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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 May 2015 (07.05.2015)

WIPO | PCT

(10) International Publication Number
WO 2015/065449 A1

(51) International Patent Classification:
*G06F 12/08* (2006.01) *G06F 11/30* (2006.01)

(21) International Application Number:
PCT/US2013/067841

(22) International Filing Date:
31 October 2013 (31.10.2013)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P.** [US/US]; 11445 Compaq Center Drive W., Houston, Texas 77070 (US).

(72) Inventors: **BOEHM, Hans**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US). **CHAKRABARTI, Dhruva**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US).

(74) Agents: **COLLINS, David** et al.; Hewlett-Packard Company, Intellectual Property Administration, 3404 E. Harmony Road, Mail Stop 35, 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, LT, 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, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

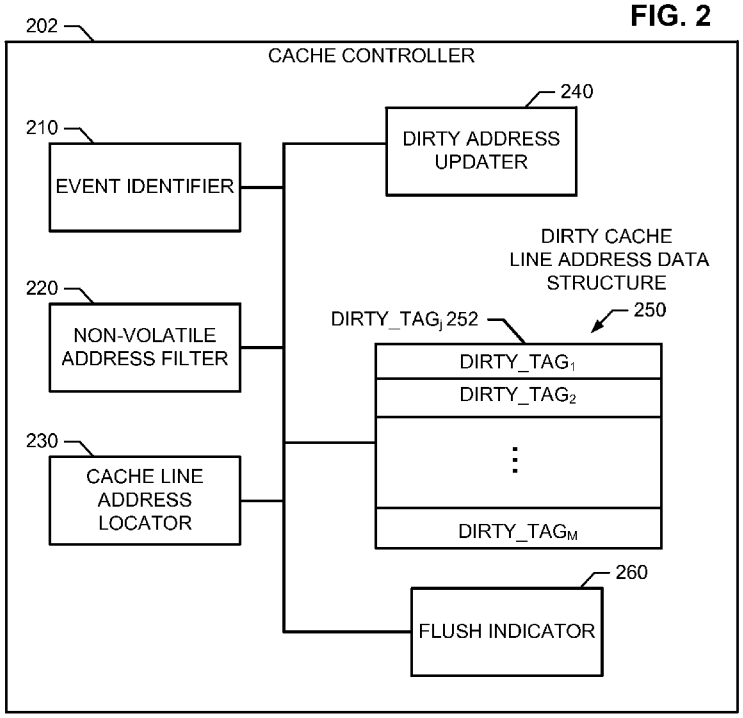
**Declarations under Rule 4.17**:

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

*[Continued on next page]*

(54) **Title**: CACHE CONTROLLER FOR NON-VOLATILE MEMORY

FIG. 2




(57) **Abstract**: Methods, apparatus, systems and articles of manufacture are disclosed to control a cache. An example method includes monitoring cache lines in a cache, the cache lines storing recently written data to the cache, the recently written data corresponding to main memory, comparing a total quantity of the cache lines to a threshold that is less than a cache line storage capacity of the cache, and causing a write back of at least one of the cache lines to the main memory when a store event causes the total quantity of the cache lines to satisfy the threshold.

WO 2015/065449 A1

**Published**:

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

# CACHE CONTROLLER FOR NON-VOLATILE MEMORY

## BACKGROUND

**[0001]** A central processing unit (CPU) cache locally stores data from a main memory (e.g., a volatile memory device, a non-volatile memory device, etc.). Caches are often implemented using volatile memory to store data when powered and lose memory state when power is removed. Accordingly, data in a cache is typically lost or deemed unreliable in the event of a system shut down or power failure.

**[0002]** A CPU uses cache to increase the speed of data operations because accessing data in cache is much faster than performing read and/or write operations in main memory. For example, a CPU retrieves data from non-volatile main memory and stores the retrieved data in a cache line (e.g., 64 bytes of data) of the cache when the data is to be used, edited, manipulated, etc. via read and/or write operations. If the data is modified in the cache, then it is eventually written back to main memory. After modifying the data in the cache line, but before the modified data is written back from the cache to the main memory, the cache line storing the modified data is considered a dirty cache line. Dirty cache lines and/or other cache lines may be written back (e.g., flushed) to main memory (e.g., a volatile memory, non-volatile memory, etc.) when instructed by the CPU. Dirty cache lines may also be written back without

an explicit software instruction to do so when the CPU requires the cache line space in the cache for other purposes.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0003]** FIG. 1 is a block diagram of an example processor system having an example cache controller implemented in accordance with the teachings of this disclosure.

**[0004]** FIG. 2 illustrates an example cache controller that may be used to implement the cache controller of FIG. 1.

**[0005]** FIG. 3 illustrates another example cache controller in communication with an example dirty address updater and an example dirty cache line address data structure implemented in accordance with the teachings of this disclosure.

**[0006]** FIG. 4 illustrates another example cache controller in communication with an example dirty address updater, an example dirty cache line address data structure, and an example instruction trapper implemented in accordance with the teachings of this disclosure.

**[0007]** FIG. 5 is a flowchart representative of example machine readable instructions that may be executed to implement the example cache controller of FIG. 2 and/or the example cache controller and/or the example dirty address updater of FIG. 3.

**[0008]** FIG. 6 is a flowchart representative of example machine readable instructions that may be executed to implement the example cache controller of FIG. 2 and/or the example cache controller and the example dirty address updater of FIGS. 3 and/or 4.

**[0009]** FIG. 7 is a flowchart representative of example machine readable instructions that may be executed to implement the example instruction trapper of FIG. 4.

## DETAILED DESCRIPTION

**[0010]** Example methods, apparatus, and articles of manufacture disclosed herein may be used to control a cache of a processor system. Examples disclosed herein may be used to control a quantity of cache lines that may be used to store data corresponding to non-volatile memory and to control a quantity of dirty cache lines storing modified data to be written back to the non-volatile memory.

**[0011]** Some examples disclosed herein involve monitoring cache lines storing recently written data to a cache, and comparing a total quantity of the cache lines to a threshold that is less than a cache line storage capacity of a cache. Examples disclosed herein further involve causing a write back of one or more of the dirty cache lines to the non-volatile memory when a store event causes the total quantity of the dirty cache lines to satisfy the threshold. In some examples, the threshold corresponds to a capacity of a data structure storing dirty cache line addresses (e.g., a number of entries in a hash table). In some examples, the threshold corresponds to a cache line capacity of a cache way of the cache. In such examples, the cache way is designated to store cache lines corresponding to only a non-volatile memory (e.g., a non-volatile memory of main memory).

**[0012]** In some examples disclosed herein utilizing a particular type of data structure, such as a hash table, a storage location in the hash table depends on a value of the cache line address to be tracked in the data structure. For example, multiple cache line addresses may be designated to be stored in a single location of the hash depending on their corresponding values. In such examples, a monitored threshold is satisfied for a particular store event when a newly monitored cache line address is to replace a previously monitored cache line address in a particular hash table location despite other empty locations being available. In some examples, each entry in the hash table will store only a particular cache line address or particular cache line addresses. In such examples, when the hash table entry is filled with a dirty cache line address but is to be used to store a newly identified dirty cache line address, the

threshold is met because that hash line entry may only store one dirty cache line address at a time.

**[0013]** In some examples, when a store event creates a new dirty cache line, the cache line address corresponding to the newly created dirty cache line is added to the data structure. If the newly added dirty cache line address causes a number of dirty cache line addresses to meet or exceed the threshold, a dirty cache line is selected to be flushed (i.e., written back to non-volatile memory) and, the dirty cache line address corresponding to the selected dirty cache line is removed from the data structure.

**[0014]** In some examples, when a cache line address of a store event corresponds to an already dirty cache line (e.g., a cache line previously modified, but not yet written back to main memory), the store event does not create a new dirty cache line, and, thus, does not increment the number of dirty cache line addresses closer to threshold (e.g., a capacity limit of the data structure). For example, when an entry of a dirty cache line address data structure (e.g., a hash entry of a hash table) is already storing a dirty cache line address matching an address of a current store event, an address of the current store event does not need to be newly added to the dirty cache line address data structure.

**[0015]** In some examples, cache lines may be flushed as a background process. Some example methods involve selecting a dirty cache line to be written back to main memory using the background process before a total quantity of cache lines satisfies a threshold. In such examples, the write back is performed without being based on a store event.

**[0016]** Traditionally, for systems having a volatile main memory, caches are made as large as possible, subject to latency constraints, area constraints (on a device, such as an integrated circuit), and power constraints. A larger cache can often improve performance because processors are able to perform faster data accesses on more data stored locally in the cache. However, for systems having a non-volatile main memory, caches are frequently flushed to persist updated data in the non-volatile main memory. In some instances, caches are so large that flushing the caches (i.e., writing cache lines back to

memory) uses many resources and causes so much latency that the benefits of having such large caches are diminished. Example methods, apparatus, and articles of manufacture disclosed herein control the number of dirty cache lines that are held in a cache and/or the number of cache lines corresponding to a non-volatile memory device to improve and/or reduce latencies for cache flushes and/or to efficiently perform cache line write backs. For example, the number of such cache lines is limited to be less than a threshold corresponding to a maximum capacity of a data structure (e.g., a hash table) storing dirty cache line addresses. In this manner, flushing a threshold number of cache lines that is less than a cache line storage capacity of a cache requires less time than flushing the entire cache.

**[0017]** As used herein, an address may be a virtual address, a physical address, and/or a physical location within a data structure. As used herein, a write back is defined as an event in which a cache line of a cache is written back to a main memory (i.e., the main memory location of the data is updated with the current version of the data that is located in the cache). As used herein, flushing a cache involves writing back data from the cache to a main memory device. In some examples, after data is written back during a cache line flush, the data is persisted in the cache, while in other examples the written back data is removed from the cache.

**[0018]** FIG. 1 is a block diagram of an example processor system 100 including an example cache controller 110 implemented in accordance with the teachings of this disclosure. The example processor system 100 also includes an example cache 120, an example processor core 130 (or a central processing unit (CPU)), an example memory controller 132, an example volatile memory 140, and an example non-volatile memory 150. An example memory bus 170 facilitates communication between the cache 120, the memory controller 132, the volatile memory 140, and the non-volatile memory 150. The processor system 100 may be, for example, a server, a personal computer, a mobile device (e.g., a cell phone, a smart phone, a tablet such as an iPad™), a personal digital assistant (PDA), an Internet appliance, a DVD player, a CD player, a digital video recorder, a Blu-ray player, a gaming console, a personal

video recorder, a set top box, or any other type of computing device. The processor core 130 of the illustrated example is hardware. For example, the processor core 130 can be implemented by one or more integrated circuits, logic circuits, microprocessors or controllers from any desired family or manufacturer.

**[0019]** The example volatile memory 140 of FIG. 1 is any volatile memory storage device that stores data when powered, but loses memory state when power is removed. For example, the volatile memory 140 may be implemented by Synchronous Dynamic Random Access Memory (SDRAM), Dynamic Random Access Memory (DRAM), Static Random Access Memory (SRAM), RAMBUS Dynamic Random Access Memory (RDRAM) and/or any other type of volatile memory. The non-volatile memory 150 is any non-volatile memory storage device (e.g., phase-change memory, memristor memory, etc.) that is capable of storing data when powered and when not powered.

**[0020]** The example cache 120 is a local storage circuit that may be collocated on a same device 175 (e.g., a semiconductor chip) as the example processor core 130, the example memory controller 132, and/or the example cache controller 110. In the illustrated example of FIG. 1, the processor core 130 can perform faster read and/or write operations when accessing data in the cache 120 than when accessing data in the volatile memory 140 and/or in the non-volatile memory 150 via the memory bus 170. Accordingly, the example cache controller 110, the example processor core 130, and/or the example memory controller 132 may load data from the volatile memory 140 and/or the non-volatile memory 150 into the cache 120 so that the processor core 130 can access the data relatively faster using the cache 120. Different techniques may be used to determine how long the data remains in the cache 120. For example, the data may remain in the cache 120 as long as the processor core 130 is accessing it frequently. When data is modified in the cache 120, write back operations are used to perform the same modifications on the corresponding data in the volatile memory 140 and/or the non-volatile memory 150 so that the data in the cache 120 and corresponding data in the volatile memory 140 or corresponding data in the non-volatile memory 150 are the

same. Without such write back operations, errors or system failures may occur as the storage devices 140 and 150 would not have an accurate and/or current version of the most recent data updated in the cache 120.

**[0021]** During operation, the example cache 120 may store both dirty cache lines and clean cache lines that correspond to data in the volatile memory 140 and/or the non-volatile memory 150. A dirty cache line is a cache line 122 of the example cache 120 that stores data (e.g., $DATA_1$, $DATA_2$, $DATA_N$) that has been modified since the time it was read into the cache 120 from the volatile memory 140 and/or the non-volatile memory 150. A clean cache line stores data (e.g., $DATA_1$, $DATA_2$, $DATA_N$) that has not been modified since the time it was read into the cache 120 from the volatile memory 140 and/or the non-volatile memory 150 (e.g., the main memory).

**[0022]** The example cache 120 of FIG. 1 has a storage capacity of N cache lines 122. Each cache line 122 includes an example valid bit ($V_i$) 123, an example dirty bit ($D_i$) 124, an example memory address ($TAG_i$) 126, and example cache line data ($DATA_i$) 128 (e.g., 64 bytes of data or any other data size). The valid bit ($V_i$) 123 indicates that corresponding cache line data ($DATA_i$) 128 is currently used to store data read into the cache 120 from a memory address ($TAG_i$) 126 of the volatile memory 140 and/or from a memory address ($TAG_i$) 126 of the non-volatile memory 150. The dirty bit ($D_i$) 124 indicates that corresponding data ($DATA_i$) 128 has been modified and that the corresponding data in the volatile memory 140 and/or in the non-volatile memory 150 at a memory address ($TAG_i$) 126 may be out of date (e.g., needs to be updated with the modified data ($DATA_i$) 128). In some examples, the cache 120 may be divided into cache ways. In such examples, each of the cache ways may store a limited number of the cache lines 122 (e.g., 128 cache lines). In some such examples, the cache controller 110 may control the type of cache lines that are stored in each cache way. For example, the cache controller 110 may control the cache such that one cache way of the cache 120 is used for data corresponding only to the non-volatile memory 150.

