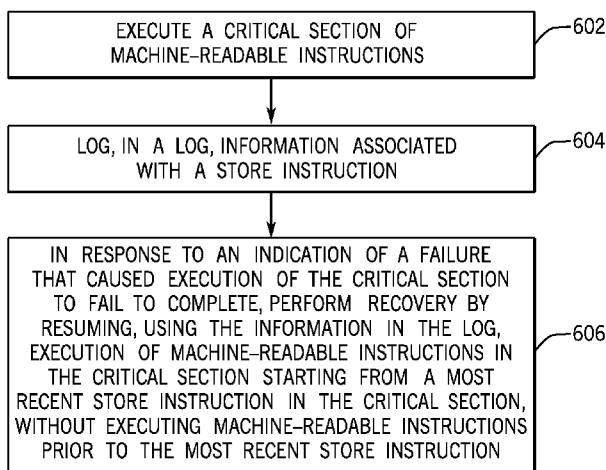




- (51) International Patent Classification:
G06F 11/16 (2006.01) *G06F 12/02* (2006.01)
- (21) International Application Number:
PCT/US2015/058446
- (22) International Filing Date:
30 October 2015 (30.10.2015)
- (25) Filing Language: English
- (26) Publication Language: English
- (71) Applicant: **HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP** [US/US]; 11445 Compaq Center Drive W., Houston, Texas 77070 (US).
- (72) Inventors: **IZRAELEVITZ, Joseph**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US). **KOLLI, Aasheesh**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US). **KELLY, Terence P.**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US). **MORREY III, Charles B.**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US).
- (74) Agents: **PAGAR, Preetam B.** et al.; Hewlett Packard Enterprise, 3404 E. Harmony Road Mail Stop 79, Fort Collins, Colorado 80528 (US).
- (81) Designated States (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.
- (84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK,

[Continued on next page]

(54) Title: RESUMING EXECUTION IN RESPONSE TO A FAILURE



(57) Abstract: A critical section of machine-readable instructions is executed, the critical section including instructions to acquire and release at least one lock. While executing the critical section, information associated with a store instruction that stores data to a persistent memory is logged in a log. In response to an indication of a failure that caused execution of the section to fail to complete, recovery is performed by resuming, using the information in the log, execution of machine-readable instructions of the section starting from a most recent store instruction in the critical section, without executing machine-readable instructions of the critical section prior to the most recent store instruction.

FIG. 6



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

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

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*

Published:

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

RESUMING EXECUTION IN RESPONSE TO A FAILURE

Background

[0001] A memory used to store information can include volatile memory (such as a dynamic random access memory or DRAM) or nonvolatile memory. Examples of nonvolatile memory include flash memory (which is block-addressable) and byte-addressable nonvolatile memory. A byte-addressable nonvolatile memory is a memory that supports access of individual bytes of data rather than larger units of data, such as blocks in flash memory.

Brief Description Of The Drawings

[0002] Some implementations are described with respect to the following figures.

[0003] Figs. 1 and 2 are block diagrams of example systems, according to some implementations.

[0004] Figs. 3A-3B depict example Failure Atomic SEctions (FASEs), according to some examples.

[0005] Fig. 4 is a block diagram of an example log according to some implementations.

[0006] Figs. 5A-5C illustrate an example high-level program code and example assembly codes transformed from the high-level program code, according to some implementations.

[0007] Fig. 6 is a flow diagram of an example process according to some implementations.

[0008] Fig. 7 is a block diagram of an example system according to further implementations.

Detailed Description

[0009] A nonvolatile memory is a memory that is able to maintain a stored state of data even when system power is removed from the nonvolatile memory. Examples of byte-addressable nonvolatile memory include phase change memory (PCM), memristor memory, spin-torque transfer memory, or another type of nonvolatile memory. A byte-addressable nonvolatile memory can be used as the replacement for volatile dynamic random access memory (DRAM) that is traditionally used to implement main memory in a system.

[0010] A nonvolatile memory is a specific category of a “persistent memory,” which can refer to memory that retains data beyond the lifetime of a process that placed the data in the memory. A process can refer to an instance of program code (including machine-readable instructions). The persistent memory is able to retain data stored by the process in the persistent memory, even after the process ends. A persistent memory can be byte-addressable.

[0011] By using persistent memory to implement main memory, data can become persistent as soon as the data reaches the main memory. By using byte addressable persistent memory, access of durable storage can be performed using processor LOADs and STOREs. Processor LOAD can refer to a load operation requested by a processor to retrieve data from memory. Processor STORE can refer to a store operation requested by a processor to store (write) data to memory.

[0012] Although reference is made to byte-addressable persistent memory in some examples, it is noted that in other examples, persistent memory can be accessed in units of sizes different from a byte.

[0013] In some examples, a processor cache can also be implemented with persistent memory. A cache refers to memory that is more closely associated with a processor than main memory, such that data in the cache can be accessed more quickly than main memory by the processor.

[0014] During execution of machine-readable instructions of a transaction (explained below) that accesses persistent memory, failures may cause the transaction to crash and thus be unable to complete. A “transaction” can refer to any collection of actions that are to be performed together, where it is expected that all of the actions of the transaction should complete for successful operation (in other words, partial completion of a subset of the actions of the transaction is not allowed). Failures can be caused by a system power outage, an operating system error, a crash of an executing program code, and so forth.

[0015] Various types of recovery techniques can be performed in response to such failures, such as recovery techniques based on logs provided by UNDO logging or REDO logging. With UNDO logging, prior to modification, states about to be changed are stored in a log (a “state about to be changed” can refer to a data value stored in memory which is about to be changed), and if a failure occurs that crashes a transaction, the entire transaction is rolled back to an initial state prior to the start of the transaction by writing the contents of the UNDO log back to the locations that were being modified within a transaction when the failure occurred. With REDO logging, changed states are stored privately in a log, and if a failure occurs that crashes the transaction, all of the transaction’s changes are repeated (a redo) to complete the transaction, assuming that the corresponding REDO log is complete. If the REDO log is not complete (e.g. a crash may have occurred while the REDO log is being written), then the REDO log is discarded. The net effect of discarding the REDO log is that none of the changes contained in the incomplete transaction are made.

