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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
31 December 2008 (31.12.2008)

PCT

(10) International Publication Number
WO 2009/002755 A2

(51) International Patent Classification:
G06F 13/00 (2006.01) *G06F 9/455* (2006.01)

(21) International Application Number:
PCT/US2008/067146

(22) International Filing Date: 16 June 2008 (16.06.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11/823,224 27 June 2007 (27.06.2007) US

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

(72) Inventors: **TAILLEFER, Martin**; One Microsoft Way, Redmond, Washington 98052-6399 (US). **MIHOCKA, Darek**; One Microsoft Way, Redmond, Washington 98052-6399 (US). **SILVA, Bruno**; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:
— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:
— *without international search report and to be republished upon receipt of that report*

(54) **Title:** LEVERAGING TRANSACTIONAL MEMORY HARDWARE TO ACCELERATE VIRTUALIZATION AND EMULATION

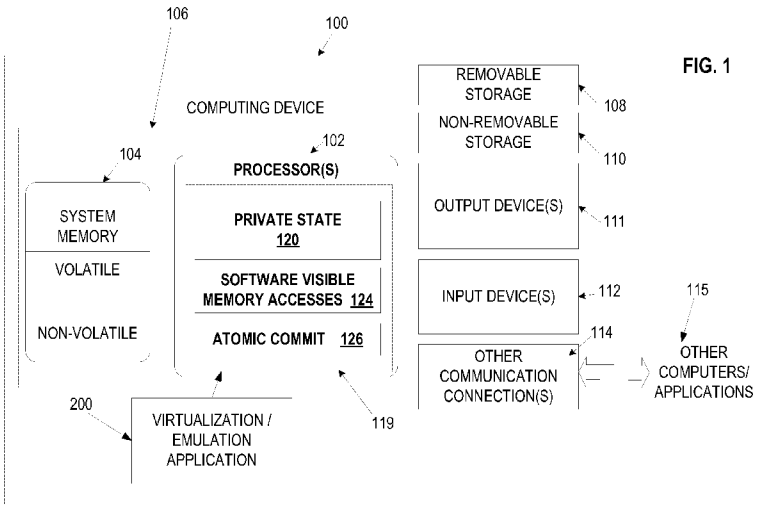


FIG. 1

(57) **Abstract:** Various technologies and techniques are disclosed for using transactional memory hardware to accelerate virtualization or emulation. State isolation can be facilitated by providing isolated private state on transactional memory hardware and storing the stack of a host that is performing an emulation in the isolated private state. Memory accesses performed by a central processing unit can be monitored by software to detect that a guest being emulated has made a self modification to its own code sequence. Transactional memory hardware can be used to facilitate dispatch table updates in multithreaded environments by taking advantage of the atomic commit feature. An emulator is provided that uses a dispatch table stored in main memory to convert a guest program counter into a host program counter. The dispatch table is accessed to see if the dispatch table contains a particular host program counter for a particular guest program counter.

WO 2009/002755 A2

LEVERAGING TRANSACTIONAL MEMORY HARDWARE TO ACCELERATE VIRTUALIZATION AND EMULATION

BACKGROUND

[001] Emulation is a technique that involves providing the functionality of a central processing unit from software. One advantage to emulation is that you can run software designed for a specific type of processor on any other type of processor. Virtualization is a technique that provides the ability to partition hardware in a way that allows more than one operating system to run simultaneously. With virtualization, a physical central processing unit is partitioned into multiple contexts. Each context then takes turns running directly on the processor. Even though virtualization products are generally faster than emulation products, both types of products today suffer from several complications that limit the speed that can be achieved. As one example, dealing with concurrency during emulation requires additional synchronization which can slow down serial performance. As a second example, tracking self-modifying code under emulation usually incurs a substantial performance burden.

SUMMARY

[002] Various technologies and techniques are disclosed for using transactional memory hardware to accelerate virtualization or emulation. One or more central processing units are provided with transactional memory hardware that is operable to accelerate virtualization. The transactional memory hardware has a facility to maintain private state, a facility to render memory accesses from other central processing units visible to software, and support for atomic commit of the private state.

[003] In one implementation, the transactional memory hardware can be used, for example, to facilitate emulation of precise exception semantics. The private state is operable to enable an emulated state to remain inconsistent with an architectural state and only synchronized on certain boundaries. An optimized sequence of instructions is executed using chunk-accurate simulation to try and achieve a same end effect as a slower instruction-accurate simulation, isolated in private state, which is only committed in case the entire chunk is successfully emulated.

[004] In another implementation, state isolation can be facilitated by providing isolated private state on transactional memory hardware and storing the stack of a host that is performing emulation in the isolated private state.

