information variable; and

in response to detecting the change, immediately aborting execution of the given processing thread.

4. A method as in claim 1, wherein verifying whether the respective globally accessible version information variable changes from the first read to the next successive read includes:

initiating the first read to retrieve respective data from the globally accessible shared data;

for the first read, retrieving and storing a respective data value associated with the globally accessible version information variable;

at a later time with respect to the first read, initiating the next successive read to retrieve respective data from the globally accessible shared data;

for the next successive read, retrieving a respective data value associated with the globally accessible version information variable; and

verifying that the respective data value of the globally accessible version information variable for the first read is identical to the respective data value of the globally accessible version information variable for the next successive read.

5. A method as in claim 1 further comprising:

for the given processing thread, performing the first read of data from the globally accessible shared data if there is no corresponding lock on the globally accessible shared data, the corresponding lock, if present, indicating that another one of the multiple processing threads is in a process of modifying the globally accessible shared data;

for the first read, storing a data value associated with the globally accessible version information variable in a local buffer with respect to the given processing thread;

for each subsequent read associated with the given processing thread, verifying that there is no lock on the globally accessible shared data and that a corresponding data value of the globally accessible version information variable at a time of a respective subsequent read is identical to the data value stored in the local buffer for the first read; and

initiating abortion of the given processing thread if there is a lock on the globally accessible shared data or if a respective data value associated with the globally accessible version information variable changes from the first read to a subsequent read.

6. A method as in claim 5 further comprising:

aborting further execution of the given processing thread if there is a corresponding lock on the globally accessible shared data, the corresponding lock indicating that another one of the multiple processing threads is in a process of modifying the globally accessible shared data.

7. A method as in claim 1 further comprising:

upon completion of the execution of the processing thread including the multiple reads, initiating storage of at least one data value associated with the respective transaction outcome to the globally accessible shared data;

prior to storage of the at least one data value to the globally accessible shared data, checking whether any of the other processing threads currently has a lock on the globally accessible shared data, presence of the lock indicating that another one of the multiple processing threads is in a process of modifying the globally accessible shared data; and

aborting the processing thread and corresponding storage of the at least one data value to the globally accessible shared data.

8. A method as in claim 1 further comprising:

prior to committing data values associated with the processing thread to the globally accessible shared data during a respective commit phase of the given processing thread, checking whether a current value of the globally accessible version information variable matches the constant value encountered from the first read to the next successive read; and

aborting the given processing thread if there is not a match between the current value of the globally accessible version information variable and the constant value encountered during the first read to the next successive read.

9. A method as in claim 1 further comprising:

checking whether any one of the other processing threads currently has a lock on the globally accessible shared data;

if there is a lock as indicated by the checking, then abort and re-execute the given processing thread;

if there is not a lock as indicated by the checking, initiating a respective commit phase with respect to the given processing thread, the commit phase including:

i) obtaining a respective lock on the globally accessible shared data preventing the other processing threads from modifying the globally accessible shared data,

ii) initiate modification of the globally accessible shared data in accordance with the respective transaction outcome associated with the given processing thread, and

iii) after the modification of globally accessible shared data, updating a respective data value associated with the globally accessible version information variable to indicate to the other processing threads that the globally accessible shared data has been modified.

10. A method as in claim 9, wherein the commit phase further comprises:

iv) releasing the respective lock on the globally accessible shared data at a same time as updating the respective data value associated with the globally accessible version information variable to a new value.

11. A computer system including:

multiple processing threads executing in parallel;

a global environment accessible by the multiple processing threads, the global environment including i) shared data utilized by the multiple processing threads, ii) version information that changes each time a respective one of the multiple processing threads modifies the shared data, and iii) respective lock information indicating whether one of the multiple processing threads has locked the shared data preventing other processing threads from modifying the shared data; and

each of the multiple processing threads accessing the version information at multiple times during execution of a respective processing thread to determine whether to abort execution of the respective processing thread as a result of another processing thread modifying the shared data.