[0016] With UNDO logging and REDO logging, a relatively large amount of bookkeeping information may have to be kept to perform rollback or redo, respectively. In accordance with some implementations of the present disclosure, recovery techniques or mechanisms are provided that support resumption from a point of failure in a critical section of program code (that is made up of machine-readable instructions). Rather than perform rollback, the recovery can be resumed from the most recent store instruction (an instruction to store or write data to the

persistent memory) executed, without having to perform (i.e. repeat) execution of instructions of the section of program code prior to the most recent store instruction. A most recent store instruction in a critical section can refer to the last store instruction of the critical section that was executed. In some implementations, a critical section of program code includes a program code section that (1) starts when the program code that currently does not hold a lock acquires a first lock and (2) ends when that program code releases its last lock, where the last lock can be the same as or different from the first lock. In further examples, the critical section of a program code can include instructions to acquire nested locks or non-nested locks (e.g. hand-over-hand locks)—examples of nested locks and hand-over-hand locks are discussed further below. A critical section of a program code can also be referred to as a Failure Atomic SEction (FASE).

[0017] By resuming execution of the FASE following failure and executing the FASE to completion, a design can be provided that employs reduced logging as compared to systems that employ UNDO logging or REDO logging.

[0018] In some implementations, the technique of resuming execution from the point of failure from the most recent store operation of a FASE is referred to as JUSTDO logging.

[0019] Fig. 1 is a block diagram of an example system 100 that includes a processor 102 and persistent memory 104 (e.g. byte-addressable persistent memory) that can be accessed by the processor 102. The persistent memory 104 can include main memory and/or cache memory. The system 100 can be a computer (e.g. a desktop computer, a server computer, a notebook computer, a tablet computer, etc.), a smart phone, a wearable device that is worn on a body of a user (e.g. a smart watch, smart eyeglasses, etc.), a game appliance, a vehicle, or any other electronic device. A processor can refer to a microprocessor, a core or multiple cores of a microprocessor, a microcontroller, a programmable gate array (PGA), an application specific integrated circuit (ASIC), or any processing hardware capable of executing machine-readable instructions.

[0020] The processor 102 can execute various machine-readable instructions, including program code 106, logging instructions 107, and recovery instructions 108 according to some implementations of the present disclosure for recovering from a failure using JUSTDO logging. The program code 106, the logging instructions 107, and the recovery instructions 108 are stored in a memory or storage 103. In further examples, any or some combination of the program code 106, the logging instructions 107, and the recovery instructions 108 can be alternatively stored in the persistent memory 104. Although Fig. 1 shows the system 100 with just one processor 102, it is noted that in other examples, the system 100 can include multiple processors.

[0021] The program code 106 can be a thread of a program, such as an application, an operating system (OS), or any other type of machine-readable instructions. In some examples, a system can support multi-threaded execution, where multiple threads of a program can concurrently execute in the system.

[0022] The program code 106 includes a FASE 110, which includes an instruction to acquire a lock (or multiple instructions to acquire locks) and an instruction to store data (or multiple instructions to store data). Note the reference to the program code 106 including an FASE 110 is a reference to a program code that includes just one FASE or multiple FASEs.

[0023] A store instruction in the FASE 110 causes a store operation 112 to be performed with respect to the persistent memory 104, by writing data 114 to the persistent memory 104.

[0024] In some implementations, a FASE that employs JUSTDO logging accesses just persistent memory, which ensures that all data for resuming an interrupted FASE will be available during recovery by the recovery instructions 100. More specifically, in some implementations, all memory loads and stores within a FASE have to access only persistent memory. This extends to thread-local locations that would normally be transient, such as variables on a stack. These thread-local locations can be moved to the persistent memory 104 prior to entering the FASE.

Note that the thread-local locations can be moved out of the persistent memory 104 at the end of the FASE. By specifying that a FASE can access only persistent memory, it can be ensured that no updates in a FASE are dependent on state destroyed by failure.

[0025] In some implementations, the persistent memory 104 can also store a log 116 for use by the recovery instructions 108 for recovering from a failure during execution of the FASE 110. In some examples, the log 116 can be referred to as a JUSTDO log. In other implementations, the log 116 can be stored in a separate persistent memory (or other persistent storage) than the persistent memory 104 used to store data 114.

[0026] The log 116 can record the most recent store instruction that the program code 106 performed within the FASE 110. In some examples, the log 116 can contain the following information elements: a destination address of the store instruction (where the destination address specifies a location in the persistent memory 104 for data to be stored by the store instruction), the value to be placed at the destination address, the size of the value, and a value of the program counter of the processor 102 during execution of the program code 106. In other examples, alternative or additional information elements can be stored in the log 116.

[0027] The program counter of the processor 102 stores an address of an instruction that is executed. As each instruction is executed, the program counter is updated to point to the next instruction.

[0028] Prior to performing a store instruction in the FASE 110, the foregoing information relating to the store instruction is loaded into the log 116. The loading of the information into the log 116 can be performed by the logging instructions 107.

[0029] Fig. 2 illustrates an example arrangement that includes multiple threads 202 and 204 of program code executed by the processor 102, where each thread 202 or 204 includes a respective FASE 206 or 208. Although two threads 202 and

204 are shown in Fig. 2, note that the processor 102 is able to execute more than two threads in other examples.

[0030] Each thread 202 or 204 can maintain a respective JUSTDO log 210 or 212, which can be stored in the persistent memory 104 or in some other persistent storage according to some implementations.

[0031] Each log 210 or 212 records the most recent store instruction that the respective thread performed within a FASE. Each log 210 or 212 can contain the same types of information elements as the log 116 discussed above.

[0032] As further shown in Fig. 2, lock ownership logs 214 can be provided (*e.g.* stored in the persistent memory 104 or in some other persistent storage) to maintain a record of mutex ownership by the respective threads 202 and 204. One lock ownership log 214 can be maintained for each thread. Although the lock ownership logs 214 are depicted as being separate from the JUSTDO logs 210 and 212, it is noted that in other examples, a lock ownership log 214 can be combined with a JUSTDO log, so that the JUSTDO log also stores lock ownership information. Use of the lock ownership logs 214 is discussed further below. Note that in further implementations, another log (referred to as a “lock intention log”) relating to lock ownership can be used (discussed further below).

[0033] Figs. 1 and 2 depict the program code 106 and the threads 202 and 204 being able to access the persistent memory 104 directly. In such examples, each of the program code 106 or the threads 202 and 204 can be an instance of a program (*e.g.* application program) that is able to issue requests to directly access the persistent memory 104. In other examples, the program (*e.g.* application program) invokes an intermediate entity, such as a library or a trusted intermediary, to access the persistent memory 104.

[0034] In some examples, the library can manage durable data structures, such as those stored in the persistent memory 104, on behalf of applications executing in the system 100. Any access of the persistent memory 104 by an instance of the

application (e.g. the program code 102 or the threads 202 and 204) is handled by the library, which assumes responsibility for orderly concurrent access to shared data in the persistent memory 104 and for recovering the data in the persistent memory 104 to a consistent state following failures. In such implementations, the logging instructions 107 and the recovery instructions 108 are part of the library.