[005] In yet another implementation, self-modifying code can be detected using the transactional memory hardware. Memory accesses performed by a central processing unit can be monitored by software to detect that a guest being emulated has made a self modification to its own code sequence.

[006] Transactional memory hardware can be used in one implementation to facilitate dispatch table updates in multithreaded environments by taking advantage of the atomic commit feature. An emulator is provided that uses a dispatch table stored in main memory to convert a guest program counter into a host program counter. The dispatch table is accessed to see if the dispatch table contains a particular host program counter for a particular guest program counter. When a host program counter is not found, transactional memory hardware can be used to introduce a new mapping in the table.

[007] In yet another implementation, code backpatching can be facilitated by providing transactional memory hardware that supports a facility to maintain private memory state and an atomic commit feature. Changes made to certain code are stored in the private state facility. Backpatching changes are enacted by attempting to commit all the changes to memory at once using the atomic commit feature.

[008] An efficient call return cache can be provided in one implementation by using transactional memory hardware. A call return cache stored in the private state facility captures a host address to return to after execution of a guest function completes. A direct-lookup hardware-based hash table is used for the call return cache.

[009] This Summary was provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used as an aid in determining the scope of the claimed subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

[010] Figure 1 is a diagrammatic view of a computer system of one implementation.

[011] Figure 2 is a diagrammatic view of a virtualization/emulation application of one implementation operating on the computer system of Figure 1.

[012] Figure 3 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in using transactional memory hardware to facilitate emulation of precise exception semantics.

[013] Figure 4 is a diagram for one implementation of the system of Figure 1 that illustrates running a chunk-accurate simulation.

[014] Figure 5 is a diagram for one implementation of the system of Figure 1 that illustrates running an instruction-accurate simulation after a chunk-accurate simulation has raised an exception.

[015] Figure 6 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in using transactional memory hardware to facilitate state isolation.

[016] Figure 7 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using transactional memory hardware to facilitate detection of self-modifying code under emulation.

[017] Figure 8 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using transactional memory hardware to facilitate dispatch table updates in multithreaded environments.

[018] Figure 9 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using transactional memory hardware to facilitate code backpatching in multithreaded environments.

[019] Figure 10 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in using transactional memory hardware to facilitate an efficient call return cache.

DETAILED DESCRIPTION

[020] The technologies and techniques herein may be described in the general context as a virtualization or emulation system, but the technologies and techniques

also serve other purposes in addition to these. In another implementation, the system may include one or more central processors that use transactional memory hardware to accelerate virtualization and/or emulation.

[021] As shown in Figure 1, an exemplary computer system to use for implementing one or more parts of the system includes a computing device, such as computing device 100. In its most basic configuration, computing device 100 typically includes at least one central processing unit 102 and memory 104. Central processing unit(s) each have transactional memory hardware 119 that includes a facility to maintain private state 120, a facility to render memory accesses from other central processing units visible to software 124, and support for atomic commit of the private state 126. The private state 120 is invisible to other central processing units until it is explicitly atomically committed by owning processing unit. The private state 120 is also discardable. When the central processing discards the private state, its view of memory is reverted to the current architectural state. The software visible memory accesses 124 allow software running on the central processing unit to detect that another central processing unit is accessing specific memory locations. The atomic commit feature 126 allows the private state of the central processing unit to enter a main memory system atomically as a unit as part of an atomic commit. These transactional memory hardware features are used in various scenarios for enhancing virtualization and/or emulation, as described in further detail herein.

[022] Depending on the exact configuration and type of computing device, memory 104 may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. This most basic configuration is illustrated in Figure 1 by dashed line 106.

[023] Additionally, device 100 may also have additional features/functionality. For example, device 100 may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or tape. Such additional storage is illustrated in Figure 1 by removable storage 108 and non-removable storage 110. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for

storage of information such as computer readable instructions, data structures, program modules or other data. Memory 104, removable storage 108 and non-removable storage 110 are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by device 100. Any such computer storage media may be part of device 100.

[024] Computing device 100 includes one or more communication connections 114 that allow computing device 100 to communicate with other computers/applications 115. Device 100 may also have input device(s) 112 such as keyboard, mouse, pen, voice input device, touch input device, etc. Output device(s) 111 such as a display, speakers, printer, etc. may also be included. These devices are well known in the art and need not be discussed at length here. In one implementation, computing device 100 includes virtualization/emulation application 200. Virtualization/emulation application 200 will be described in further detail in Figure 2.

[025] Turning now to Figure 2 with continued reference to Figure 1, a virtualization/emulation application 200 operating on computing device 100 is illustrated. Virtualization/emulation application 200 is one of the application programs that reside on computing device 100. However, it will be understood that virtualization/emulation application 200 can alternatively or additionally be embodied as computer-executable instructions on one or more computers and/or in different variations than shown on Figure 1. Alternatively or additionally, one or more parts of virtualization/emulation application 200 can be part of system memory 104, on other computers and/or applications 115, or other such variations as would occur to one in the computer software art.