**[0023]** As disclosed herein, the cache controller 110 of FIG. 1 may be implemented via hardware, software, and/or firmware. The example cache

controller 110 controls the amount and/or type of data that is stored in the cache 120 in accordance with this disclosure. More specifically, examples disclosed herein enable the cache controller 110 to control (e.g., limit) the number of cache lines that can be stored in the cache 120 (e.g., the number of cache lines from the non-volatile memory 150). The example cache controller 110 also controls (e.g., limits) the number of clean (i.e., unmodified) cache lines and the number of dirty cache lines in the cache 120 that correspond to the volatile memory 140 and/or the non-volatile memory 150.

**[0024]** The processor platform 100 of the illustrated example of FIG. 1 also includes an interface circuit 180. The interface circuit 180 may be implemented by any type of interface standard, such as an Ethernet interface, a universal serial bus (USB), and/or a Peripheral Component Interconnect (PCI) express interface.

**[0025]** In the illustrated example, one or more input devices 182 are connected to the interface circuit 180. The input device(s) 182 permit(s) a user to enter data and commands into the processor core 130. The input device(s) can be implemented by, for example, an audio sensor, a microphone, a camera (still or video), a keyboard, a button, a mouse, a touchscreen, a track-pad, a trackball, isopoint and/or a voice recognition system.

**[0026]** One or more output devices 184 are also connected to the interface circuit 180 of the illustrated example. The output devices 184 can be implemented, for example, by display devices (e.g., a light emitting diode (LED), an organic light emitting diode (OLED), a liquid crystal display, a cathode ray tube display (CRT), a touchscreen, a tactile output device, a light emitting diode (LED), a printer and/or speakers). The interface circuit 180 of the illustrated example, thus, typically includes a graphics driver card, a graphics driver chip or a graphics driver processor.

**[0027]** The interface circuit 180 of the illustrated example also includes a communication device such as a transmitter, a receiver, a transceiver, a modem and/or network interface card to facilitate exchange of data with external machines (e.g., computing devices of any kind) via a network 186 (e.g., an

Ethernet connection, a digital subscriber line (DSL), a telephone line, coaxial cable, a cellular telephone system, etc.).

**[0028]** The processor platform 100 of the illustrated example also includes one or more mass storage devices 190 for storing software and/or data. Examples of such mass storage devices 190 include floppy disk drives, hard drive disks, compact disk drives, Blu-ray disk drives, RAID systems, and digital versatile disk (DVD) drives. In some examples, the mass storage devices 190 may be implemented using the non-volatile memory 150.

**[0029]** Examples disclosed herein may be implemented using one or more different hardware-based and/or software/firmware-based techniques. For example, FIG. 2 discloses example hardware-based implementations to track and control the number of dirty cache lines. FIGS. 3 and 4 illustrate software-based implementations to track and control a number of dirty cache lines.

**[0030]** FIG. 2 is a block diagram of an example hardware representation of a cache controller 202, which may be used to implement the cache controller 110 of FIG. 1. The example cache controller 202 of the illustrated example of FIG. 2 includes an example event identifier 210, an example non-volatile address filter 220, an example cache line address locator 230, an example dirty address updater 240, an example dirty cache line address data structure 250, and an example flush indicator 260. In the illustrated example of FIG. 2, the example cache controller 202 is implemented using hardware. For example, the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example dirty address updater 240, the example dirty cache line address data structure 250, the example flush indicator 260, and/or more generally, the example cache controller 202 can be implemented by one or more integrated circuits, logic circuits, microprocessors or controllers from any semiconductor device manufacturer and/or any other semi-conductor based devices made by the semiconductor device manufacturer.

**[0031]** The example cache controller 202 of FIG. 2 controls the persistence of the data ($DATA_i$) 128 in the cache 120 of FIG. 1 using the dirty

cache line address data structure 250. In the illustrated example, the cache controller 202 uses the dirty cache line address data structure 250 to control the quantity of cache lines 122 in the cache 120 that are allowed to store dirty data at any given time. For example, the cache controller 202 may limit the number of dirty cache lines 122 that are stored in the cache 120 for the non-volatile memory 150 to a number of entries corresponding to a maximum capacity M of the cache line address data structure 250, or some other threshold (e.g., a cache line capacity of a single cache way or multiple selected cache ways). For example, the cache controller 202 may use a capacity limit M (e.g., one cache line, 64 cache lines, 128 cache lines, etc.) of the dirty cache line address data structure 250 to limit the number of cache lines 122 in the cache 120 that store dirty data and/or that correspond to a location in the non-volatile memory 150. In some examples, the threshold number of dirty cache lines corresponding to the non-volatile memory 150 that are allowed to persist in the cache 120 at any given time is defined by a cache line capacity of a single cache way (or any other number of cache ways) of the cache 120. Some examples disclosed herein may limit the number of dirty cache lines corresponding to the non-volatile memory 150 to not exceed one cache way or multiple selected cache ways of the cache 120. For example, if only a single cache way having a 128 cache line capacity is selected to store dirty cache lines corresponding to the non-volatile memory 150, the capacity limit M of the dirty cache line address data structure 250 is 128 entries (or hashes). In some examples, for a particular store event, a relative threshold may be triggered when the store event identifies a dirty cache line address that is to be stored in a particular location of the dirty cache line address data structure 250. In such examples, the relative threshold is satisfied for a particular store event when the newly tracked cache line address replaces a previously tracked cache line address at the particular location.

**[0032]** In some examples, a threshold value M' that is less than the capacity limit of M entries in the dirty cache line address data structure 250 may be implemented. In such examples, the threshold value M' may be a user-selectable threshold value and/or automatically adjustable by the cache

controller 202, the processor core 130, the memory controller 132, etc. In some examples, the capacity limit M of the dirty cache line address data structure 250 is used when the dirty cache line address data structure 250 is implemented in hardware, and the threshold value M' is used when the dirty cache line address data structure 250 is implemented in software (e.g., the dirty cache line address data structure 350 of FIGS. 3 and/or 4). As used herein, the capacity limit M and the threshold value M' may be used interchangeably to control the number of dirty cache line addresses that are tracked in the dirty cache line address data structure 250.

**[0033]** The example dirty cache line address data structures 250 of FIG. 2 may be a hash table or other suitable type of data structure implemented in hardware. In some examples, the dirty cache line address data structure 250 may be located in the cache 120 or any other suitable memory subsystem. In the illustrated example of FIG. 2, the dirty cache line address data structure 250 stores dirty cache line addresses ($DIRTY_TAG_j$) 252 (e.g., address values, physical locations, etc.). In some examples, the dirty cache line addresses ($DIRTY_TAG_j$) 252 correspond to ones of the cache line addresses ($TAG_i$) 126 (FIG. 1) that may be (or are) associated with dirty cache lines corresponding to locations in the non-volatile memory 150. In some such examples, the cache controller 110 uses the dirty cache line address data structure 250 to limit the number of dirty cache line addresses ($DIRTY_TAG_j$) 252 corresponding to locations in the non-volatile memory 150 to a quantity of M dirty cache lines. In such examples, the quantity of M dirty cache lines corresponds to the capacity limit M (or a threshold value M') of the dirty cache line address data structure 250. In some examples, the capacity limit M of the dirty cache line address data structure 250 is equal to the number of cache lines in a cache way of the cache 120. In the illustrated example, the dirty cache line addresses ($DIRTY_TAG_j$) 252 are indicated as corresponding to ones of the cache line addresses ($TAG_i$) 126 by having the same address values, by referring to a same virtual address, and/or by having some other similar mapping or association to a same address or location of a virtual memory space and/or a physical memory space.

**[0034]** In FIG. 2, the example event identifier 210 monitors data access events (e.g., read operations, write operations, etc.) performed in the cache 120. In some examples, the event identifier 210 monitors the cache 120 for store events. As used herein, a store event corresponds to a write event in which data ($DATA_i$) 128 in the cache 120 is modified and is to be written back to the volatile memory 140 and/or to the non-volatile memory 150. In the illustrated example, upon identification of a store event, the event identifier 210 notifies the non-volatile address filter 220 of the store event to analyze the location and/or address indicated in the store event. In the illustrated example, the non-volatile address filter 220, the cache line address locator 230, and the dirty address updater 240 use the analyzed location and/or address indicated in the store event to determine whether the store event requires an update to the dirty cache line address data structure 250. For example, the dirty address updater 240 determines if the store event indicates a new cache line address ($TAG_i$) 126 that is not currently being tracked (e.g., stored) in the dirty cache line address data structure 250. For example, if the cache line address ($TAG_i$) 126 indicated by the store event is not already stored (e.g., in a hash entry) in the dirty cache line address data structure 250, the indicated cache line address ($TAG_i$) 126 is not currently being tracked as a corresponding dirty cache line. In such instances, the dirty address updater 250 determines that it needs to track the cache line address ($TAG_i$) 126 by creating a new dirty cache line address ($DIRTY_TAG_j$) 252 corresponding to the indicated cache line address ($TAG_i$) 126. The dirty address updater 240 is also configured to determine when a dirty cache line address ($DIRTY_TAG_j$) 252 is to be removed from the dirty cache line address data structure 250 based on whether a flush of a dirty cache line 122 in the cache 120 is to occur.

**[0035]** The example non-volatile address filter 220 of FIG. 2 determines whether a store event corresponds to an address (and/or a location) in the volatile memory 140 and/or corresponds to an address (and/or a location) in the non-volatile memory 150. The cache line address locator 230 of the illustrated example determines cache line addresses ($TAG_i$) 126 corresponding to addresses indicated in store events. The example cache line address locator

230 provides the dirty address updater 240 with cache line address(es) ($TAG_i$) 126 corresponding to the store events.

**[0036]** The example dirty address updater 240 of FIG. 2 determines whether a cache line address ($TAG_i$) 126 associated with a store event is already stored as a dirty cache line address ($DIRTY_TAG_j$) 252 in the cache line address data structure 250. In this manner, the example dirty address updater 240 facilitates avoiding any unnecessary increases in the number of dirty cache line addresses ($DIRTY_TAG_j$) 252 with duplicate entries. Preventing duplicate entries keeps the dirty cache line address data structure 250 relatively compact and avoids performing multiple write backs to the non-volatile memory 150 for a same cache line address ($TAG_i$) 126 that could otherwise occur if the addresses were stored in duplicate in the dirty cache line address data structure 250. For example, when a store event modifies data ($DATA_1$) 128 corresponding to cache line address ($TAG_i$) 126, and when the modified data ($DATA_1$) 128 is to be written back to the non-volatile memory 150, the non-volatile address filter 220 determines if a dirty cache line address ($DIRTY_TAG_j$) 252 matches and/or corresponds to the cache line address ($TAG_1$) 126 (e.g., $DIRTY_TAG_2$ matches and/or corresponds to $TAG_1$) to determine if the cache line address ($TAG_1$) 126 is to be added to the dirty cache line address data structure 250 for tracking. If the cache line address ($TAG_1$) 126 of the example store event does not match one of the dirty cache line addresses ($DIRTY_TAG_j$) 252 (e.g., none of the dirty cache line addresses ($DIRTY_TAG_j$) 252 match or correspond to the example cache line address ($TAG_1$) 126), the dirty address updater 240 updates the dirty cache line address data structure 250 to include a new reference (e.g., a newly added dirty cache line address ($DIRTY_TAG_j$) 252) corresponding to the cache line address ($TAG_1$) 126 indicated in the store event. On the other hand, if one of the dirty cache line addresses ($DIRTY_TAG_j$) 252 does correspond to the cache line address ($TAG_1$) 126 indicated in the store event, the dirty address updater 240 does not update the dirty cache line address data structure 250 because the cache line address ($TAG_1$) 126 is already tracked as dirty in the dirty cache line address data structure 250. Thus, the cache controller 202 continues to track modified ones of the cache lines 122 without increasing the

number of dirty cache line addresses (DIRTY_TAG$_j$) 252 toward the threshold that causes the flush indicator 260 to instruct the memory controller 132 or other flush controller to write back and/or clear corresponding data (DATA$_1$) 128 from the cache 120 corresponding to the cache line address (TAG$_1$) 126 indicated in the store event.

**[0037]** The example dirty address updater 240 updates the dirty cache line address data structure 250 by first determining if the dirty cache line address data structure 250 has reached a capacity limit of M (or threshold of M') dirty cache line addresses. If there is less than the capacity limit of M dirty cache line addresses in the dirty cache line address data structure 250, the dirty address updater 240 adds the cache line address (TAG$_i$) 126 corresponding to the store event to the dirty cache line address data structure 250 as disclosed above. If the number of dirty cache line addresses has reached the capacity limit of M dirty cache line addresses, the dirty address updater 240 determines one or more dirty cache line address(es) (DIRTY_TAG$_j$) 252 to be removed from the dirty cache line address data structure 250. In examples disclosed herein, when reference is made to removing a (DIRTY_TAG$_j$) 252 from the dirty cache line address data structure 250, such removal is done in connection with the cache controller 110 causing a write back of its corresponding cache line 122.

**[0038]** In some examples, the dirty cache line address data structure 250 is a hash table in which locations in the dirty cache line address data structure 250 are designated for a particular address or particular addresses. In such examples, no two cache line addresses can be tracked in a same location in the dirty cache line address data structure 250 at the same time. Accordingly, in such examples, the threshold for a corresponding store event is satisfied when the store event indicates a first dirty cache line address (DIRTY_TAG$_1$) 252 that is to be stored at a particular location of the hash table that is already storing a second dirty cache line address (DIRTY_TAG$_2$) 252 regardless of other locations being empty in the dirty cache line address data structure 250. In such examples, the threshold is satisfied for the corresponding store event because only one dirty cache line address can be stored in each storage location of a hash table (i.e., the dirty cache line address data structure 250).

Accordingly, in such examples, the threshold is relative to a particular address indicated in the store event. Thus, once the example threshold is satisfied, a write back of a cache line address ($TAG_i$) 126 corresponding to the dirty cache line address ($DIRTY_TAG_j$) 252 that is removed from the dirty cache line address data structure 250 is written back to the non-volatile memory 150 (or main memory).

**[0039]** In the illustrated example, the capacity limit M (or the threshold M') is less than the cache line storage capacity N (i.e., $M < N$). That is, the cache controller 110 artificially suppresses or limits the number of dirty cache lines in the cache 120 to be less than the cache space that is available for storing dirty cache lines. In some examples, a threshold number of dirty cache lines may be user selected and/or automatically adjusted by the cache controller 110 and/or processor core 130 (FIG. 1) based on settings of the processor system 100 to a threshold value M'. For example, when the processor core 130 determines that cache flushes have become less frequent during a particular time period, the processor core 130 may increase the threshold number M' of dirty cache lines because the processor core 130 has extra time/resources to account for longer write back operations to the non-volatile memory 150.