[0035] JUSTDO logging enables expert library developers to write lock-based FASEs in library routines and employ JUSTDO logging to ensure consistent recoverability with low overhead during failure-free operation. A well-designed JUSTDO-based library can consolidate persistent data updates in small FASEs that lend themselves readily to powerful software quality assurance techniques.

[0036] A related use case involves application logic of questionable quality or security that is be constrained to manipulate valuable durable data only indirectly, via a trusted intermediary. JUSTDO logging provides a new high-performance logging strategy for the intermediary in mediated access scenarios. OS process boundaries coupled with user permissions can isolate untrusted application code from trusted intermediary code, allowing only the latter direct access to persistent data in the persistent memory 104.

[0037] FASE

[0038] Fig. 3A and 3B depict two examples of lock-delimited FASEs. A FASE 302 shown in Fig. 3A includes nested locks, whereas a FASE 304 shown in Fig. 3B includes hand-over-hand locks.

[0039] In the FASE 302, a first lock (lock1) is acquired by instruction 1 (lock1.lock()). The first lock, lock1, is released by instruction 4 (lock1.unlock()). Instructions 1 and 4 thus define a first critical section 306. In some implementations, a critical section is a section of program code that is delimited by a lock acquisition (of a given lock) and a lock release (of the given lock).

[0040] Within the critical section 306 is instruction 2 (lock2.lock()) to acquire a second lock, lock2. The second lock, lock2, is released by instruction 3

(lock2.unlock()). Instructions 2 and 3 define a second critical section 308, which is nested within the first critical section 306.

[0041] The FASE 304 of Fig. 3B includes non-nested locking, in which instruction 5 acquires a first lock, lock1. Before release of the first lock, instruction 6 acquires a second lock, lock2. Release of the first lock is performed at instruction 7, and release of the second lock is received at instruction 8. Thus, instructions 5 and 7 define a first critical section 310, and instructions 6 and 8 define a second critical section 312. The critical sections 310 and 312 overlap, but one is not nested within the other. This type of overlapping critical sections is referred to as hand-over-hand locking.

[0042] In the examples of Figs. 3A and 3B, a FASE is an outermost critical section protected by one or multiple locks. For example, in Fig. 3A, the outermost critical section 306 is protected by lock1 (where lock1 is acquired at instruction 1 and released at instruction 4); thus, in Fig. 3A, the FASE 302 is this outermost critical section 306. In contrast, in Fig. 3B, the outermost critical section starts with the acquisition of lock1 at instruction 5 and the release of lock2 at instruction 8. Thus, the lock acquired at the start of a FASE does not have to be the same as the lock released at the end of the FASE, as shown in Fig. 3B.

[0043] In some examples, a lock used in a FASE can be a mutex. A mutex is a synchronization object whose state is set to a first value when the mutex is not owned by any thread, and set to a second value when the mutex is owned by a thread. Just one thread at a time can own a mutex. For example, to prevent two threads from writing to shared memory at the same time, each thread waits to acquire ownership of a mutex before executing the code that accesses the memory. After writing to the shared memory, the thread releases the mutex.

[0044] A lock ownership log 214 depicted in Fig. 2 can maintain a record of mutex ownership (e.g. ownership of lock1 or lock2) by a respective thread. Note that there can be one lock ownership log 214 maintained per thread. Thus, in response to a given thread acquiring ownership of a first mutex, the respective lock ownership

log 214 is updated to indicate ownership of the first mutex by the given thread. In response to the given thread subsequently releasing ownership of the first mutex, the respective lock ownership log 214 is updated to indicate that the first mutex has been released and no thread owns the first mutex.

[0045] JUSTDO Log

[0046] Fig. 4 illustrates an example format of JUSTDO log 400 (e.g. the log 116 in Fig. 1 or the log 210 or 212 in Fig. 2). In examples according to Fig. 4, the JUSTDO log 400 can include two entries 0 and 1, where one of the entries 0 and 1 is an active entry and the other of the entries 0 and 1 is an inactive entry. Just one entry of the log is active at any one time (i.e. either entry 0 or entry 1 is active at one time). An indicator bit 402 can be used to indicate which of the entries 0 and 1 is active. For example, the indicator bit 402 having a first value indicates that entry 0 is active and entry 1 is inactive, while the indicator bit 402 having a second value indicates that entry 0 is inactive and entry 1 is active. The JUSTDO log 400 is a per-thread log; in other words, there is one JUSTDO log 400 per thread. The JUSTDO log 400 is written to by a FASE before a memory store instruction in the FASE is performed.

[0047] The single active log entry contains a store address 404 to which data is to be written, a value 406 to be written there, a size 408 of the write (where the size 408 specifies the size of the value 406), and a value 410 of the program counter of the processor that executes the respective thread. The program counter value 410 includes the address of the most recent store instruction of the FASE executed by the processor 102. Immediately after the active log entry is updated, the corresponding store operation is performed.

[0048] The JUSTDO log 400 is updated for every store instruction within a FASE. Thus, each store instruction in a FASE is converted to both a log update (to write information to the active entry of the JUSTDO log 400) and the store operation of data to the persistent memory 104.

[0049] In examples according to Fig. 4, the program counter value 410 is shared between the entries 0 and 1, and a low order bit of the program counter value 410 can be used as the indicator bit 402 to indicate which entry is active. To update the JUSTDO log 400, both the new store value 406 and the store address 404 are stored in the inactive entry, followed by a release fence to ensure that the writes have reached the persistent cache. Subsequently, the new program counter (with the indicator bit set for the recently updated entry) is stored. The use of an inactive entry and active entry in the JUSTDO log 400 allows for an atomic update of the log 400.

[0050] After the log has been successfully updated, we execute a release fence (again to ensure that the updates are persistent), then complete the persistent STORE by writing the new value to the destination address.

[0051] Store Annotation

[0052] As noted above, each store instruction in a FASE is converted to a JUSTDO log update and a store operation specified by the store instruction. Such conversion can be performed by a compiler that compiles the program code into executable code, such as machine assembly code.

[0053] Fig. 5A shows an example high-level program code, which sets $A = X + 5$ and $B = Y + 5$. A compiler can compile the high-level program code of Fig. 5A into a “regular” assembly code as shown in Fig. 5B, where a regular assembly code refers to assembly code not configured to support JUSTDO logging recovery. In the regular assembly code of Fig. 5B, A, B, X, and Y are persistent memory locations, LDR is a load instruction, STR is a store instruction, and ADD is an instruction to add two values.