[026] Virtualization/emulation application 200 includes program logic 204, which is responsible for carrying out some or all of the techniques described herein. Program logic 204 includes logic for using transactional memory hardware to

accelerate virtualization or emulation 206 (as described below with respect to Figure 3); logic for using transactional memory hardware to facilitate emulation of precise exception semantics 208 (as described below with respect to Figures 3-5); logic for using transactional memory hardware to facilitate state isolation 210 (as described below with respect to Figure 6); logic for using transactional memory hardware to facilitate detection of self-modifying code 212 (as described below with respect to Figure 7); logic for using transactional memory hardware to facilitate dispatch table updates 214 (as described below with respect to Figure 8); logic for using transactional memory hardware to facilitate code backpatching 216 (as described below with respect to Figure 9); logic for using transactional memory hardware to facilitate an efficient call return cache 218 (as described below with respect to Figure 10); and other logic for operating the application 220. In one implementation, program logic 204 is operable to be called programmatically from another program, such as using a single call to a procedure in program logic 204.

[027] Turning now to Figures 3-10 with continued reference to Figures 1-2, the stages for implementing one or more implementations of virtualization/emulation application 200 are described in further detail. In some implementations, the processes of Figures 3-10 are at least partially implemented in the operating logic of computing device 100. Figure 3 illustrates one implementation of the stages involved in using transactional memory hardware to facilitate emulation of precise exception semantics. The process begins at start point 240 with using invisible and discardable private state in the transactional memory hardware to enable the emulated state to remain inconsistent with architectural state and only synchronized on coarse boundaries (stage 242). The system runs the optimized sequence of instructions using a chunk-accurate simulation (stage 244). If no exception occurs (decision point 246), then the system is able to complete the simulation with generally better performance than would be possible using fully instruction-accurate simulation. However, if an exception occurs (decision point 246), then the pending writes to memory are discarded (stage 248). The emulated processor state is rolled back to the last synchronization point (stage 250). The particular code sequence is executed again conservatively using instruction-accurate simulation so

that the right architectural state is present when the exception is encountered for the second time (stage 252). In one implementation, that architectural state is precisely the same as the real, non-virtual system being emulated would be in, which is important for correct system-level emulation. This process is illustrated in further detail in the diagrams of Figures 4 and 5. The process ends at end point 254.

[028] Turning now to Figures 4 and 5, the use of transactional memory hardware to facilitate emulation of precise exceptions semantics in emulated environments is illustrated in further detail. Figure 4 is a diagram 270 for one implementation that illustrates running a chunk-accurate simulation. The chunk-accurate simulation runs an optimized sequence of instructions to try and achieve the same end effect as the original sequence of guest instructions, but in a more optimized way than is possible using an instruction-accurate simulation. The chunk-accurate simulation allows the emulated state to remain inconsistent with architectural state and only be synchronized on coarse boundaries, which is basically the starting point and the ending point of the simulation. If an exception is encountered at any point during the chunk-accurate simulation, then the instruction-accurate simulation illustrated on the diagram 290 of Figure 5 is executed. The instruction-accurate simulation runs the code sequence conservatively to ensure that the right architectural state is present when the exception is encountered for the second time.

[029] Figure 6 illustrates one implementation of the stages involved in using transactional memory hardware to facilitate state isolation. The process begins at start point 310 with providing isolated private state on transactional memory hardware (stage 312). In an emulated environment, the host doing the emulation stores its stack in the isolated private state (stage 314). The isolated private state enables the host to keep the stack in a buffered mode, which allows the host to avoid expensive runtime checks. The isolated private state is discardable. This means that when the central processing unit discards the private state, its view of memory is reverted to the current architectural state. The guest being emulated does not have access to the isolated private state (stage 316). The process ends at end point 318.

[030] Figure 7 illustrates one implementation of the stages involved in using transactional memory hardware to facilitate detection of self-modifying code. The process begins at start point 340 providing software-visible memory accesses using transactional memory hardware (stage 342). The system monitors the memory accesses performed by a central processing unit to detect that the guest being emulated has modified its own code sequence (stage 344). Even memory accesses from a separate central processing unit can be detected. The system then takes an appropriate action based on the detection (stage 346). A few non-limiting examples of such appropriate actions can include recompiling the code sequence to incorporate the self modification made by the guest, raising an exception, and/or stopping execution. Another non-limiting example of an appropriate action can include switching to an interpretation of machine instructions (as opposed to compilation) for just the part of the code that is self-modifying. In one implementation, this approach would be slower than compilation, but very accurate and would work exactly as the self-modifying code was intended to. Other appropriate actions are also possible. The process ends at end point 348.