**[0040]** In the illustrated example, the dirty address updater 240 may remove a dirty cache line address ($DIRTY_TAG_j$) 252 and cause a write back of its corresponding cache line 122 using any suitable type of hierarchical cache line removal technique. For example, the dirty address updater 240 may use a first-in, first-out (FIFO) approach in which the first dirty address that was added to the dirty cache line address data structure 250 is the first to be removed. As another example, the dirty address updater 240 may use a least-frequently used cache line removal approach, in which the cache line that is the least frequently used and/or accessed is removed from the dirty cache line address data structure 250. In yet another example, the dirty address updater 240 may use least recently used (LRU) cache line removal in which the cache line that has not been accessed in the longest amount of time is removed from the dirty cache line address data structure 250. Once the dirty cache line address ($DIRTY_TAG_j$) 252 corresponding to the determined cache line address ($TAG_i$)

126 is identified, the dirty address updater 240 removes the appropriate dirty cache line address ($DIRTY_TAG_j$) 252 and notifies the flush indicator 260 that the cache line 122 of the determined cache line address ($TAG_i$) 126 is to be written back to the non-volatile memory 150 and cleared from the cache 120 (i.e., flushed). The flush indicator 260 may then provide a flush instruction with the cache line address ($TAG_i$) 126 to the memory controller 132 of the processor core 130 or other cache flush controller to flush the corresponding cache line 122.

**[0041]** While an example hardware representation of implementing the cache controller 110 of FIG. 1 is illustrated as the example cache controller 202 of FIG. 2, one or more of the elements, processes and/or devices of the example cache controller 202 illustrated in FIG. 2 may be combined, divided, re-arranged, omitted, eliminated and/or implemented in any other way. Further, although the example cache controller 202 of FIG. 2 is implemented via hardware, the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example dirty address updater 240, the example dirty cache line address data structure 250, and the example flush indicator 260 and/or, more generally, the example cache controller 202 of FIG. 2 may be implemented by hardware, software, firmware and/or any combination of hardware, software and/or firmware. Thus, for example, any of the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example dirty address updater 240, the example dirty cache line address data structure 250, and the example flush indicator 260 and/or, more generally, the example cache controller 202 could be implemented by one or more analog or digital circuit(s), logic circuits, programmable processor(s), application specific integrated circuit(s) (ASIC(s)), programmable logic device(s) (PLD(s)) and/or field programmable logic device(s) (FPLD(s)). When reading any of the apparatus or system claims of this patent to cover a purely software and/or firmware implementation, at least one of the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example dirty address updater 240, the example dirty cache line

address data structure 250, and/or the example flush indicator 260 is/are hereby expressly defined to include a tangible computer readable storage device or storage disk such as a memory, a digital versatile disk (DVD), a compact disk (CD), a Blu-ray disk, etc. storing the software and/or firmware. Further still, the example cache controller 202 of FIG. 2 may include one or more elements, processes and/or devices in addition to, or instead of, those illustrated in FIG. 2, and/or may include more than one of any or all of the illustrated elements, processes and devices.

**[0042]** FIG. 3 is a block diagram showing an example software representation of an example dirty address updater 340 and a software representation of an example dirty cache line address data structure 350 in communication with another example hardware representation of a cache controller 302. In the illustrated example of FIG. 3, the cache controller 302 is used to implement the cache controller 110 of FIG. 1. The example cache controller 302 of the illustrated example of FIG. 3 includes an example event identifier 210, an example non-volatile address filter 220, an example cache line address locator 230, and an example flush indicator 260. In the illustrated example of FIG. 3, the example cache controller 302 is implemented using hardware. For example, the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example flush indicator 260, and/or more generally, the example cache controller 302 can be implemented by one or more integrated circuits, logic circuits, microprocessors or controllers from any desired family or manufacturer.

**[0043]** The example cache controller 302 of FIG. 3 controls the cache 120 of FIG. 1 using the example dirty address updater 340 and the example dirty cache line address data structure 350 of FIG. 3. In the illustrated example, the dirty address updater 340 and the dirty cache line address data structure 350 are implemented using software and/or firmware. Accordingly, the example dirty address updater 340 and/or the example dirty cache line address data structure 350 of FIG. 3 are implemented using instructions stored on a storage device or storage disc, such as, the non-volatile memory 150 and/or the mass storage device 190 of FIG. 1.

**[0044]** In the illustrated example of FIG. 3, the example cache controller 302, the example dirty address updater 340, and the dirty cache line address data structure 350 perform similar operations and/or have similar functionality as the cache controller 202 of FIG. 2. For example, the event identifier 210 performs similar operations and/or has similar functionality as the event identifier 210 of FIG. 2, the example non-volatile memory address 220 performs similar operations and/or has similar functionality as the example non-volatile memory address 220 of FIG. 2, the example cache line address locator 230 performs similar operations and/or has similar functionality as the cache line address locator 230 of FIG. 2, and/or the flush indicator 260 performs similar operations and/or has similar functionality as the flush indicator 260 of FIG. 2. In a similar manner, even though the example dirty address updater 340 of FIG. 3 and the example dirty cache line address database 350 of FIG. 3 are implemented using software and/or firmware, the example dirty address updater 340 performs similar operations and/or has similar functionality as the example address updater 240 of FIG. 2, and the example dirty cache line address data structure 350 performs similar operations and/or has similar functionality as the dirty cache line address data structure 250. In the interest of brevity, details of the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example flush indicator 260, the example dirty address updater 340, and the example dirty cache line address data structure 350 are not repeated here. Instead, the interested reader is referred to the above descriptions of the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example flush indicator 260, the example dirty address updater 240, and the example dirty cache line address data structure 250 of FIG. 2 for relevant details.

**[0045]** As disclosed herein, the dirty address updater 340 and the dirty cache line address data structure 350 of FIG. 3 are implemented in software and/or firmware. Accordingly, the example dirty address updater 340 and/or the example dirty cache line address data structure 350 may be compiled ahead-of-time prior to runtime to define a software/firmware function that facilitates

tracking dirty cache lines. In such examples, an instruction may be created so that a program can call into the function to track dirty cache lines. For purposes of this disclosure and ease of reference in the following, such instruction is referred to as a store_with_tracking instruction . Example operations that may be used to implement the store_with_tracking instruction are described below in connection with FIG. 5. The following is example pseudo code of an example store_with_tracking instruction that may be executed to perform the operations disclosed herein.

```
[LOC1]  store_with_tracking(addr, value)
        {
[LOC2]  cache_line_addr = high_order_bits (addr);
[LOC3]  h = hash(cache_line_addr) mod <length of may_be_dirty array>;
[LOC4]  if (may_be_dirty[h] !=0 && may_be_dirty[h] != cache_line_addr)
[LOC5]      asynchronously_flush_cache_line(may_be_dirty[h]);
[LOC6]  may_be_dirty[h] = cache_line_addr;
[LOC7]  *addr = value;
        }
```

**[0046]** Line of code [LOC 1] of the above example pseudo code is called each time store_with_tracking instruction is called. At line of code [LOC2] of the above example pseudo code, a variable cache_line_addr is set to the cache line address of the cache line, and bits corresponding to the byte addresses within the cache line are ignored or masked using a high_order_bits function. At line of code [LOC3], a variable h is set to a hash value of the cache line address. At line of code [LOC4], a may_be_dirty array (e.g., the dirty cache line address data structure 250, 350 of FIGS. 2 and 3) is checked to determine if the cache line address is already located in the may_be_dirty array (e.g., the dirty cache line address data structure 250, 350 of FIGS. 2 and 3) by determining that the hash entry is not equal to 0 and the hash entry is not equal to the cache line address. At line of code [LOC5] if cache_line_addr is not in may_be_dirty and the entry at h is not equal to zero, the cache line at entry h is flushed asynchronously relative to other scheduled flush operations. At line of code [LOC6], the may_be_dirty array (e.g., the dirty cache line address data structure 250, 350 of FIGS. 2 and 3) is updated (same value may be rewritten at entry h if

address was already in the may_be_dirty array). At line of code [LOC7] the store operation is performed.

**[0047]** In the illustrated example of FIG. 3, the store_with_tracking instruction is exposed or made available to software developers that want to take advantage of enhanced caching operations disclosed herein. Accordingly, when such example software developers desire to take advantage of the disclosed enhanced caching operations, the software developers recompile their legacy programs to replace standard store instructions with the new store_with_tracking instruction disclosed herein. In such examples, the cache controller 302, the dirty address updater 340 and the dirty cache line address data structure 350 are used to implement the operations necessary to execute the store_with_tracking instructions rather than standard store instructions during execution of the recompiled programs. Upon execution of the store_with_tracking instructions, the example cache controller 302, the example dirty address updater 340, and the example dirty address updater 350 perform operations disclosed herein to track dirty cache lines and limit the number of dirty cache lines that store data corresponding to the non-volatile memory 150.

**[0048]** While an example hardware representation of implementing the cache controller 110 of FIG. 1 using the cache controller 302 and example software-based implementations of the example dirty address updater 340 and the example dirty cache line address data structure 350 are illustrated in FIG. 3, one or more of the elements, processes and/or devices of the example cache controller 302, the example dirty address updater 340, and/or the dirty cache line address data structure 350 illustrated in FIG. 3 may be combined, divided, re-arranged, omitted, eliminated and/or implemented in any other way. Further, although the example cache controller 302 is implemented in hardware, the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example and the example flush indicator 260 and/or, more generally, the example cache controller 302 of FIG. 3, as well as the example dirty address updater 340 and/or the dirty cache line address data structure 350 may be implemented by hardware, software, firmware and/or any combination of hardware, software and/or firmware. Thus,

for example, any of the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, and the example flush indicator 260 and/or, more generally, the example cache controller 302, as well as, the example dirty address updater 340 and/or the dirty cache line address data structure 350, could be implemented by one or more analog or digital circuit(s), logic circuits, programmable processor(s), application specific integrated circuit(s) (ASIC(s)), programmable logic device(s) (PLD(s)) and/or field programmable logic device(s) (FPLD(s)). When reading any of the apparatus or system claims of this patent to cover a purely software and/or firmware implementation, at least one of the example event identifier 210, the example non-volatile address filter 220, the example cache line address locator 230, the example dirty address updater 340, the example dirty cache line address data structure 350, and/or the example flush indicator 260 is/are hereby expressly defined to include a tangible computer readable storage device or storage disk such as a memory, a digital versatile disk (DVD), a compact disk (CD), a Blu-ray disk, etc. storing the software and/or firmware. Further still, the example cache controller 302, the example dirty address updater 340, and/or the example dirty cache line address data structure 350 of FIG. 3 may include one or more elements, processes and/or devices in addition to, or instead of, those illustrated in FIG. 3, and/or may include more than one of any or all of the illustrated elements, processes and devices.

**[0049]** Example methods, apparatus, and articles of manufacture disclosed herein allow managing cache as disclosed herein without recompiling legacy software programs. Such examples involve mapping a read-only memory address space and a writable memory address space, and when a store event corresponds to the read-only memory address space, the store instruction can be replaced by alternate code that performs a store operation at a memory location corresponding to the writable memory address space.

**[0050]** FIG. 4 is a block diagram of an example instruction trapper 420 implemented using software and/or firmware to implement example cache control techniques disclosed herein. In the illustrated example, the instruction trapper 420 is in communication with an example hardware-based cache

controller 402, which may be used to implement the cache controller 110 of FIG. 1. The example of FIG. 4 also includes an example software-based dirty address updater 340 and an example software-based dirty cache line address data structure 350, and an example software based code modifier 454. The example cache controller 402 of the illustrated example of FIG. 4 includes an example event identifier 210, an example cache line address locator 230, and an example flush indicator 260. In the illustrated example, the example event identifier 210, the example cache line address locator 230, the example flush indicator 260, and/or more generally, the example cache controller 402 can be implemented by one or more integrated circuits, logic circuits, microprocessors or controllers from any desired family or manufacturer.

**[0051]** The example cache controller 402 of FIG. 4 controls the cache 120 of FIG. 1 using the example instruction trapper 420, the example dirty address updater 340, and the example dirty cache line address data structure 350 of FIG. 4. The dirty address updater 340 and the dirty cache line address data structure 350 are implemented via software and/or firmware. Accordingly, the example instruction trapper 420, the example dirty address updater 340 and/or the dirty cache line address data structure 350 of FIG. 4 are implemented via instructions stored on a storage device or storage disc, such as, the non-volatile memory 150 and/or the mass storage device 190 of FIG. 1.

**[0052]** In the illustrated example of FIG. 4, the example cache controller 402, the example dirty address updater 340, and the dirty cache line address data structure 350 perform similar operations and/or have similar functionality as the cache controller 202 of FIG. 2. For example, the event identifier 210 performs similar operations and/or has similar functionality as the event identifier 210 of FIG. 2, the example cache line address locator 230 performs similar operations and/or has similar functionality as the cache line address locator 230 of FIG. 2, and/or the flush indicator 260 performs similar operations and/or has similar functionality as the flush indicator 260 of FIG. 2. In a similar manner, even though the example dirty address updater 340 of FIG. 4 and the example dirty cache line address data structure 350 of FIG. 4 are implemented using software and/or firmware, the example dirty address updater 340

performs similar operations and/or has similar functionality as the example address updater 240 of FIG. 2, and the example dirty cache line address data structure 350 performs similar operations and/or has similar functionality as the dirty cache line address data structure 250 of FIG. 2. In the interest of brevity, details of the example event identifier 210, the example cache line address locator 230, the example flush indicator 260, the example dirty address updater 340, and the example dirty cache line address data structure 350 are not repeated here. Instead, the interested reader is referred to the above descriptions of the example event identifier 210, the example cache line address locator 230, the example flush indicator 260, the example dirty address updater 240, and the example dirty cache line address data structure 250 of FIG. 2 for relevant details.