[0054] To support JUSTDO logging recovery, the compiler according to some implementations is able to translate the high-level program code to the JUSTDO assembly code shown in Fig. 5C. In the JUSTDO assembly code, a log update instruction to update a JUSTDO log is added prior to each STR instruction (store

instruction), at lines J3 and J7, where the log update instruction is generally expressed as JD L, <Addr, Val, PC>, where JD indicates a log update, L represents the JUSTDO log, Addr is the store address (e.g. 404 in Fig. 4), Val is the store value (e.g. 406 in Fig. 4), and PC is the program counter value (e.g. 410 in Fig. 4). The JD L instruction writes the values of Addr, Val, and PC to the respective JUSTDO log 400.

[0055] Example Process

[0056] Fig. 6 is a flow diagram of an example process according to some implementations. The processor 102 executes (602) a critical section of machine-readable instructions (also referred to as a FASE above).

[0057] While executing the critical section, the processor 102 logs (at 604), in a log (e.g. the JUSTDO log 400), information associated with a store instruction that stores data to the persistent memory 104. As examples, the information stored includes the information stored to an entry of the JUSTDO log 400 depicted in Fig. 4. The logging of the information into the log can be performed by the logging instructions 107 (Fig. 1 or 2).

[0058] In response to an indication of a failure that caused execution of the critical section to fail to complete, the recovery instructions 108 (Fig. 1 or 2) are executed by the processor 102 to perform (at 606) recovery by resuming, using the information in the log, execution of machine-readable instructions of the critical section starting from a most recent store instruction in the critical section, without executing machine-readable instructions of the section prior to the most recent store instruction. The indication of the failure can be provided by persistently stored state information of a crashed process, for example.

[0059] The point of failure at which the recovery resumes can be the address represented by the program counter value 410 that corresponds to the most recently executed store instruction of the section that has experienced the failure. More specifically, the recovery instructions 108 can cause the interrupted section (e.g.

FASE) to re-enter execution at the program counter value 410 stored in the active entry of the JUSTDO log 400, to cause re-execution of the store instruction identified by the program counter value 410, and continue execution to the end of the section (e.g. FASE).

[0060] In some implementations, the execution of the FASE starting at the point identified by the program counter value 410 and continuing to the end of the FASE can be performed by a recovery thread created by the recovery instructions 108 for failure recovery. The recovery thread acquires locks in the respective lock ownership log (e.g. 214 in Fig. 2), and proceeds with execution of the FASE starting at the point identified by the program counter value 410 and continuing to the end of the FASE. By acquiring and releasing locks according to the lock ownership log, recovery-time completion of a FASE is able to respect all mutual exclusion constraints present in the program code.

[0061] Note that in cases where a program code (or thread) that has experienced a crash has multiple FASEs, then multiple recovery threads are started by the recovery instructions 108, where the multiple recovery threads use respective multiple JUSTDO logs corresponding to the multiple FASEs to perform JUSTDO recovery of the multiple FASEs. The multiple recovery threads can execute in parallel until all of the FASEs are complete.

[0062] In some implementations, both cache memory and main memory can be implemented with the persistent memory 104, which can be made up of one or multiple persistent memory devices. In other implementations, main memory is implemented using the persistent memory 104, while cache memory is implemented using volatile memory.

[0063] In implementations where the cache memory is implemented using volatile memory, a flush-on-failure technique or mechanism can be employed to flush the content of the cache memory to the persistent memory 104 in response to a failure.

[0064] Lock Logs

[0065] As discussed further above, Fig. 2 depicts the lock ownership log 214 associated with each thread. In some implementations, another log, referred to as a lock intention log, can be used in conjunction with the lock ownership log 214. To recover from failure, each recovery thread has to know which locks the recovery thread holds.

[0066] To preserve lock ownership information across crashes, locks can reside in persistent memory. In some implementations, threads can maintain two per-thread persistent logs to facilitate proper restoration of lock ownership during recovery: a lock intention log and a lock ownership log. The purpose of the lock intention log is to speed recovery by obviating having to inspect all locks in persistent memory, whereas the lock ownership log is used to re-assign locks to recovery threads.

[0067] Immediately prior to attempting a lock acquisition, a thread declares its intent by recording the lock address in the lock intention log. Immediately after acquiring the lock, the thread records the acquisition in the lock ownership log using a JUSTDO store. To release a lock, a thread performs the same operations in reverse order. The thread first uses a JUSTDO store operation to remove the lock from the thread's lock ownership log, then unlocks the lock, and finally removes the lock from the lock intention log. This protocol ensures that following a crash the per-thread lock intention logs collectively record all locks that might be locked, and the lock ownership logs record which thread has locked each lock that is certainly locked.

[0068] Recovery

[0069] The following provides further details regarding recovery from failure of a FASE, according to further implementations.

[0070] Recovery can begin by using the per-thread lock intention logs to unlock all locks that may have been locked at the moment of failure. Without lock intention

logs, unlocking all locks would involve inspecting the locks or using generational locks. The lock intention log enables both arbitrary lock implementations and fast recovery. In other implementations, lock intention logs can be omitted.

[0071] After unlocking all locks, the recovery instructions 108 spawn one thread per non-empty JUSTDO log 400. A recovery thread's duty is to execute to completion a corresponding FASE that had been cut short by failure. Each recovery thread inherits a JUSTDO log and the pair of lock logs left behind by its deceased predecessor.

[0072] Recovery threads begin by acquiring all locks in their respective lock ownership logs, then waiting at a barrier for all other threads to do likewise. Once all locks have been acquired by all recovery threads, each thread re-executes the store instruction contained in the thread's JUSTDO log. Next, each recovery thread jumps to the program counter value (410) contained in the JUSTDO log and executes the remainder of the interrupted FASE. Recovery threads track the number of locks the threads hold, and when this count drops to zero the FASE has been completed and the recovery threads exit.

[0073] Recovery is executed with an interleaving of instructions (either in parallel or by context switching across recovery threads). Some FASEs may be blocked waiting for other FASEs to release locks. This interleaving arrangement allows for parallel recovery. Furthermore, once the recovery threads have re-acquired all of their locks and passed the barrier, access to a shared persistent state is properly synchronized by the appropriate locks. Consequently, the resurrected program code may spawn ordinary (non-recovery) threads that operate, with appropriate synchronization, upon persistent memory even before the recovery threads have completed the execution of interrupted FASEs. In other words, the restoration of consistency to persistent memory can proceed in parallel with resumed program code execution.