12. A computer system as in claim 11, wherein the respective processing thread initiates retrieval of an instantaneous data value associated with the version information at a time of a first read of data from the set of shared data and checks whether the instantaneous data value for the first read matches an instantaneous data value of the version information at a time of a second read of data from the set of shared data, a respective match of the instantaneous data values indicating that no other processing thread modified contents of the shared data during execution of the respective processing thread.

13. A computer system as in claim 11, wherein each of the multiple processes continues to execute as long as the version information associated with the set of shared data does not change but otherwise abort if the respective processing thread identifies that the version information associated with the set of shared data changes over time.

14. A computer system as in claim 13, wherein each of the respective processing threads checks for changes associated with the version information each time the respective processing thread performs a respective successive read from the set of shared data.

15. A computer readable medium including:

instructions associated with a given processing thread that produces a respective transaction outcome based on use of globally accessible shared data also accessible by other processing threads executing in parallel;

instructions to initiate multiple accesses of data with respect to the globally accessible shared data for purposes of producing the transaction outcome during execution of the given processing thread; and

instructions to verify, at successive times during execution of the given processing thread, that a respective data value associated with a globally accessible version information variable does not change over time, the given processing thread aborting further execution if the globally accessible version information variable happens to change during a course of executing the given processing thread, the data value associated with the globally accessible version information variable being updated to a new value each time a respective one of the multiple processing threads modifies the globally accessible shared data.

16. A computer readable medium as in claim 15, wherein instructions to verify include:

instructions to carry out each of multiple reads and verify whether the globally accessible version information variable remains set to a constant value from a first read to a next successive read associated with the given processing thread.

17. A computer readable medium as in claim 15 further including:

instructions to initiate a first read to retrieve respective data from the globally accessible shared data;

instructions to retrieve and store a respective data value associated with the globally accessible version information variable at a time associated with the first read;

instructions to initiate a second read to retrieve respective data from the globally accessible shared data at a later time with respect to the first read;

instructions to retrieve a respective data value associated with the globally accessible version information variable at a time associated with the second read; and

instructions to verify that the respective data value of the globally accessible version information variable for the first read is identical to the respective data value of the globally accessible version information variable for the next successive read.

18. A computer readable medium as in claim 17 further including:

instructions to abort further execution of the given processing thread if there is a corresponding lock on the globally accessible shared data, the corresponding lock indicating that another one of the multiple processing threads is in a process of modifying the globally accessible shared data.

19. A computer readable medium as in claim 15 further including:

instructions to initiate storage of at least one data value associated with the respective transaction outcome to the globally accessible shared data upon completion of the execution of the given processing thread;

instructions to check whether any of the other processing threads currently has a lock on the globally accessible shared data, presence of the lock indicating that another one of the multiple processing threads is in a process of modifying the globally accessible shared data prior to storage of the at least one data value to the globally accessible shared data; and

instructions to abort the given processing thread and store the at least one data value to the globally accessible shared data.

20. A computer readable medium method as in claim 15 further comprising:

instructions to check whether any one of the other processing threads currently has a lock on the globally accessible shared data;

instructions to abort and re-execute the given processing thread if there is a lock as indicated by a checking process;

instructions to initiate a respective commit phase with respect to the given processing thread if there is not a lock as indicated by a checking process, the commit phase including:

i) instructions to obtain a respective lock on the globally accessible shared data preventing the other processing threads from modifying the globally accessible shared data,

ii) instructions to initiate modification of the globally accessible shared data in accordance with the respective transaction outcome associated with the given processing thread,

iii) instructions to update a respective data value associated with the globally accessible version information variable to indicate to the other processing threads that the globally accessible shared data has been modified after the modification of globally accessible shared data, and

iv) instructions to release the respective lock on the globally accessible shared data at a same time as updating the respective data value associated with the globally accessible version information variable to a new value.

* * * * *