**[0053]** The illustrated example of FIG. 4 may be used to provide software developers with access to the enhanced cache controlling techniques disclosed herein without requiring such software developers to recompile their legacy software to explicitly use any special instructions (e.g., a store_with_tracking instruction) that invoke the disclosed cache control techniques. That is, the software-based examples disclosed in connection with FIG. 4 enable legacy software programs 490 or software programs with standard 'store' instructions to advantageously use the store_with_tracking instruction without needing a software developer and/or an ahead-of-time software compiler to explicitly include the store_with_tracking instruction in programs. In such examples, the instruction trapper 420 of FIG. 4 performs a trapping technique to trap normal store operations to perform the store_with_tracking operations disclosed herein. Accordingly, in such examples, a software program 456 having standard store instructions can operate using enhanced cache performance using techniques disclosed herein when the instruction trapper 420 traps store events and replaces corresponding store instructions with new store_with_tracking instructions in the software program 456. For example, the example code modifier 454 modifies and replaces a store instruction during runtime with a store_with_tracking instruction so that operations of the store_with_tracking instruction provide the program with enhanced cache performance as disclosed

herein. In some examples, the replacement of the store instruction with the store_with_tracking instruction may be made in the binary code of a program executing in an operating system environment of the processor system 100.

**[0054]** FIG. 4 illustrates an example representation of an instruction trapper 420 implemented in software and/or firmware to determine when a store event is to be replaced with a store_with_tracking operation as disclosed herein. In the illustrated example, the instruction trapper 420 identifies store events corresponding to the non-volatile memory 150 using a trapping technique. In the illustrated example of FIG. 4, the instruction trapper 420 includes an example address mapper 260, an example trap identifier 470, and an example trap handler 480. The example instruction trapper 420 of FIG. 4 uses the example address mapper 465 to trap stores (e.g., via write fault detection) to a read-only address of the read-only mapping 490 of data from the non-volatile memory 150 to determine that an address (i.e., the read-only address) indicated in a store event is in the non-volatile memory 150. In addition, the instruction trapper 420 tracks a writable address and/or location in a writable mapping 492 that corresponds to the read-only address of the read-only mapping 490 by identifying the writable address in the writable mapping 492 that is to be tracked by the cache controller 402 as disclosed herein.

**[0055]** In the illustrated example of FIG. 4, the address mapper 465 maps data of the example non-volatile memory 150 to a virtual address space in hardware, software, firmware, etc. For example, the processor core 130 may identify corresponding addresses of data in the non-volatile memory 150 via the address mapper 465 by identifying an address and/or location in the address mapper 465 that points to an address and/or location of the corresponding data in the non-volatile memory 150. The example address mapper 465 maintains the read-only mapping 490 of the data that is stored in the non-volatile memory 150 in a read-only memory address space or a read-only virtual address space and the writable mapping 492 of the data that is stored in non-volatile memory 150 in a writable memory address space or writable virtual address space. The read-only mapping 490 is read-only in that only read operations are allowed for the data in the non-volatile memory 150 that is represented in the read-only

mapping 490. The writable mapping 492 is writable in that write operations are allowed to data in the non-volatile memory 150 that is represented in the writable mapping 492.

**[0056]** The example address mapper 465 of FIG. 4 is implemented using software and/or firmware (e.g., as a client address space) stored in the processor system 100 (e.g., in a virtual address space in software, firmware, etc.). For example, instructions to implement the address mapper 465 may be stored in the non-volatile memory 150, or in any other storage device. In the illustrated example, the read-only mapping 490 of the non-volatile memory 150 is the only mapping of the non-volatile memory 150 that is directly accessed by the legacy software on processor core 130 (e.g., by client code executing via the processor core 130). That is, the processor core 130 initially does not access the writable mapping 492. Accordingly, the processor core 130, when executing a store event, attempts to write to the read-only mapping 490 because it is the only mapping of the non-volatile memory 150 that is used by software on processor core 130. Thus, when an event (e.g., a store event) includes an attempt to write data to a location in the read-only mapping 490 (e.g., the location (X) 494), a trap (e.g., a write fault error) is triggered because the processor 130 cannot write to data represented in the read-only mapping (because it is read-only). Based on the trap identifier 470 identifying a trap, the instruction trapper 420 determines that the address indicated in the store event is in the non-volatile memory 150. Because attempting to write to a data location represented in the read-only mapping 490 causes a trap or exception due to its read-only attribute, the instruction trapper 420 uses the trap or exception to facilitate identification of a corresponding location (e.g., the location (X +D) 496) in the writable mapping 492 through which the data can be updated in the non-volatile memory 150. The address and/or location of the writable mapping 492 is then tracked via a store_with_tracking operation by the cache controller 402 as disclosed herein as a dirty cache line address ($DIRTY_TAG_j$) 352. To facilitate a store_with_tracking in accordance with FIG. 4, the example instruction trapper 420 provides the storage location in the writable mapping 492 to the code modifier 454. The code modifier 454 generates a

store_with_tracking instruction and replaces, in the software program 456, a store instruction that caused the trapped store event. In the illustrated example, the code modifier 454 replaces the store instruction with the store_with_tracking instruction in a portion of RAM (e.g., the volatile memory 140) in which the software program 456 is loaded during runtime.

**[0057]** In the illustrated example of FIG. 4, the read-only mapping 490 of the data in the non-volatile memory 150 corresponds to addresses and/or locations of the data in the non-volatile memory 150. In the illustrate example, the address mapper 465 generates and/or stores the writable mapping 492 at an example offset (D) 498 (e.g., a number of addresses, physical locations, etc.) from the read-only mapping 490. For example, a read-only address (X) 494 (e.g., a pointer address) in the read-only mapping 490 may correspond to a writable address (X + D) 496 in the writable mapping 492, in which the offset (D) 498 is the offset value and the addresses (X) 494 and (X + D) 496 are addresses in the address mapper 465. In such examples, the data at the read-only address (X) 494 is the same as the data at the writable address (X +D) 496 (e.g., the data at writable address (X+D) 496 is a copy of the data at read-only address (X) 494). In such examples, using the offset (D) 498 aids the instruction trapper 420 in relatively quickly identifying the location (X+D) 496 in the writable mapping 492 that corresponds to the location (X) 494 of the read-only mapping 490 to which the processor core 130 attempted to perform a write operation.

**[0058]** In response to identifying a trap at a location (e.g., the read-only address (X) 494) in the read-only mapping 490, the example trap identifier 470 of FIG. 4, instructs the trap handler 480 to generate a special instruction (e.g., a store_with_tracking instruction indicating the writable address (X+D) 496) to track store events to the corresponding location in the writable mapping 492. In the illustrated example, the example trap handler 480 provides the store_with_tracking instruction to the code modifier 454. The code modifier 454 replaces the original store instruction in the software program 456 (e.g., the instruction in the binary code of the software program 456 that caused the trapped store event) with the store_with_tracking instruction to track the address

of the writable mapping 492. The example trapping technique implemented via the instruction trapper 420 as disclosed herein enables the cache controller 402 to determine which addresses indicated in store events are to be tracked in the dirty cache line address data structure 350 by identifying an attempt to write data to the read-only mapping 490 of the non-volatile memory 150. When the code modifier 454 replaces a legacy store instruction with a new store_with_tracking instruction, when the same store_with_tracking instruction is encountered again during execution of the software program 454(e.g., during multiple iterations of a loop), a trap will not be triggered because the store_with_tracking instruction corresponds to the writable mapping address (X+D) 496 in the writable mapping 492, which does not trigger a trap.

**[0059]** Accordingly, in the illustrated example of FIG. 4, the example trap handler 480 determines the location (e.g., the writable address (X+D) 496) in the writable mapping 492 that is to be tracked in the dirty cache line address data structure 350 in the event of a trap. The trap handler 480 provides the storage location (e.g., the writable address (X+D) 496) of the writable mapping 492 to the cache line address locator 230 and/or the dirty address updater 340 of FIG. 4 for tracking as disclosed herein. Thus, in such examples, the instruction trapper 420 of FIG. 4 identifies store events indicating addresses in the non-volatile memory 150 and replaces corresponding store instructions with store_with_tracking instructions.

**[0060]** Although FIG. 4 is representative of an example manner of implementing the instruction trapper 420 using software and/or firmware, the instruction trapper 420 may be implemented using any other suitable techniques including software, firmware, and/or hardware.

**[0061]** While an example hardware-based implementation of implementing 110 of FIG. 1 using the cache controller 402 of FIG. 4 and example software-based implementations of the example instruction trapper 420, the example dirty address updater 340 and the example dirty cache line address data structure 350 are illustrated in FIG. 4, one or more of the elements, processes and/or devices illustrated in FIG. 4 may be combined, divided, re-arranged, omitted, eliminated and/or implemented in any other

way. Further, although the example cache controller 402 is implemented in hardware, the example event identifier 210, the example cache line address locator 230, the example flush indicator 260, and/or more generally the example cache controller 402, as well as the, the dirty address updater 340, the dirty cache line address data structure 350, and the example code modifier 454, the example address mapper 465, the example trap identifier 470, the example trap handler 480, the example read-only mapping 490, the example writable mapping 492 and/or, more generally, the example instruction trapper 420 of FIG. 4 may be implemented by hardware, software, firmware and/or any combination of hardware, software and/or firmware. Thus, for example, any of the example event identifier 210, the example cache line address locator 230, the example flush indicator 260, and/or more generally the example cache controller 402, as well as the, the dirty address updater 340, the dirty cache line address data structure 350, and the example code modifier 454, the example address mapper 465, the example trap identifier 470, the example trap handler 480, the example read-only mapping 490, the example writable mapping 492 and/or, more generally, the example instruction trapper 420 could be implemented by one or more analog or digital circuit(s), logic circuits, programmable processor(s), application specific integrated circuit(s) (ASIC(s)), programmable logic device(s) (PLD(s)) and/or field programmable logic device(s) (FPLD(s)). When reading any of the apparatus or system claims of this patent to cover a purely software and/or firmware implementation, at least one of the example event identifier 210, the example cache line address locator 230, the example code modifier 454, the example flush indicator 260, the example dirty address updater 340, the example dirty cache line address data structure 350, the example address mapper 465, the example trap identifier 470, the example trap handler 480, the example read-only mapping 490, and/or the example writable mapping 492 is/are hereby expressly defined to include a tangible computer readable storage device or storage disk such as a memory, a digital versatile disk (DVD), a compact disk (CD), a Blu-ray disk, etc. storing the software and/or firmware. Further still, the example cache controller 402, the example dirty address updater 340, the example dirty cache line address data

structure 350 and/or the example instruction trapper 420 of FIG. 4 may include one or more elements, processes and/or devices in addition to, or instead of, those illustrated in FIG. 3, and/or may include more than one of any or all of the illustrated elements, processes and devices.

**[0062]** In examples disclosed herein, the cache controller 110 of FIG. 1 (e.g., implemented by one of the cache controllers 202, 302, 402 of FIGS. 2, 3, and/or 4) uses an example cache line address data structure (e.g., one of the cache line address data structures 250, 350 of FIGS. 2, 3, and/or 4) for efficient cache control by avoiding potentially unnecessary cache line flushes. For example, instead of writing back a cache line to the volatile memory 140 and/or the non-volatile memory 150 each time the cache line is made dirty (e.g., the data stored therein is modified), the example cache line address data structure tracks multiple modifications to the cache lines 122 so that the data ($DATA_i$) 128 in the cache lines 122 can be modified multiple times in the cache 120 when they are already dirty without having to perform multiple write backs for each modification when a dirty cache line, which has yet to be written back, is further modified.

**[0063]** Example methods, apparatus, and articles of manufacture disclosed herein enable the cache controllers 110 (implemented by one or more of the cache controllers, 202, 302, 402 of FIGS. 2, 3, and/or 4) to facilitate a full flush (i.e., a write back and/or clear/removal) of all dirty cache lines in the cache 120 of FIG. 1. In examples disclosed herein, an example event identifier (e.g., one of the event identifier 210 of FIGS. 2, 3, and/or 4) of the cache controller 110 identifies a request to flush all cache lines 122 (e.g., those cache lines corresponding to the non-volatile memory 150) in the cache 120 that are dirty. In such examples, the cache controller 110 identifies the cache lines 122 in the cache 120 that are dirty using an example dirty cache line address (e.g., one of the dirty cache line address data structures 250, 350 of FIGS. 2, 3, and/or 4). An example dirty address updater (e.g., one of the dirty address updaters 240, 340 of FIGS. 2, 3, and/or 4) provides the dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 to the memory controller 132 and/or to any other cache flush controller, and all cache lines 122 corresponding to the dirty cache line

addresses (DIRTY_TAG$_j$) 252, 352 are written back to the non-volatile memory 150 and, in some examples, cleared. Accordingly, rather than flushing the entire contents of the cache 120, only the dirty ones of the cache lines 122 are flushed, ensuring that the data that has yet to be modified in the non-volatile memory 150 gets updated with corresponding dirty cache lines. In such examples, a flush of the cache 120 involves flushing a number of dirty cache lines limited to the capacity limit of M dirty cache lines in the example dirty cache line address data structure. Because the number of dirty cache lines is limited by the cache controller 110, a flush of the cache 120 in FIG. 1 as disclosed herein is faster than a full flush of the cache 120, in which potentially every cache line 122 of the cache 120 is to be flushed, if all cache lines 122 in the entire capacity of the cache 120 are dirty.

**[0064]** The following is example pseudo code of an example of a full flush function that may be executed to perform a full flush operation of all the dirty cache lines in the cache 120.

```
[LOC1]   flush_all()
         {
[LOC2]       for (i=0; i< N; ++1) {
[LOC3]         if (may_be_dirty[i] !=0) {
[LOC4]           asynchronously_flush_cache_line(may_be_dirty[i]);
[LOC5]           may_be_dirty[i] = 0;
               }
             }
[LOC6]       wait_for_all_asynchronous_flushes_to_complete();
         }
```

**[0065]** Line of code [LOC1] of the above example pseudo code specifies the example name of the full flush function. At lines of code [LOC2], [LOC3], and [LOC4], the pseudo code flushes all cache lines corresponding to entries in the may_be_dirty array (e.g., the dirty cache line address data structure 250, 350 of FIGS. 2 and 3) that are non-zero asynchronously relative to other flushes. At line of code [LOC5], the may_be_dirty array (e.g., the dirty cache line address data structure 250, 350 of FIGS. 2 and 3) is cleared (i.e., zeroed out) to ensure that a second flush of the cache is not initiated. At line of code [LOC6], the system waits until all lines of code have been flushed.