[0074] Register Promotion in a FASE

[0075] To allow proper recovery of a FASE, techniques or mechanisms can be provided to ensure that instructions in the FASE do not depend on data held only in volatile registers, such as registers in the processor 102. In some implementations, ensuring that instructions in the FASE do not depend on data held only in volatile registers can be accomplished by preventing register promotion of persistent memory values within a FASE.

[0076] Register promotion is an optimization performed by a compiler that eliminates redundant load operations from the persistent memory 104, by caching data in persistent memory locations in processor registers. Register promotion in FASEs is problematic for JUSTDO logging. Consider a value in the persistent memory 104 that is loaded by a load instruction in an example FASE from the persistent memory 104 into a register. Also, in the example FASE, two subsequent store instructions (first store instruction followed by the second store instruction) depend on this value that is loaded into the register. If, due to register promotion, the value is not re-loaded from the persistent memory 104 into the register after the first store instruction but prior to performing the second store instruction, recovery from a crash immediately after the first instruction is executed is not possible, since the crash causes loss of the value in the register, on which the second store instruction depends.

[0077] In some example implementations, register promotion of persistent memories used by a store instruction in a FASE is prevented. As an example, preventing register promotion can be accomplished by using a “volatile” keyword according to the C or C++ programming language. The “volatile” keyword is a qualifier that can be applied to a variable when the variable is declared in the program code. The “volatile” keyword indicates to the compiler that the value of the variable may change at any time.

[0078] In accordance with some implementations of the present disclosure, within a FASE, each load instruction occurs via a volatile pointer, and thus the

corresponding value that is being loaded cannot be subjected to register promotion by compiler optimization; register promotion would elide the load instruction to load the value from the persistent memory 104.

[0079] Although reference is made to use of the “volatile” keyword according to C or C++, it is noted that if other programming languages are used, other similar keywords to prevent register promotion can be employed.

[0080] In other implementations, instead of using the “volatile” or other like keyword to indicate to the compiler that register promotion is not to be performed, a compiler can be configured to selectively disable register promotion in FASEs, while allowing register promotion in code outside FASEs.

[0081] Fig. 7 is a block diagram of a system 700 according to further implementations. The system 700 includes a non-volatile machine-readable or processor-readable storage medium that stores machine-readable instructions executable in the system 700, such as by a processor.

[0082] The machine-readable instructions include compiling instructions 704 (e.g. a compiler) that compiles program code to generate executable code, such as the assembly code shown in Fig. 5B. When compiling a program code that includes a FASE, the compiling instructions 704 generates executable code including a store instruction in the FASE, and adds a log update instruction (to update a JUSTDO log) to the executable code before the store instruction.

[0083] The machine-readable instructions also include recovery instructions 706 to, in response to a failure during execution of the executable code that prevents the FASE from completing, perform recovery by resuming, using information in the JUSTDO log provided by the log update instruction, execution of machine-readable instructions of the FASE starting at a point of the failure, without executing machine-readable instructions of the FASE prior to the point of the failure.

[0084] The storage medium (or storage media) 702 can include one or multiple different forms of memory including semiconductor memory devices such as

dynamic or static random access memories (DRAMs or SRAMs), erasable and programmable read-only memories (EPROMs), electrically erasable and programmable read-only memories (EEPROMs) and flash memories; magnetic disks such as fixed, floppy and removable disks; other magnetic media including tape; optical media such as compact disks (CDs) or digital video disks (DVDs); or other types of storage devices. Note that the instructions discussed above can be provided on one computer-readable or machine-readable storage medium, or alternatively, can be provided on multiple computer-readable or machine-readable storage media distributed in a large system having possibly plural nodes. Such computer-readable or machine-readable storage medium or media is (are) considered to be part of an article (or article of manufacture). An article or article of manufacture can refer to any manufactured single component or multiple components. The storage medium or media can be located either in the machine running the machine-readable instructions, or located at a remote site from which machine-readable instructions can be downloaded over a network for execution.

[0085] In the foregoing description, numerous details are set forth to provide an understanding of the subject disclosed herein. However, implementations may be practiced without some of these details. Other implementations may include modifications and variations from the details discussed above. It is intended that the appended claims cover such modifications and variations.

What is claimed is:

1. A method comprising:
 - executing, in a system comprising a processor, a critical section of machine-readable instructions, the critical section including instructions to acquire and release at least one lock;
 - while executing the critical section, logging, in a log, information associated with a store instruction that stores data to a persistent memory; and
 - in response to an indication of a failure that caused execution of the critical section to fail to complete, performing recovery by resuming, in the system using the information in the log, execution of machine-readable instructions of the critical section starting from a most recent store instruction in the critical section, without executing machine-readable instructions of the critical section prior to the most recent store instruction.
2. The method of claim 1, wherein executing the critical section of machine-readable instructions comprises executing the critical section that includes instructions to acquire nested locks or un-nested locks.
3. The method of claim 1, wherein logging the information associated with the store instruction comprises logging a value of a program counter, a value to be stored by the store instruction, and an address that is a subject of the store instruction.
4. The method of claim 3, wherein the log includes a plurality of entries, wherein each entry of the plurality of entries is to store a value of the program counter, a value to be stored by a respective store instruction, and an address that is a subject of the respective store instruction, the log further comprising an indicator to indicate a first entry of the plurality of entries as active, and a second entry of the plurality of entries as inactive.

5. The method of claim 1, wherein the log stores information for the most recent store instruction in the critical section.
6. The method of claim 1, further comprising:
starting a recovery thread to perform the recovery, the recovery thread resuming execution at the value of the program counter in an active entry of the log.
7. The method of claim 1, further comprising:
executing another critical section of machine-readable instructions, the another critical section including instructions to acquire and release at least one lock;
while executing the another section, logging, in another log, information associated with a store instruction that stores data to the persistent memory;
in response to the indication of the failure, starting respective recovery threads to perform recovery of the respective critical sections; and
executing the recovery threads in parallel until all the critical sections are complete.
8. The method of claim 1, further comprising:
determining, as part of performing the recovery, a lock held by the critical section.
9. The method of claim 1, further comprising:
storing, in the log or another log, information relating to acquisition of a lock by the critical section and ownership of the lock; and
as part of performing the recovery, using the stored information relating to acquisition of the lock and ownership of the lock.

10. A system comprising:

a persistent memory;

a processor;

program code comprising a critical section including instructions to acquire and release at least one lock on data, and a store instruction to store data to the persistent memory; and

program instructions executable on the processor to:

prior to executing the store instruction, store information relating to the store instruction in a log, the information comprising a program counter value, a value to be written by the store instruction, and an address that is a subject of the store instruction,

in response to an indication of a failure that prevents completion of execution of the program code, start a recovery thread to resume execution of the section starting from a point specified by the program counter value, the recovery thread to complete execution of the critical section.