[031] Figure 8 illustrates one implementation of the stages involved in using transactional memory hardware to facilitate dispatch table updates in multithreaded environments. The process begins at start point 370 with providing an atomic commit feature using transactional memory hardware (stage 372). The system provides an emulator that uses a dispatch table stored in main memory to convert a guest program counter into a host program counter (stage 374). The system accesses the dispatch table to see if it contains a host program counter for the given guest program counter (stage 376). If no host program counter is found, the system compiles the associated block of guest code (stage 378). The update to the dispatch table is performed in private memory, and when the update is fully performed, it is atomically committed to main memory. Leveraging the atomic commit feature, the system updates the dispatch table in an atomic way with minimal overhead (stage 380). The process ends at end point 382.

[032] Figure 9 illustrates one implementation of the stages involved in using transactional memory hardware to facilitate code backpatching in multithreaded

environments. The process begins at start point 400 with providing invisible private state and atomic commit facilities using transactional memory hardware (stage 402). The system stores the changes to be made to code in the invisible private state (stage 404). The changes to be made to the code can be received from one or more threads. The backpatching changes are enacted by attempting to commit all the changes to memory at one time using atomic commit (stage 406). If another thread is trying to execute code that needs to be changed, commit will fail and the backpatching process re-executes by attempting to commit all the changes to memory atomically again (stage 408). By using such backpatching process with transactional memory functionality, safe dynamic code modification is enabled in a multithreaded environment (stage 410). The process ends at end point 412.

[033] Figure 10 illustrates one implementation of the stages involved in using transactional memory hardware to facilitate an efficient call return cache. The process begins at start point 430 with providing invisible and discardable private state using transactional memory hardware (stage 432). The system provides a call return cache stored in the private state facility that captures the host address to return to after execution of the guest function completes (stage 434). A direct-lookup hardware-based hash table is used for the call return cache (stage 436). In one implementation, the call return cache supports more efficient lookups than normal lookups in a dispatch table. The process ends at end point 438.

[034] Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims. All equivalents, changes, and modifications that come within the spirit of the implementations as described herein and/or by the following claims are desired to be protected.

[035] For example, a person of ordinary skill in the computer software art will recognize that the examples discussed herein could be organized differently on one or more computers to include fewer or additional options or features than as portrayed in the examples.

What is claimed is:

1. A method for using transactional memory hardware to facilitate state isolation comprising the steps of:

accessing isolated private state on transactional memory hardware (312); and

storing a stack of a host that is performing an emulation in the isolated private state (314).

2. The method of claim 1, wherein the isolated private state is only visible to a central processing unit (312).

3. The method of claim 2, wherein the isolated private state ensures that a guest being emulated does not have access to the stack of the host (316).

4. The method of claim 2, wherein the isolated private state cannot be accessed by other central processing units or hardware devices (316).

5. The method of claim 1, wherein the isolated private state enables the host to keep the stack in a buffered mode (314).

6. The method of claim 5, wherein the buffered mode allows the host to avoid expensive runtime checks (314).

7. The method of claim 1, wherein the isolated private state is discardable (248).

8. The method of claim 7, wherein when a central processing unit discards the private state, a view of memory by the central processing unit is reverted to a current architectural state (252).

9. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 1(200).

10. A computer-readable medium having computer-executable instructions for causing a computer to perform steps comprising:

monitoring memory accesses performed by a central processing unit to detect that a guest being emulated has made a self modification to a code sequence of the guest (344); and

taking an appropriate action based on detection of the self modification to the code sequence (346).

11. The computer-readable medium of claim 10, wherein the appropriate action includes recompiling the code sequence to incorporate the self modification made by the guest (346).

12. The computer-readable medium of claim 10, wherein the appropriate action includes switching to an interpretation of machine instructions for the code sequence that was modified (346).

13. The computer-readable medium of claim 10, wherein memory accesses from a separate central processing unit can be detected (344).

14. A method for using transactional memory hardware to facilitate dispatch table updates in multithreaded environments comprising the steps of:

providing access to a central processing unit with transactional memory hardware, the transactional memory hardware comprising an atomic commit feature (372); and

providing an emulator that uses a dispatch table stored in main memory to convert a guest program counter into a host program counter (374); and

accessing the dispatch table to see if the dispatch table contains a particular host program counter for a particular guest program counter (376).

15. The method of claim 14, further comprising:

if the particular host program counter is not found for the particular guest program counter, then compiling an associated block of guest code (378).