**[0066]** Example methods, apparatus, and articles of manufacture disclosed herein enable the cache controller 110 of FIG. 1 (implemented by one or more of the cache controllers, 202, 302, 402 of FIGS. 2, 3, and/or 4) to facilitate flushing of dirty cache lines as a background process. In some examples, the cache controller 110 may determine, via an example event identifier (e.g., one of the event identifiers 210 of FIGS. 2, 3, and/or 4) when the processor core 130 has available resources to perform a flush of one or more dirty cache lines identified by an example dirty cache line address data structure (e.g., one of the dirty cache line address data structures 250, 350 of FIGS. 2, 3, and/or 4). In such examples, the cache controller 110 may determine which of the cache lines 122 in the cache 120 are to be flushed based on any type of hierarchical cache line removal technique. For example, the cache controller 110 may determine that a cache line is to be flushed based on a FIFO approach, a least frequently used approach, and/or an oldest accessed cache line approach. Such example methods allow the cache controller 110 to efficiently limit the number of cache lines 122 in the cache 120 that are dirty and/or correspond to the non-volatile memory 150 while preserving the processing efficiency of the processor core 130. In such examples, the background process to flush one or more dirty cache lines in the cache 120 is performed without being based on a store event (e.g., a store event that would cause a new dirty cache line address (e.g., one of the dirty cache line addresses ($DIRTY_TAG_j$) 252, 352) to be tracked in the example dirty cache line address data structure.

**[0067]** Flowcharts representative of example machine readable instructions for implementing the cache controller 202, 302, and/or 402 of FIGS. 2, 3, and/or 4 are shown in FIGS. 5 and 6. In this example, the machine readable instructions comprise processes for execution by a processor such as the processor core 130 shown in the example processor platform 100 discussed above in connection with FIG. 1. The processes may be embodied in software stored on a tangible computer readable storage medium such as a CD-ROM, a floppy disk, a hard drive, a digital versatile disk (DVD), a Blu-ray disk, or a memory associated with the processor core 130, but the entirety and/or parts of

the processes thereof could alternatively be executed by a device other than the processor core 130 and/or embodied in firmware or dedicated hardware. Further, although the example processes are described with reference to the flowcharts illustrated in FIGS. 5 and/or 6, many other methods of implementing the example cache controller 202, 302, and/or 402 may alternatively be used. For example, the order of execution of the blocks may be changed, and/or some of the blocks described may be changed, eliminated, or combined.

**[0068]** The process 500 of FIG. 5 begins with an initiation of the processor system 100 of FIG. 1 and/or an initiation of the cache controller 202, 302, 402 of FIGS. 2, 3, and/or 4 to implement the cache controller 110 of FIG. 1 to track dirty cache lines corresponding to the non-volatile memory 150 (block 505). The example process 500 may be executed to control the number of dirty cache lines in the cache 120 allocated to the non-volatile memory 150.

**[0069]** At block 510 of the illustrated example, the example event identifier 210 identifies a store event. Upon receiving instructions from the event identifier 210, at block 520 the example non-volatile address filter 220 determines an address in memory indicated in the store event to determine an address in memory (e.g., the volatile memory 140 and/or the non-volatile memory 150 of FIG. 1) to which data is to be stored. At block 530, the cache line address locator 230 determines a cache line address ($TAG_i$) 126 corresponding to the address indicated in the store event.

**[0070]** At block 540, the dirty address updater 240, 340 determines whether the cache line address ($TAG_i$) 126 corresponding to the address indicated in the store event address is already being tracked in the dirty cache line address data structure 250, 350 (e.g., a cache line address ($TAG_i$) 126 of FIG. 1 corresponds to one of the dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 of FIGS. 2, 3, and/or 4). If the dirty address updater 240, 340 determines at block 540 that the cache line address ($TAG_i$) 126 is in the dirty cache line address data structure 250, 350 (or corresponds to a dirty cache line address ($DIRTY_TAG_j$) 252, 352), control advances to block 580. In such examples, the dirty address updater 240, 340 does not perform an update (e.g., add a dirty cache line address ($DIRTY_TAG_j$) 252) to the dirty cache line address data

structure 250, 350 because the cache line address ($TAG_i$) 126 indicated in the store event is already being tracked in the dirty cache line address data structure 250, 350. In such examples, the cache line address ($TAG_i$) 126 may still be tracked in the dirty cache line address data structure 250, 350 because the cache line address has yet to be flushed from a previous modification to the data of the cache line. In such examples, the cache controller 110 avoids unnecessarily flushing a cache line 122 multiple times for multiple modifications. If the cache line address ($TAG_i$) 126 is not being tracked in the dirty cache line address data structure 250, 350, control advances to block 550.

**[0071]** At block 550 of FIG. 5, the dirty address updater 240, 340 determines whether the capacity limit M (or a threshold M') of the dirty cache line address data structure 250, 350 has been reached or exceeded. For example, the dirty address updater 240, 340 determines whether there are M dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 in the dirty cache line address data structure 250, 350. When the dirty address updater 240, 340 determines that the capacity limit M of the dirty cache line address data structure 250, 350 has not been reached or exceeded, control advances to block 570.

**[0072]** When the dirty address updater 240, 340 determines that the capacity limit M of the dirty cache line address data structure 250, 350 has been reached or exceeded, at block 560, the address updater 240, 340 selects a dirty cache line 122 corresponding to a dirty cache line address ($DIRTY_TAG_j$) 252, 352 from the dirty cache line address data structure 250, 350 to be written back to the non-volatile memory 150. At block 565, the cache controller 110 instructs the memory controller 132 and/or processor core 130 to flush the dirty cache line of the dirty cache line address and remove the dirty cache line address from the dirty cache line address data structure 250, 350. For example, at block 565, the cache controller 110 may instruct the memory controller 132 and/or processor core 130 to write back the data ($DATA_i$) 128 of the cache line address ($TAG_i$) 126 corresponding to the selected dirty cache line address ($DIRTY_TAG_j$) 252, 352 to the non-volatile memory 150. In some examples, the cache controller 110 facilitates the write back and clear of the dirty cache line

from the cache 120. The address updater 240, 340 removes the selected dirty cache line address ($DIRTY_TAG_j$) 252, 352 from the dirty cache line address data structure 250, 350. After the selected dirty cache line address ($DIRTY_TAG_j$) 252, 352 has been removed from the dirty cache line address data structure 250, 350, the dirty address updater 240, 340 adds the cache line address ($TAG_i$) 126 corresponding to the store event address to the dirty cache line address data structure 250, 350.

**[0073]** At block 580, the cache controller 110 determines whether to continue tracking dirty cache line addresses corresponding to the non-volatile memory 150. If the cache controller 110 is to continue to track dirty cache line addresses using the dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 (e.g., to continue to limit the quantity of cache lines allocated for the non-volatile memory 150), control returns to block 510. If the cache controller 110 is no longer to continue tracking dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 (e.g., due to a removal of power, a system failure, a change in cache control, etc.) the process 500 ends (block 590).

**[0074]** The process 600 of FIG. 6 begins with an initiation of the processor system 100 of FIG. 1 and/or an initiation of the cache controller 110 (which may be implemented using the cache controllers 202, 302, and/or 402 of FIGS. 2, 3, and/or 4) to monitor the processor system 100 (block 605). The example process 600 of FIG. 6 may be executed in parallel or in seriatim with the process 500 of FIG. 5. In some examples, the example process 600 may be executed to provide the memory controller 132 and/or any other flush controller all dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 of dirty cache lines in the cache 120 that are to be written back to the non-volatile memory 150 and/or cleared from the cache 120. In some examples, the process 600 may be performed in response to a cache flush command during normal system operation, and/or a cache flush command arising from a system failure and/or removal of power from the processor system 100. In such examples, dirty cache lines are relatively quickly written back to main memory (e.g., the non-volatile memory 150) to preserve the data state of any modifications made to data in the cache lines 122 of FIG. 1. In some examples, the process 600 may

additionally or alternatively be performed to identify when to perform a background flush of one or more dirty cache lines.

**[0075]** At block 610 of the illustrated example, the event identifier 210 (FIGS. 2, 3, and 4) of the cache controller 110 monitors the processor system 100 to determine when a full flush event is detected (e.g., in response to a system failure, removal of power, a user instruction, etc.). When a full flush event is detected by the event identifier 210 at block 610, control advances to block 620. At block 620, the cache controller 110 retrieves dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 from the dirty cache line address data structure 250, 350. At block 630, the cache controller 110 provides the cache line addresses ($TAG_i$) 126 corresponding to the dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 to the memory controller 132 (and/or other flush controller) to identify the cache line addresses ($TAG_i$) 126 that are to be flushed. In this manner, the memory controller 132, the processor core 130, and/or other flush controller may perform a flush of the identified dirty cache lines in the cache 120 that have yet to be written back to the non-volatile memory 150. Because examples disclosed herein limit the number of dirty cache lines corresponding to the non-volatile memory 150, this increases the likelihood of having sufficient power and/or resources to execute the operations of blocks 610, 620, and 630 of the process 600. This increases the likelihood that the non-volatile memory 150 is fully updated with recently modified cache data prior to a shutdown of the processor system 100 and/or in response to a full flush instruction from the processor system 100. After block 630, control advances to block 670.

**[0076]** Returning to block 610, when the event identifier 210 does not detect a full flush event, control advances to block 640. At block 640, the event identifier 210 (FIG. 2, 3, and/or 4), determines whether to initiate a background process to flush one or more dirty cache lines. For example, the event identifier 210 may determine whether a background flush can be performed, for example, to increase the likelihood that a number of dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 does not reach or exceed the capacity limit M. Accordingly, at block 640, the event identifier 210 monitors the processor

system 100 to determine whether a dirty cache line in the cache 120 may be flushed via a background process. In some examples, at block 640, the event identifier 210 may determine whether one or more dirty caches line in the cache 120 corresponding to a dirty cache line address ($DIRTY_TAG_j$) 252, 352 may be flushed based on availabilities of resources (e.g., processing power of the processor core 130). In some examples at block 640, the event identifier 210 identifies a number of operations (e.g., instructions that are being executed) that are occurring or that are expected to occur (and/or the corresponding resources needed for the operations). If sufficient resources remain to flush one or more dirty cache lines while normal operation of the processor system 100 is ongoing without relatively affecting the normal operation, the event identifier 210 may determine that a background flush can be performed. In such examples, a background process may be performed to flush one or more dirty cache lines to the non-volatile memory regardless of whether the number of dirty cache lines is less than, equal to, or greater than a capacity limit of M dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 in the dirty cache line address data structure 250, 350. In such examples, the background process may be performed without being based on (or caused by) a received store event (e.g., the store event does not cause a flush of a dirty cache line address to allow a new dirty cache line address ($DIRTY_TAG_j$) 252, 352 to be tracked in the dirty cache line address data structure 250, 350). If the event identifier 210 determines at block 640 that no dirty cache lines should be flushed using a background process (e.g., because there are insufficient resources to perform a flush), then control returns to block 610.

**[0077]** When the event identifier 110 determines at block 640 that a dirty cache line can be flushed using a background process, the cache controller 110 uses dirty address updater 240, 340 at block 650 to select one or more dirty cache line address(es) ($DIRTY_TAG_j$) 252, 352 to remove from the dirty cache line address data structure 250, 350. In the illustrated example, the dirty cache line address(es) ($DIRTY_TAG_j$) 252, 352 indicate that the corresponding cache line address(es) ($TAG_i$) 126 is to be flushed. Selection of the dirty cache line address(es) ($DIRTY_TAG_j$) 252, 352, at block 650, may be made based on any

type of hierarchical cache line removal technique. For example, the cache controller 110 may select one or more dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 based on a FIFO approach, a least frequently used cache line removal approach, and/or an LRU cache line removal approach.

**[0078]** At block 660 of the illustrated example, the flush indicator 260 the cache line address(es) ($TAG_i$) 126 corresponding to the selected dirty cache line addresses ($DIRTY_TAG_j$) 252, 352 to the memory controller 132, the processor core 130, and/or any other flush controller identifying which of the cache lines 122 in the cache 120 are to be flushed. In this manner, the memory controller 132, the processor core 132, and/or any other flush controller can perfrom a background flush. Accordingly, the operations of blocks 640, 650, and 660 of the example process 600 of FIG. 6 may be used to determine when to flush dirty cache lines in the cache 120 using a background process. After block 660, control advances to block 670.

**[0079]** At block 670, the cache controller 110 determines whether to continue monitoring the processor system 100 for cache flush events. When the cache controller 110 is to continue monitoring the processor system 100, control returns to block 610. When the cache controller 110 is not to continue monitoring the processor system 100, the process 600 ends (block 690).

**[0080]** A flowchart representative of example machine readable instructions for implementing the instruction trapper 420 of FIG. 4 is shown in FIG. 7. In this example, the machine readable instructions comprise a process for execution by a processor such as the processor core 130 shown in the example processor platform 100 discussed above in connection with FIG. 1. The process may be embodied in software stored on a tangible computer readable storage medium such as a CD-ROM, a floppy disk, a hard drive, a digital versatile disk (DVD), a Blu-ray disk, or a memory associated with the processor core 130, but the entire process and/or parts thereof could alternatively be executed by a device other than the processor core 130 and/or embodied in firmware or dedicated hardware. Further, although the example program is described with reference to the flowchart illustrated in FIG. 7, many other methods of implementing the example instruction trapper 420 may

alternatively be used. For example, the order of execution of the blocks may be changed, and/or some of the blocks described may be changed, eliminated, or combined.

**[0081]** The process 700 of FIG. 7 begins with an initiation of the cache controller 402 of FIG. 4 to perform trap handling to identify store instructions that are to be replaced with store_with_tracking instructions (block 705). The process 700 is executed to implement the instruction trapper 420 of FIG. 4 to analyze an address indicated in a store event to determine if the store event corresponds to the non-volatile memory 150 of FIG. 1, and if so, whether to trap the store event and replace a corresponding store instruction with a store_with_tracking instruction as disclosed herein.

**[0082]** At block 710 of FIG. 7, the trap identifier 210 determines when a store event (e.g., executed by the processor core 130 and/or memory controller 132) is detected. If a store event is not detected, the trap identifier 210 continues to monitor for store events. When a store event is detected at block 710, the trap identifier 470 (FIG. 4) determines (e.g., via a write-fault error detection), at block 720, whether the store event writes to an address location represented in the read-only mapping 490 (FIG. 4) corresponding to the non-volatile memory 150. If the store event does not write to a location represented in the read-only mapping 490, the trap handler 480 does not trap the store event (block 725). In some examples, a store event may not write to the read-only mapping 490 because the store event does not correspond to the non-volatile memory 150 (e.g., the store event corresponds to the volatile memory 140 of FIG. 1). In other examples, the store event may not write to the read-only mapping because the store event writes to the writable mapping 492, as the code modifier 454 previously replaced a store instruction with a store_with_tracking instruction, as disclosed herein.