11. The system of claim 10, wherein the program code comprises another critical section including instructions to acquire and release a lock on data, and a store instruction to store data to the persistent memory, wherein the program instructions are executable on the processor to:

prior to executing the store instruction in the another critical section, store information relating to the store instruction in the another critical section in another log, the information stored in the another log comprising a program counter value, a value to be written by the store instruction in the another section, and an address that is a subject of the store instruction in the another section,

in response to the indication of the failure, start another recovery thread to resume execution of the another critical section starting from a point specified by the program counter value in the another log, the another recovery thread to complete execution of the another critical section.

12. The system of claim 10, further comprising a lock ownership log to store information relating to ownership of a lock, acquisition of the lock, and release of the lock, wherein the recovery thread is to acquire and release the lock according to the lock ownership log.

13. An article comprising at least one non-transitory machine-readable storage medium storing program instructions that upon execution cause a system to:

- compile a program code that includes a critical section including instructions to acquire and release at least one lock on data, and a store instruction to store data to a persistent memory,

- the compiling comprising:

- generating executable code comprising the store instruction, and

- adding a log update instruction to the executable code before the store instruction; and

- in response to a failure during execution of the executable code that prevents the critical section from completing,

- perform recovery by resuming, using information in a log provided by the log update instruction, execution of machine-readable instructions of the critical section starting from a most recent store instruction in the critical section, without executing machine-readable instructions of the critical section prior to the most recent store instruction.

14. The article of claim 13, wherein the compiling avoids performing register promotion for a load instruction in the critical section.

15. The article of claim 13, wherein the critical section is a program code section that starts when the program code that currently does not hold a lock acquires a first lock and ends when the program code releases a last lock, where the last lock can be the same as or different from the first lock.