16. The method of claim 15, wherein by leveraging the atomic commit feature of the transactional memory hardware, the dispatch table is updated in an atomic way (380).

17. The method of claim 16, wherein the dispatch table is updated with minimal overhead (380).

18. The method of claim 14, wherein the emulator is provided in a multithreaded environment (374).

19. The method of claim 18, wherein the dispatch table can be updated by the central processing unit while being used concurrently by a plurality of other central processing units without any locking overhead (380).

20. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 14 (200).

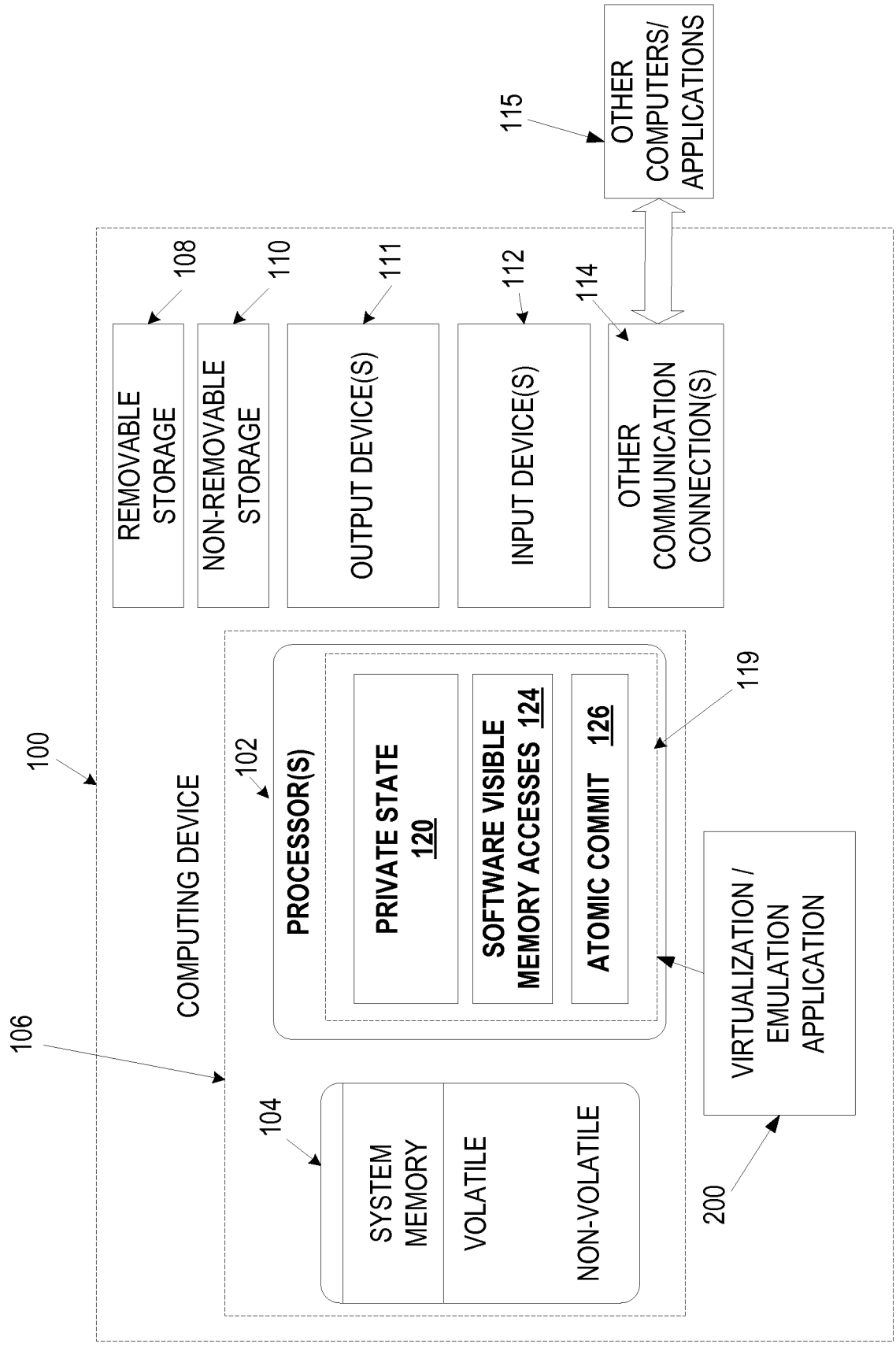

100
COMPUTING DEVICE
106
102
PROCESSOR(S)
PRIVATE STATE 120
SOFTWARE VISIBLE MEMORY ACCESSES 124
ATOMIC COMMIT 126
119
104
SYSTEM MEMORY
VOLATILE
NON-VOLATILE
200
VIRTUALIZATION / EMULATION APPLICATION
108
REMOVABLE STORAGE
110
NON-REMOVABLE STORAGE
111
OUTPUT DEVICE(S)
112
INPUT DEVICE(S)
114
OTHER COMMUNICATION CONNECTION(S)
115
OTHER COMPUTERS/ APPLICATIONS