**[0083]** When the trap identifier 470 detects a write to the read-only mapping 490 (block 720), the trap handler 480 (FIG. 4), at block 730, identifies a writable mapping address (e.g., the location (X+D) 496 of FIG. 4) in the writable mapping 492 that corresponds to the read-only mapping address (e.g., the location (X) 494 of FIG. 4) identified in the store event at block 720. In the

illustrated example, at block 730, the trap handler 480 identifies the writable mapping address in the writable mapping 492 (FIG. 4) using an offset (e.g., the offset (D) 498 of FIG. 3) relative to the read-only mapping address in the read-only mapping 490.

**[0084]** At block 740, the instruction trapper 420 (FIG. 4) provides the writable mapping address to the cache line address locator 230 (FIG. 4) and/or to the dirty address updater 340 (FIG. 4) for tracking the writable mapping address in the dirty cache line address data structure 350. After block 740, the code modifier 454 replaces the store instruction of the store event with a store_with_tracking instruction as disclosed herein (block 745). The example code modifier 454, at block 745, generates the store_with_tracking instruction using the writable mapping address (e.g., the location (X+D) 496). Once the store_with_tracking instruction has replaced the store instruction, for example, by replacing the store instruction in the software program 456, control advances to block 750. At block 750, the cache controller 402 instructs the memory controller 132 to execute the store at the corresponding address (block 750). In some examples, if, at block 720, the trap identifier 470 determines that the store event does not write to the read-only address mapping 490, the cache controller 402 instructs the memory controller 132 to perform the store event at the address indicated in the store event. In other examples, if the trap identifier 470 determines at block 720 that the store event does write to the read-only mapping 490, then the cache controller 110 instructs the memory controller 132 to perform the store event at a writable address (e.g., an address offset from the address of the read-only mapping) of the writable mapping 492.

**[0085]** At block 760 of the illustrated example, the instruction trapper 420 determines whether to continue trapping writes to the read-only mapping 490 (e.g., via write fault detection, etc.). If the instruction trapper 420 is to continue trap handling, control returns to block 710. If the instruction trapper 420 is not to continue trap handling, the program 700 ends (block 790).

**[0086]** As mentioned above, the example processes of FIGS. 5, 6, and/or 7 may be implemented using coded instructions (e.g., computer and/or machine readable instructions) stored on a tangible computer readable storage medium

such as a hard disk drive, a flash memory, a read-only memory (ROM), a compact disk (CD), a digital versatile disk (DVD), a cache, a random-access memory (RAM) and/or any other storage device or storage disk in which information is stored for any duration (e.g., for extended time periods, permanently, for brief instances, for temporarily buffering, and/or for caching of the information). As used herein, the term tangible computer readable storage medium is expressly defined to include any type of computer readable storage device and/or storage disk and to exclude propagating signals and to exclude transmission media. As used herein, "tangible computer readable storage medium" and "tangible machine readable storage medium" are used interchangeably. Additionally or alternatively, the example processes of FIGS. 5, 6, and/or 7 may be implemented using coded instructions (e.g., computer and/or machine readable instructions) stored on a non-transitory computer and/or machine readable medium such as a hard disk drive, a flash memory, a read-only memory, a compact disk, a digital versatile disk, a cache, a random-access memory and/or any other storage device or storage disk in which information is stored for any duration (e.g., for extended time periods, permanently, for brief instances, for temporarily buffering, and/or for caching of the information). As used herein, the term non-transitory computer readable medium is expressly defined to include any type of computer readable storage device and/or storage disk and to exclude propagating signals and to exclude transmission media. As used herein, when the phrase "at least" is used as the transition term in a preamble of a claim, it is open-ended in the same manner as the term "comprising" is open ended.

**[0087]** The coded instructions of FIGS 5, 6, and/or 7 may be stored in the cache 120, in another local storage of the processor core 130, in the non-volatile memory 150, in the mass storage device 190 of FIG. 1, and/or on a removable tangible computer readable storage medium such as a CD or DVD

**[0088]** From the foregoing, it will be appreciated that the above disclosed methods, apparatus, and articles of manufacture enable efficient cache flushing by monitoring a processor system and controlling a quantity of cache allocated to non-volatile memory. Example methods, apparatus, and articles of

manufacture provide efficient cache flushing by avoiding unnecessary cache flushes based on whether a modified cache line has yet to be written back to non-volatile memory following a previous modification (e.g., is tracked as a dirty cache line).

**[0089]** Although certain example methods, apparatus and articles of manufacture have been disclosed herein, the scope of coverage of this patent is not limited thereto. On the contrary, this patent covers all methods, apparatus and articles of manufacture fairly falling within the scope of the claims of this patent.

What Is Claimed Is:

1. A method to limit caching comprising:

monitoring cache lines in a cache, the cache lines storing recently written data to the cache, the recently written data corresponding to main memory;

comparing a total quantity of the cache lines to a threshold that is less than a cache line storage capacity of the cache; and

causing a write back of at least one of the cache lines to the main memory when a store event causes the total quantity of the cache lines to satisfy the threshold.

2. A method according to claim 1, wherein the threshold corresponds to at least one of:

a capacity of a data structure storing dirty cache line addresses, or

a cache line capacity of a cache way of the cache, the cache way designated to store cache lines corresponding to only the main memory.

3. A method according to claim 1, wherein the store event does not cause the total quantity of cache lines to increase toward the threshold when the store event modifies a dirty cache line.

4. A method according to claim 1, further comprising:

mapping a read-only memory address space and a writable memory address space; and

when the store event corresponds to the read-only memory address space, performing a store operation at a memory location corresponding to the writable memory address space.

5. A method according to claim 1, further comprising, when processor resources are available, selecting a second one of the cache lines to be written back to the main memory using a background process before the total quantity of the cache lines satisfies the threshold.

6. A method according to claim 1, wherein the total quantity of the cache lines satisfies the threshold when the total quantity of the cache lines is greater than or equal to the threshold.

7. A machine readable storage medium comprising instructions that, when executed, cause a machine to at least:

monitor cache lines in a cache, the cache lines storing recently written data to the cache, the recently written data corresponding to main memory;

compare a total quantity of the cache lines to a threshold that is less than a cache line storage capacity of the cache; and

cause a write back of at least one of the cache lines to the main memory when a store event causes the total quantity of the cache lines to satisfy the threshold.

8. A machine readable storage medium according to claim 7, wherein the threshold corresponds to at least one of:

a capacity of a data structure storing dirty cache line addresses,

or a cache line capacity of a cache way of the cache, the cache way designated to store cache lines corresponding to only the main memory.

9. A machine readable storage medium according to claim 7, wherein the store event does not cause the total quantity of cache lines to increase toward the threshold when the store event modifies a dirty cache line.

10. A machine readable storage medium according to claim 7, wherein the instructions, when executed further cause the machine to:

map a read-only memory address space and a writable memory address space; and

when the store event corresponds to the read-only memory address space, perform a store operation at a memory location corresponding to the writable memory address space.

11. A machine readable storage medium according to claim 7, wherein the instructions, when executed further cause the machine to:

when processor resources are available, select a second one of the cache lines to be written back to the main memory using a background process before the total quantity of the cache lines satisfies the threshold.

12. An integrated circuit comprising:

a dirty address updater to:

monitor dirty cache lines in a cache, the dirty cache lines storing data to be written back to a non-volatile memory, and

compare a total quantity of the dirty cache lines to a threshold that is less than a cache line storage capacity of the cache; and

a flush indicator to cause a write back of at least one of the dirty cache lines to the non-volatile memory when a store event causes the total quantity of the of dirty cache lines to satisfy the threshold.

13. An integrated circuit according to claim 12, wherein the dirty address updater is to compare the total quantity of the dirty cache lines to the threshold by comparing a quantity of dirty cache line addresses corresponding to the quantity of dirty cache lines to the threshold, wherein the threshold corresponds to a capacity of a data structure storing the dirty cache line addresses.

14. An integrated circuit according to claim 12, wherein the store event does not cause the total quantity of dirty cache lines to increase when the store event modifies a dirty cache line.

15. An integrated circuit according to claim 12, wherein, when processor resources are available, the address updater is select a second one of the dirty cache lines to be written back to the non-volatile memory using a background process before the total quantity of the dirty cache lines satisfies the threshold.

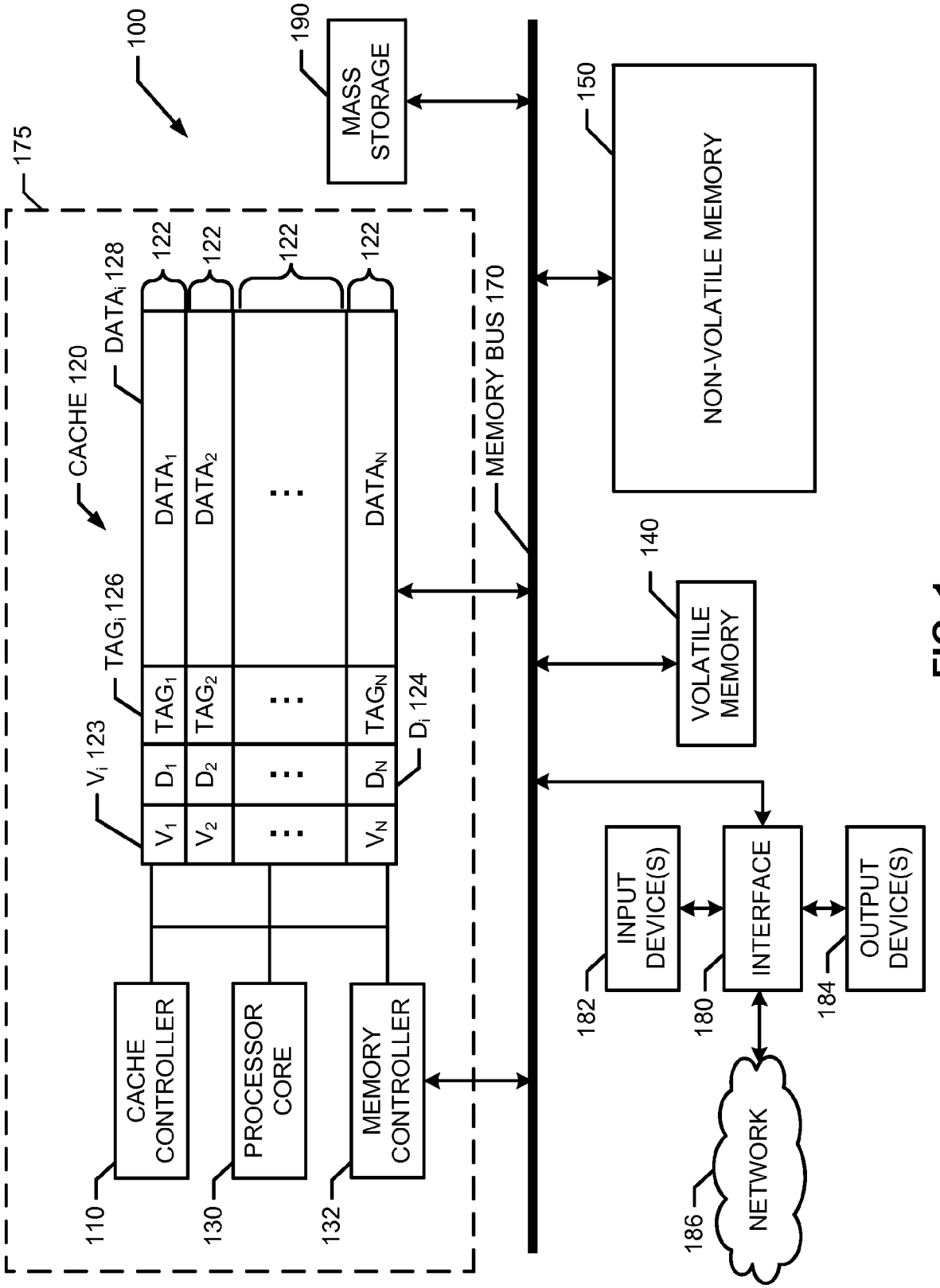

100
190
MASS STORAGE
150
NON-VOLATILE MEMORY
175
CACHE 120
DATA$_i$ 128
122
122
122
122
DATA$_1$
DATA$_2$
DATA$_N$
MEMORY BUS 170
TAG$_i$ 126
TAG$_1$
TAG$_2$
TAG$_N$
V$_i$ 123
D$_i$ 124
V$_1$
V$_2$
V$_N$
D$_1$
D$_2$
D$_N$
140
VOLATILE MEMORY
CACHE CONTROLLER
PROCESSOR CORE
MEMORY CONTROLLER
110
130
132
INPUT DEVICE(S)
INTERFACE
OUTPUT DEVICE(S)
182
180
184
NETWORK
186