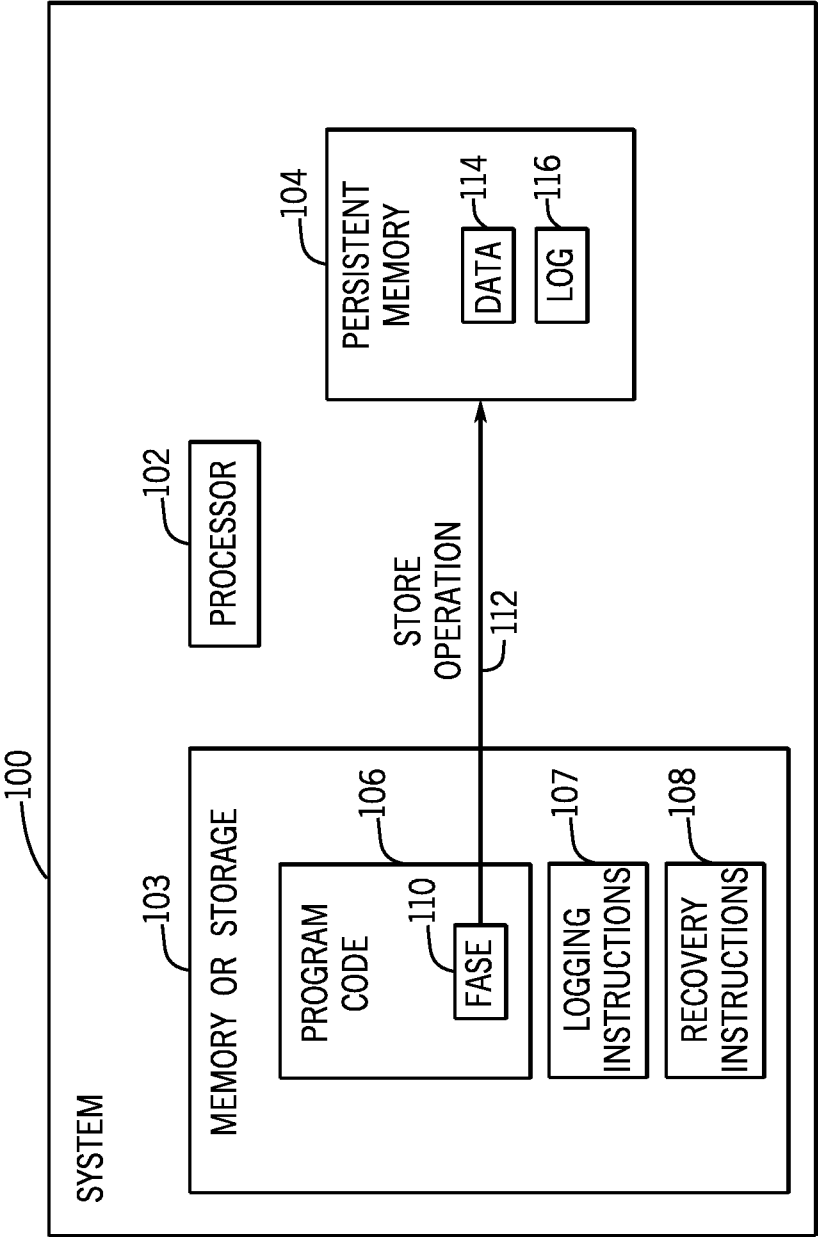


FIG. 1

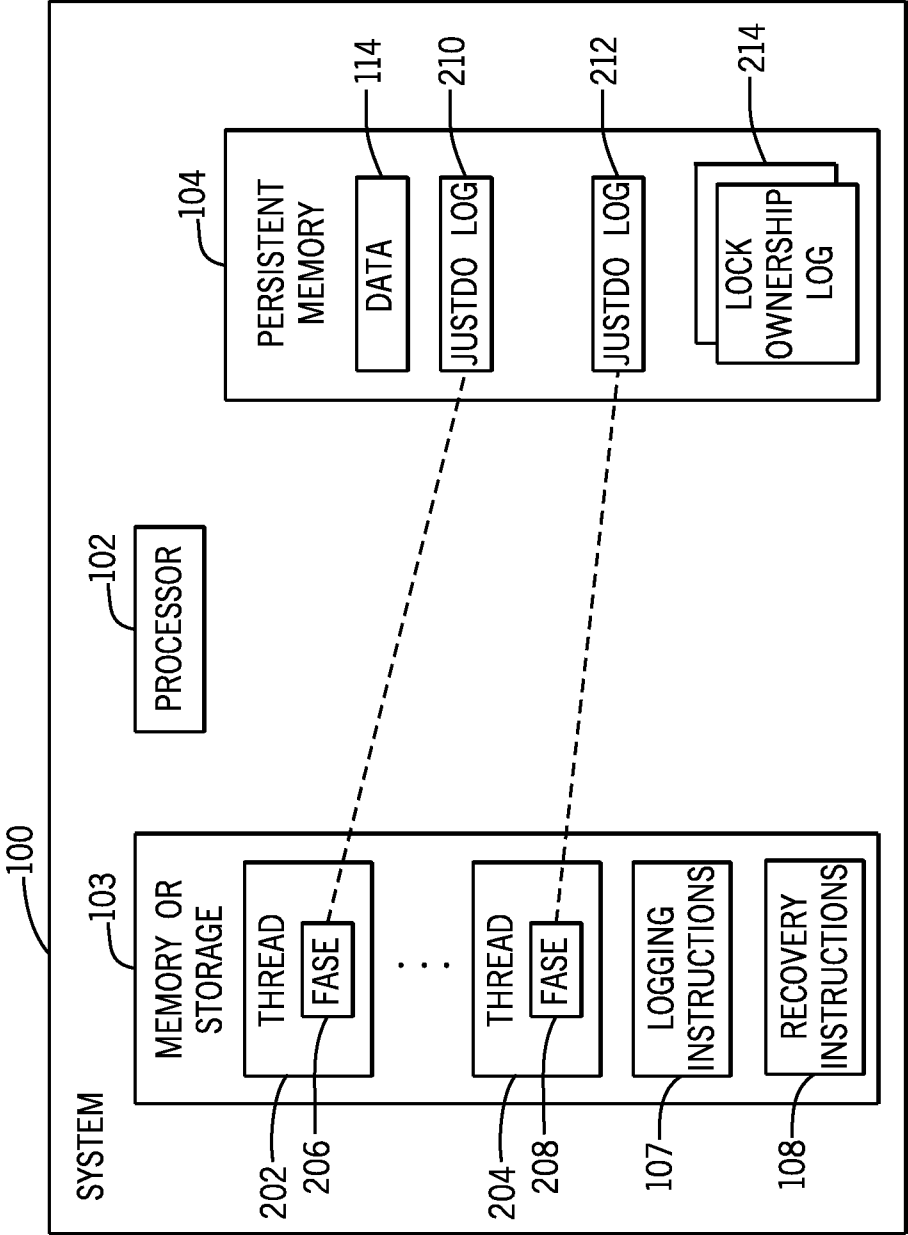


FIG. 2



FIG. 3B

FIG. 3A

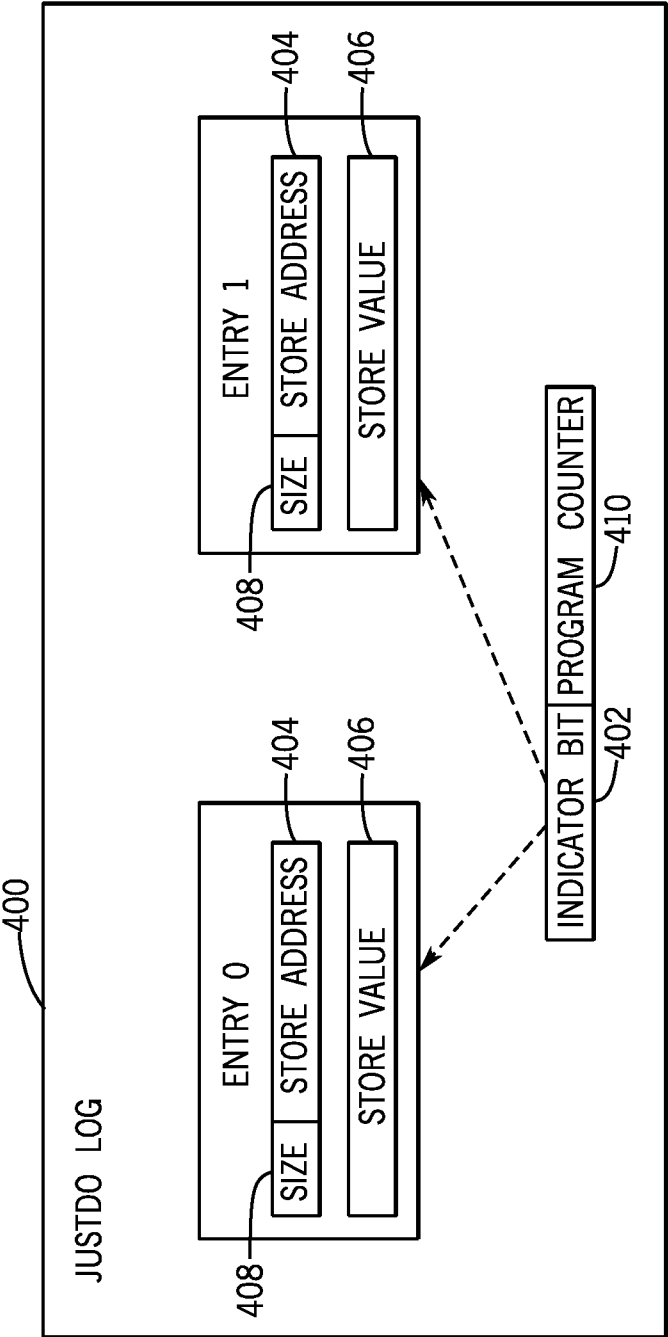


FIG. 4

1: A=X+5
2: B=Y+5

HIGH-LEVEL
PROGRAM
CODE

A1: LDR r1, X
A2: ADD r2, r1, 5
A3: STR r2, A
A4: LDR r3, Y
A5: ADD r4, r3, 5
A6: STR r4, B

REGULAR ASSEMBLY CODE

J1: LDR r1, X
J2: ADD r2, r1, 5
J3: JD L, <A, r2, J4>
J4: STR r2, A
J5: LDR r3, Y
J6: ADD r4, r3, 5
J7: JD L, <B, r4, J8>
J8: STR r4, B