FIG. 1

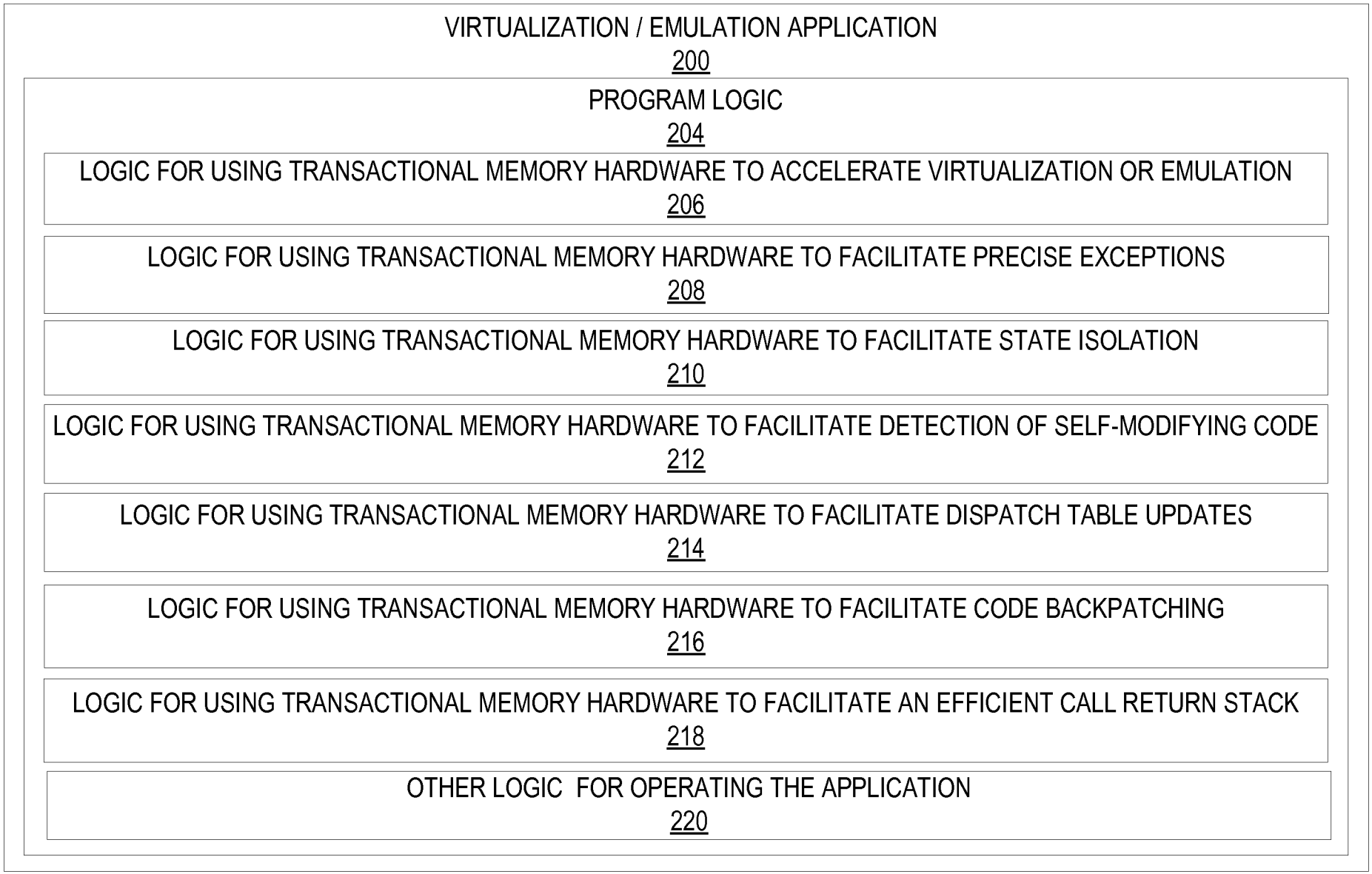

VIRTUALIZATION / EMULATION APPLICATION
200
PROGRAM LOGIC
204
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO ACCELERATE VIRTUALIZATION OR EMULATION
206
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO FACILITATE PRECISE EXCEPTIONS
208
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO FACILITATE STATE ISOLATION
210
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO FACILITATE DETECTION OF SELF-MODIFYING CODE
212
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO FACILITATE DISPATCH TABLE UPDATES
214
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO FACILITATE CODE BACKPATCHING
216
LOGIC FOR USING TRANSACTIONAL MEMORY HARDWARE TO FACILITATE AN EFFICIENT CALL RETURN STACK
218
OTHER LOGIC FOR OPERATING THE APPLICATION
220