FIG. 1

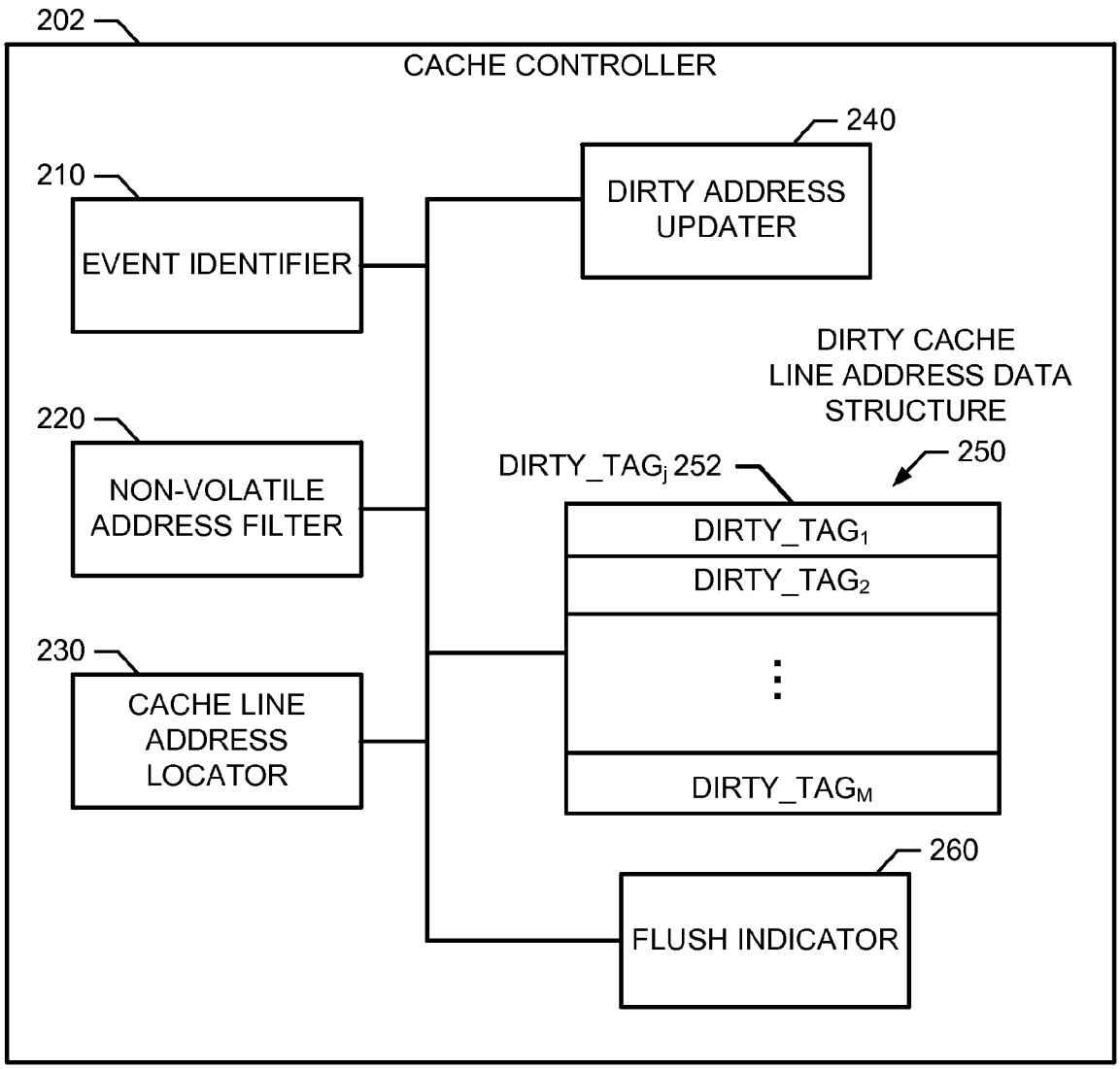

202
CACHE CONTROLLER
240
DIRTY ADDRESS UPDATER
210
EVENT IDENTIFIER
DIRTY CACHE LINE ADDRESS DATA STRUCTURE
220
NON-VOLATILE ADDRESS FILTER
DIRTY_TAG$_j$ 252
250
DIRTY_TAG$_1$
DIRTY_TAG$_2$
230
CACHE LINE ADDRESS LOCATOR
DIRTY_TAG$_M$
260
FLUSH INDICATOR

FIG. 2

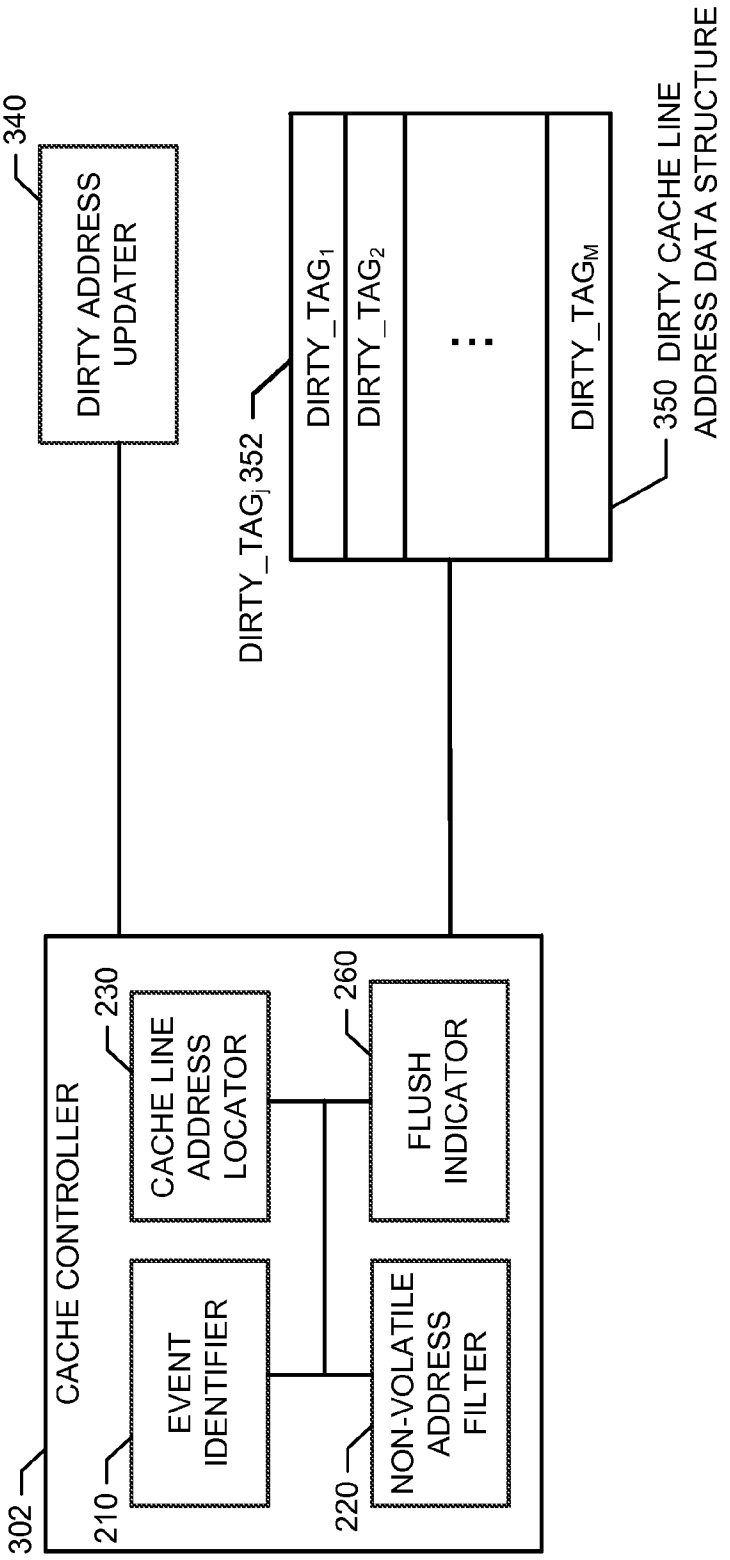

340
DIRTY ADDRESS UPDATER
DIRTY_TAG1
DIRTY_TAG2
DIRTY_TAGM
DIRTY_TAGJ 352
350 DIRTY CACHE LINE ADDRESS DATA STRUCTURE
CACHE CONTROLLER
230
CACHE LINE ADDRESS LOCATOR
260
FLUSH INDICATOR
EVENT IDENTIFIER
NON-VOLATILE ADDRESS FILTER
210
220
302

FIG. 3

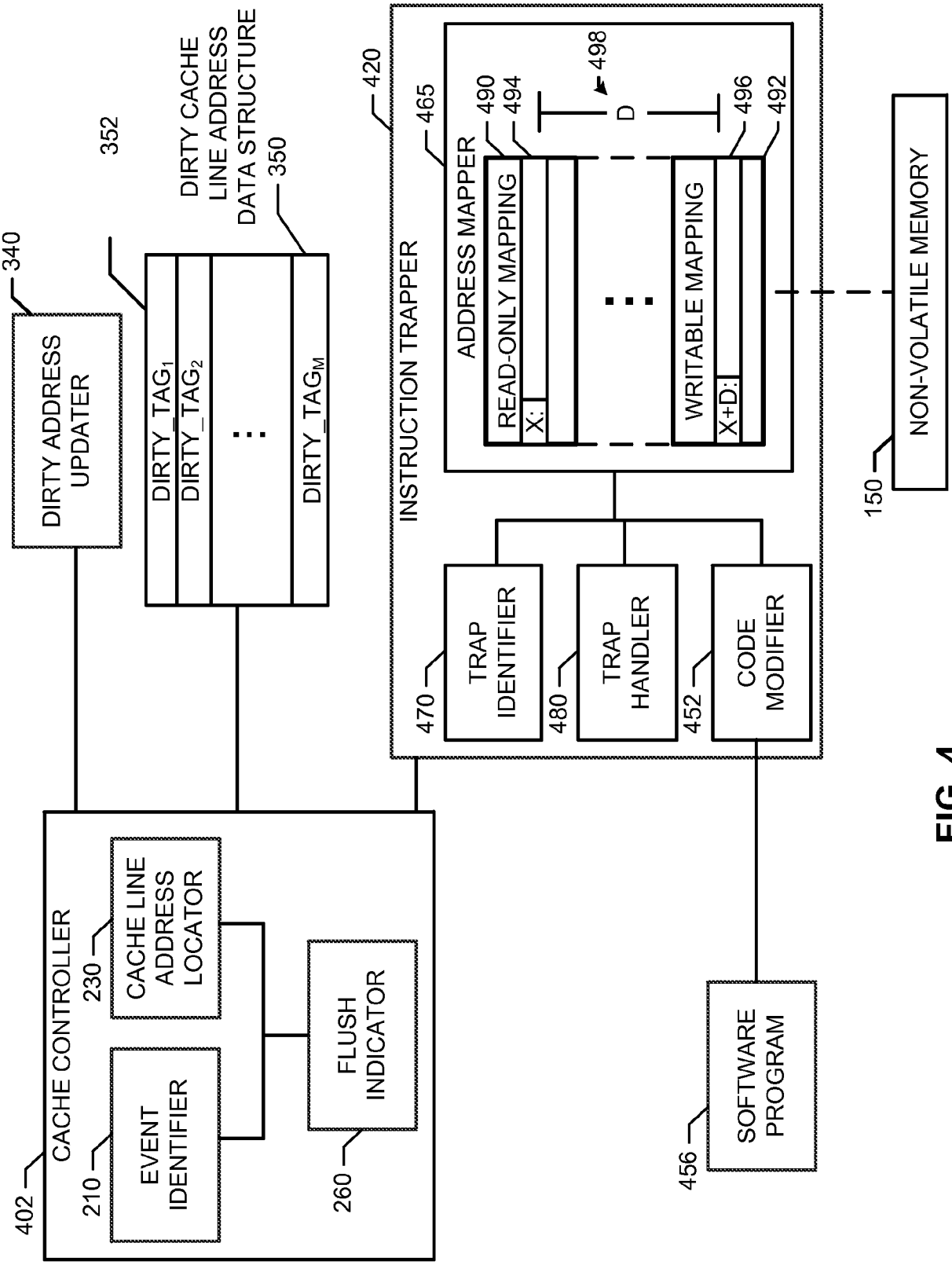

CACHE CONTROLLER
402
EVENT IDENTIFIER
210
CACHE LINE ADDRESS LOCATOR
230
FLUSH INDICATOR
260
DIRTY ADDRESS UPDATER
340
352
DIRTY CACHE LINE ADDRESS DATA STRUCTURE
350
DIRTY_TAG1
DIRTY_TAG2
DIRTY_TAGM
INSTRUCTION TRAPPER
420
ADDRESS MAPPER
465
READ-ONLY MAPPING
490
494
X:
498
D
WRITABLE MAPPING
496
492
X+D:
TRAP IDENTIFIER
470
TRAP HANDLER
480
CODE MODIFIER
452
SOFTWARE PROGRAM
456
NON-VOLATILE MEMORY
150