JUSTDO ASSEMBLY CODE

FIG. 5A

FIG. 5B

FIG. 5C

6 / 7

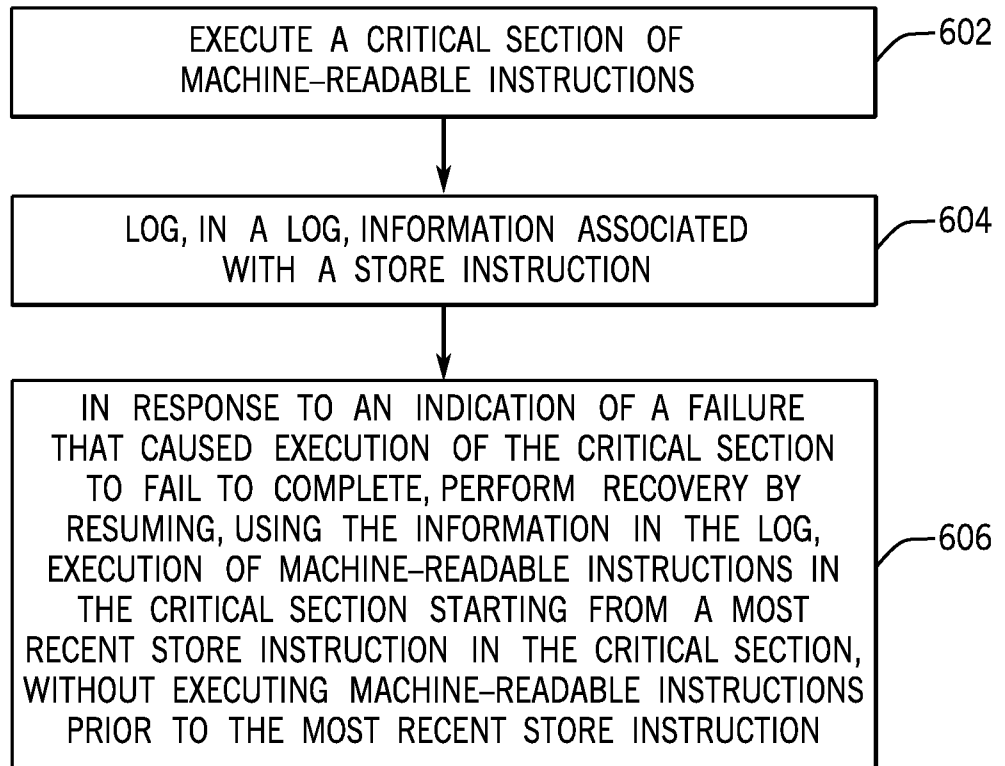


FIG. 6

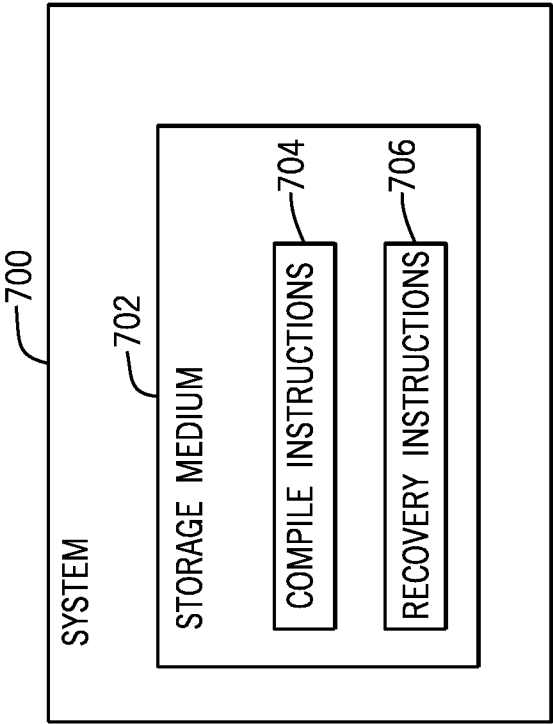


FIG. 7

A. CLASSIFICATION OF SUBJECT MATTER**G06F 11/16(2006.01)i, G06F 12/02(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)

G06F 11/16; G06F 3/06; G06F 17/30; G06F 12/02

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

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

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

eKOMPASS(KIPO internal) & Keywords: failure, recovery, critical section, program code, log, most recent, store instruction, persistent memory, and similar terms.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2014-0281269 A1 (HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P.) 18 September 2014 See paragraphs [0008]–[0009], [0014], and [0035]–[0036]; claims 1 and 4–5; and figures 1 and 4–5.	1–15
A	FAISAL NAWAB et al., "Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience," HP Laboratories, HPL-2014-70, 16 December 2014 (http://www.hpl.hp.com/techreports/2014/HPL-2014-70.pdf) See sections 3 and 4.2.	1–15
A	DHRUVA R. CHAKRABARTI et al., "Atlas: Leveraging Locks for Non-volatile Memory Consistency," HP Laboratories, HPL-2013-78R1, 21 February 2014 (http://www.hpl.hp.com/techreports/2013/HPL-2013-78.pdf) See sections 3.2 and 4.4–4.5.	1–15
A	REVECCA SMITH et al., "Surviving Peripheral Failures in Embedded Systems," Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15), pp. 125–137, 8–10 July 2015 See sections 3 and 4.2.1–4.2.2; and figures 1–2.	1–15

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

* Special categories of cited documents:

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

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

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

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

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

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

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

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

"&" document member of the same patent family

Date of the actual completion of the international search

25 July 2016 (25.07.2016)

Date of mailing of the international search report

26 July 2016 (26.07.2016)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea

Facsimile No. +82-42-481-8578

Authorized officer

NHO, Ji Myong

Telephone No. +82-42-481-8528



INTERNATIONAL SEARCH REPORT

International application No.

PCT/US2015/058446

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT		
Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2011-0276611 A1 (SURENDRA VERMA et al.) 10 November 2011 See paragraphs [0087]-[0105] and figure 11.	1-15

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2015/058446

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2014-0281269 A1	18/09/2014	None	
US 2011-0276611 A1	10/11/2011	AU 4580601 A	23/10/2001
		CN 100337233 C	12/09/2007
		CN 100445998 C	24/12/2008
		CN 1449530 A	15/10/2003
		CN 1746892 A	15/03/2006
		CN 1746893 A	15/03/2006
		CN 1746893 B	06/10/2010
		EP 1269353 A2	02/01/2003
		JP 2003-530646 A	14/10/2003
		JP 4219589 B2	04/02/2009
		US 2005-0120036 A1	02/06/2005
		US 2005-0120059 A1	02/06/2005
		US 2005-0138085 A1	23/06/2005
		US 2005-0149525 A1	07/07/2005
		US 2010-0042626 A1	18/02/2010
		US 2013-0325830 A1	05/12/2013
		US 6856993 B1	15/02/2005
		US 7257595 B2	14/08/2007
		US 7418463 B2	26/08/2008
		US 7512636 B2	31/03/2009
		US 7613698 B2	03/11/2009
		US 8010559 B2	30/08/2011
		US 8510336 B2	13/08/2013
		WO 01-77908 A2	18/10/2001
		WO 01-77908 A3	18/07/2002