FIG. 2

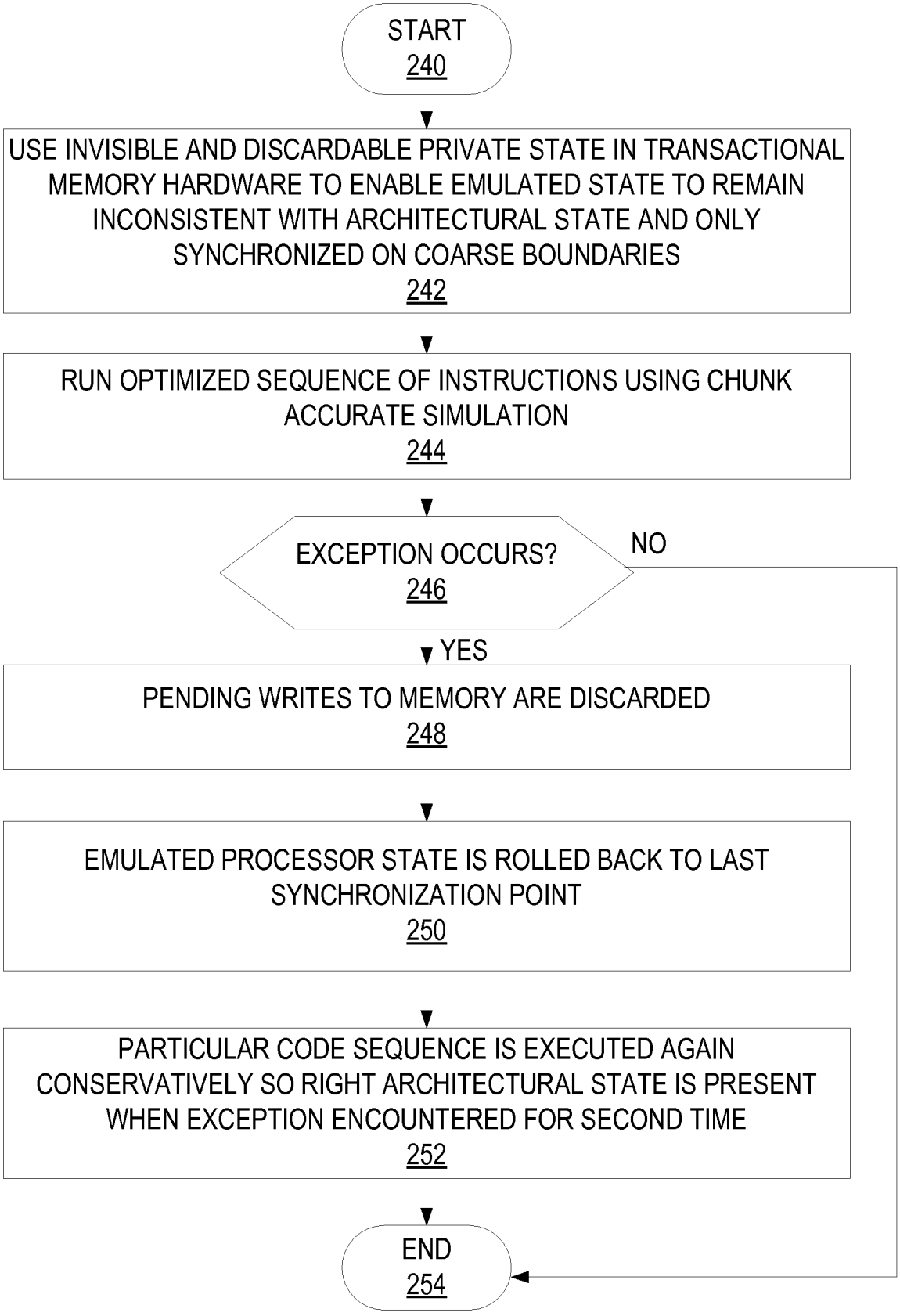

START
240
USE INVISIBLE AND DISCARDABLE PRIVATE STATE IN TRANSACTIONAL MEMORY HARDWARE TO ENABLE EMULATED STATE TO REMAIN INCONSISTENT WITH ARCHITECTURAL STATE AND ONLY SYNCHRONIZED ON COARSE BOUNDARIES
242
RUN OPTIMIZED SEQUENCE OF INSTRUCTIONS USING CHUNK ACCURATE SIMULATION
244
EXCEPTION OCCURS?
246
NO
YES
PENDING WRITES TO MEMORY ARE DISCARDED
248
EMULATED PROCESSOR STATE IS ROLLED BACK TO LAST SYNCHRONIZATION POINT
250
PARTICULAR CODE SEQUENCE IS EXECUTED AGAIN CONSERVATIVELY SO RIGHT ARCHITECTURAL STATE IS PRESENT WHEN EXCEPTION ENCOUNTERED FOR SECOND TIME
252
END
254