FIG. 4

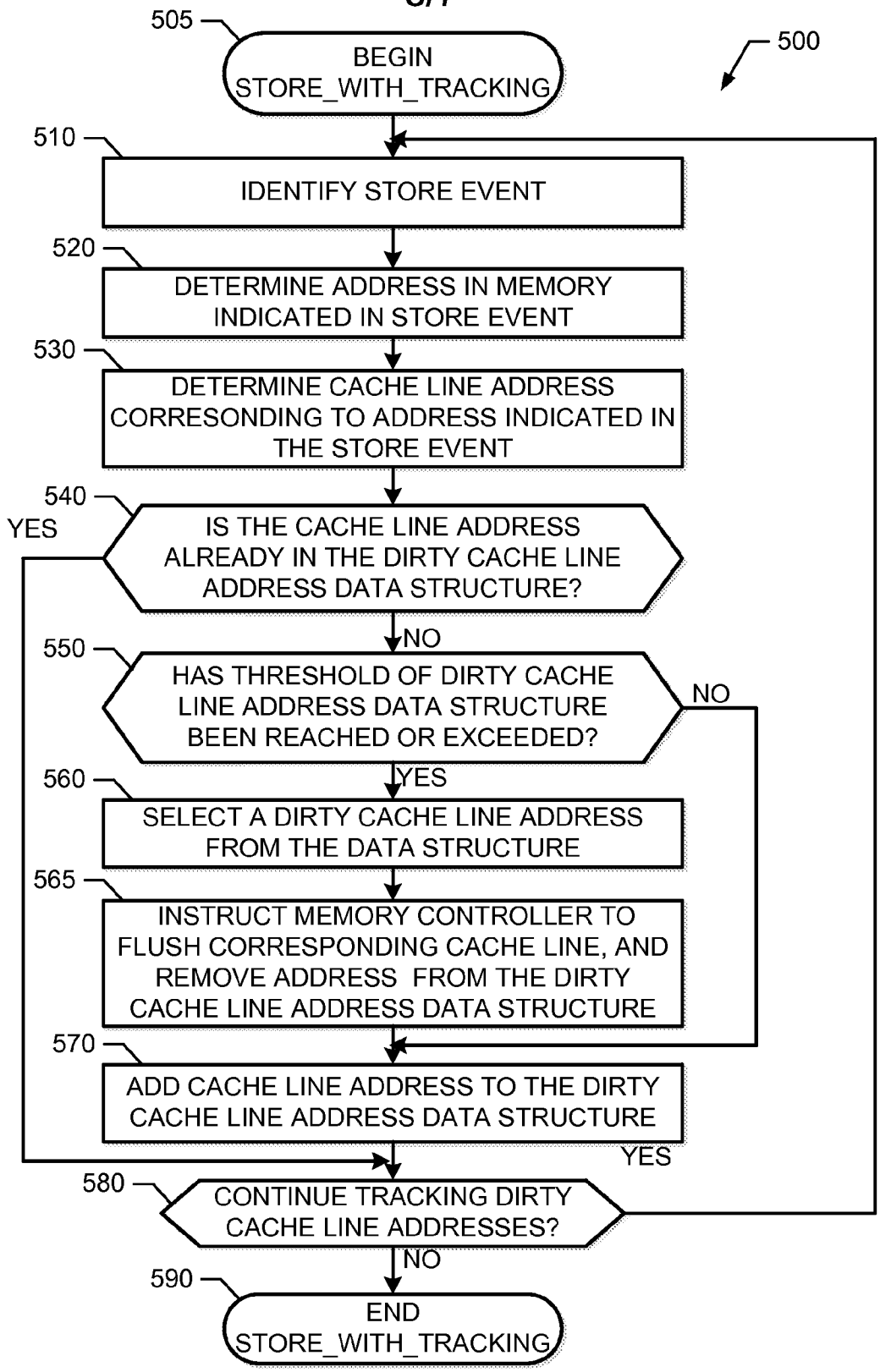

505
BEGIN STORE_WITH_TRACKING
500
510
IDENTIFY STORE EVENT
520
DETERMINE ADDRESS IN MEMORY INDICATED IN STORE EVENT
530
DETERMINE CACHE LINE ADDRESS CORRESONDING TO ADDRESS INDICATED IN THE STORE EVENT
540
YES
IS THE CACHE LINE ADDRESS ALREADY IN THE DIRTY CACHE LINE ADDRESS DATA STRUCTURE?
NO
550
HAS THRESHOLD OF DIRTY CACHE LINE ADDRESS DATA STRUCTURE BEEN REACHED OR EXCEEDED?
NO
YES
560
SELECT A DIRTY CACHE LINE ADDRESS FROM THE DATA STRUCTURE
565
INSTRUCT MEMORY CONTROLLER TO FLUSH CORRESPONDING CACHE LINE, AND REMOVE ADDRESS FROM THE DIRTY CACHE LINE ADDRESS DATA STRUCTURE
570
ADD CACHE LINE ADDRESS TO THE DIRTY CACHE LINE ADDRESS DATA STRUCTURE
YES
580
CONTINUE TRACKING DIRTY CACHE LINE ADDRESSES?
NO
590
END STORE_WITH_TRACKING

FIG. 5

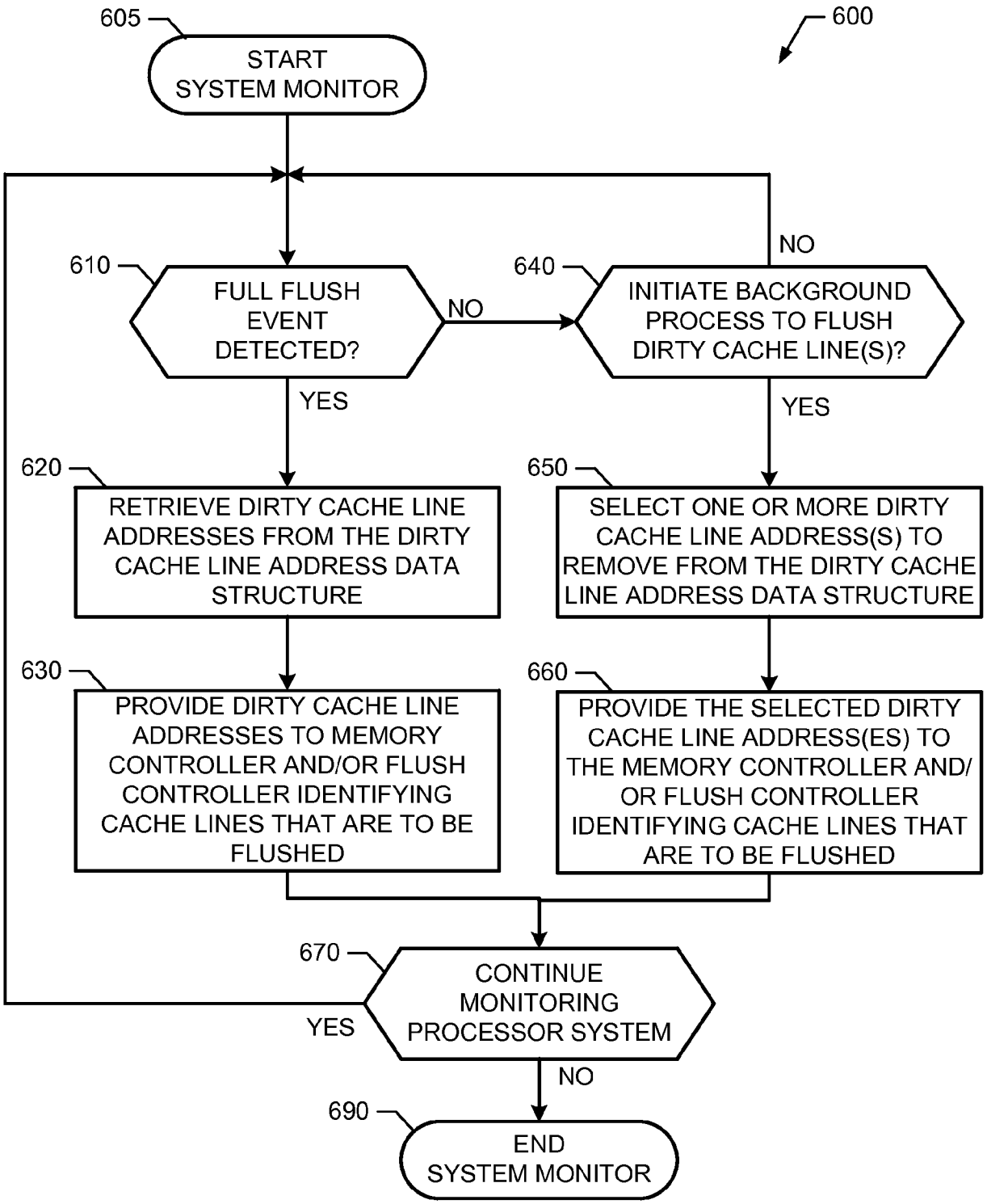

605
START SYSTEM MONITOR
600
610
FULL FLUSH EVENT DETECTED?
NO
640
INITIATE BACKGROUND PROCESS TO FLUSH DIRTY CACHE LINE(S)?
NO
YES
YES
620
RETRIEVE DIRTY CACHE LINE ADDRESSES FROM THE DIRTY CACHE LINE ADDRESS DATA STRUCTURE
650
SELECT ONE OR MORE DIRTY CACHE LINE ADDRESS(S) TO REMOVE FROM THE DIRTY CACHE LINE ADDRESS DATA STRUCTURE
630
PROVIDE DIRTY CACHE LINE ADDRESSES TO MEMORY CONTROLLER AND/OR FLUSH CONTROLLER IDENTIFYING CACHE LINES THAT ARE TO BE FLUSHED
660
PROVIDE THE SELECTED DIRTY CACHE LINE ADDRESS(ES) TO THE MEMORY CONTROLLER AND/OR FLUSH CONTROLLER IDENTIFYING CACHE LINES THAT ARE TO BE FLUSHED
670
CONTINUE MONITORING PROCESSOR SYSTEM
YES
NO
690
END SYSTEM MONITOR

FIG. 6

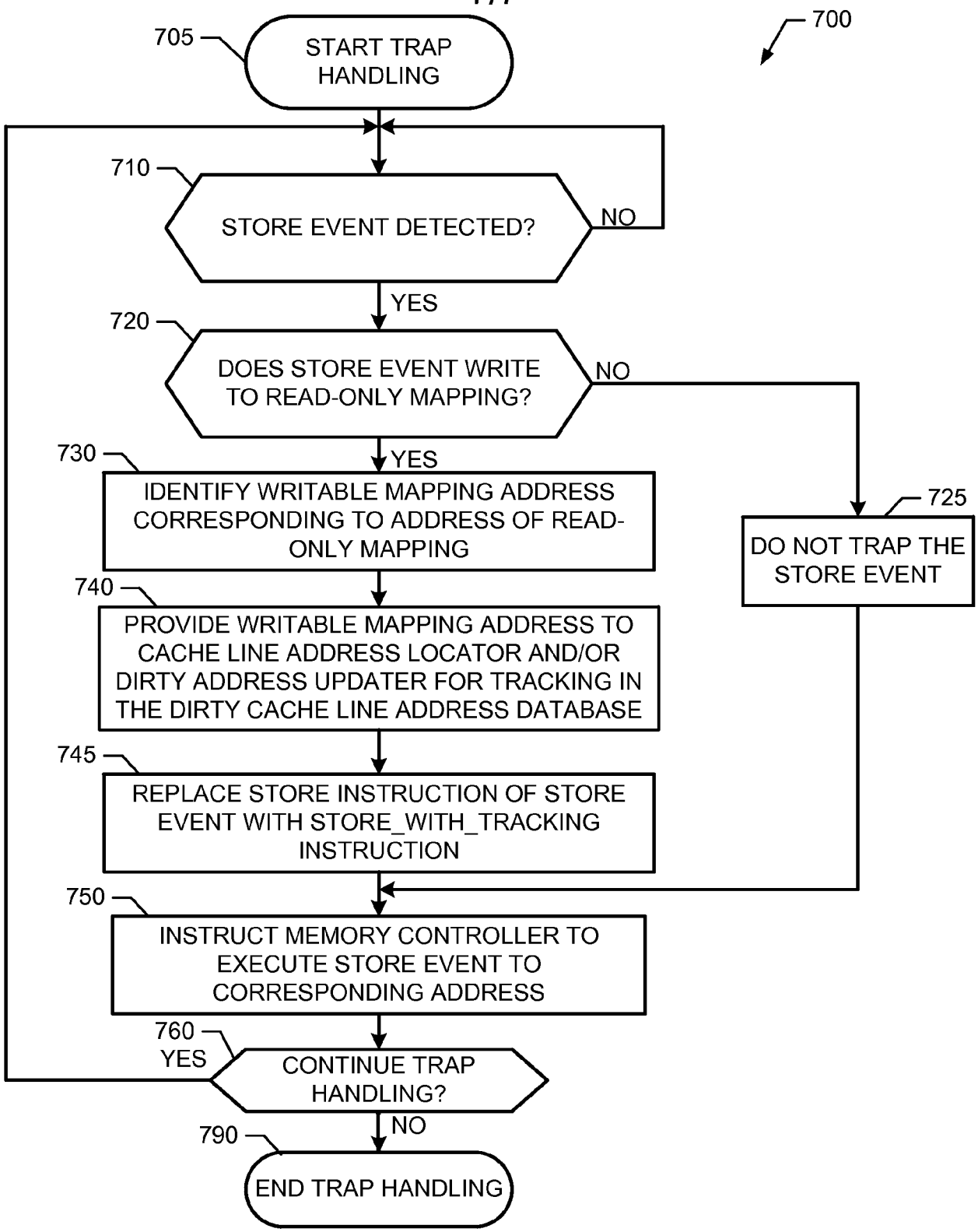

700
705
START TRAP HANDLING
710
STORE EVENT DETECTED?
NO
YES
720
DOES STORE EVENT WRITE TO READ-ONLY MAPPING?
NO
YES
730
IDENTIFY WRITABLE MAPPING ADDRESS CORRESPONDING TO ADDRESS OF READ-ONLY MAPPING
725
DO NOT TRAP THE STORE EVENT
740
PROVIDE WRITABLE MAPPING ADDRESS TO CACHE LINE ADDRESS LOCATOR AND/OR DIRTY ADDRESS UPDATER FOR TRACKING IN THE DIRTY CACHE LINE ADDRESS DATABASE
745
REPLACE STORE INSTRUCTION OF STORE EVENT WITH STORE_WITH_TRACKING INSTRUCTION
750
INSTRUCT MEMORY CONTROLLER TO EXECUTE STORE EVENT TO CORRESPONDING ADDRESS
760
YES
CONTINUE TRAP HANDLING?
NO
790
END TRAP HANDLING

FIG. 7

**INTERNATIONAL SEARCH REPORT**

International application No.
**PCT/US2013/067841**

**A. CLASSIFICATION OF SUBJECT MATTER**

**G06F 12/08(2006.01)i, G06F 11/30(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 12/08; G06F 12/00; G06F 13/00; G06F 11/00; G06F 11/30

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: cache, dirty cache line, address, capacity, write back, flush, non-volatile memory, limit, hash table

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| A | US 2009-0089508 A1 (SANJEEV N. TRIKA et al.) 02 April 2009<br>See paragraphs [0007]-[0013]; claim 1; and figure 1. | 1-15 |
| A | US 7062675 B1 (JOHN N. KEMENY et al.) 13 June 2006<br>See column 3, line 4 - column 6, line 32; claim 1; and figures 1-2D. | 1-15 |
| A | US 2005-0138283 A1 (JOHN I. GARNEY) 23 June 2005<br>See paragraphs [0014]-[0018]; and figure 2. | 1-15 |
| A | US 6119205 A (THOMAS M. WICKI et al.) 12 September 2000<br>See column 8, line 60 - column 9, line 67; and figure 4. | 1-15 |
| A | US 2012-0246410 A1 (HUI XU) 27 September 2012<br>See paragraphs [0029]-[0036]; and figures 3-6. | 1-15 |

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

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

| Date of the actual completion of the international search | Date of mailing of the international search report |
|---|---|
| 25 July 2014 (25.07.2014) | **28 July 2014 (28.07.2014)** |

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701, Republic of Korea
Facsimile No. +82-42-472-7140

Authorized officer
KIM, Do Weon
Telephone No. +82-42-481-5560

**INTERNATIONAL SEARCH REPORT**

Information on patent family members

International application No.
**PCT/US2013/067841**

| Patent document cited in search report | Publication date | Patent family member(s) | Publication date |
|---|---|---|---|
| US 2009-0089508 A1 | 02/04/2009 | US 7966456 B2 | 21/06/2011 |
| US 7062675 B1 | 13/06/2006 | None | |
| US 2005-0138283 A1 | 23/06/2005 | US 7130962 B2 | 31/10/2006 |
| US 06119205 A | 12/09/2000 | None | |
| US 2012-0246410 A1 | 27/09/2012 | JP 2012-203560 A | 22/10/2012 |