FIG. 3

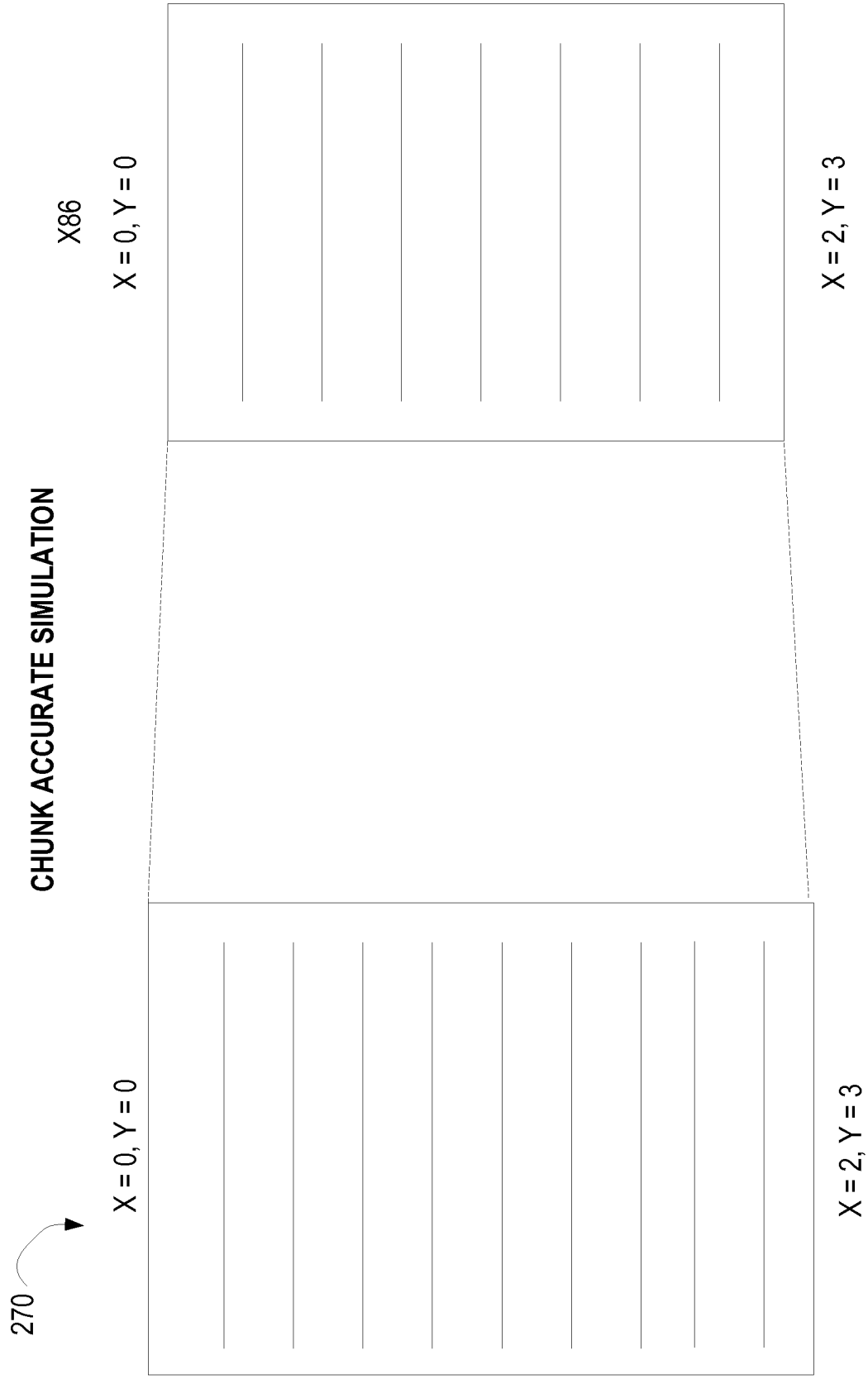

270
X = 0, Y = 0
X = 2, Y = 3
CHUNK ACCURATE SIMULATION
X86
X = 0, Y = 0
X = 2, Y = 3
RUN OPTIMIZED SEQUENCE OF INSTRUCTIONS TO TRY AND ACHIEVE
SAME END EFFECT

FIG. 4

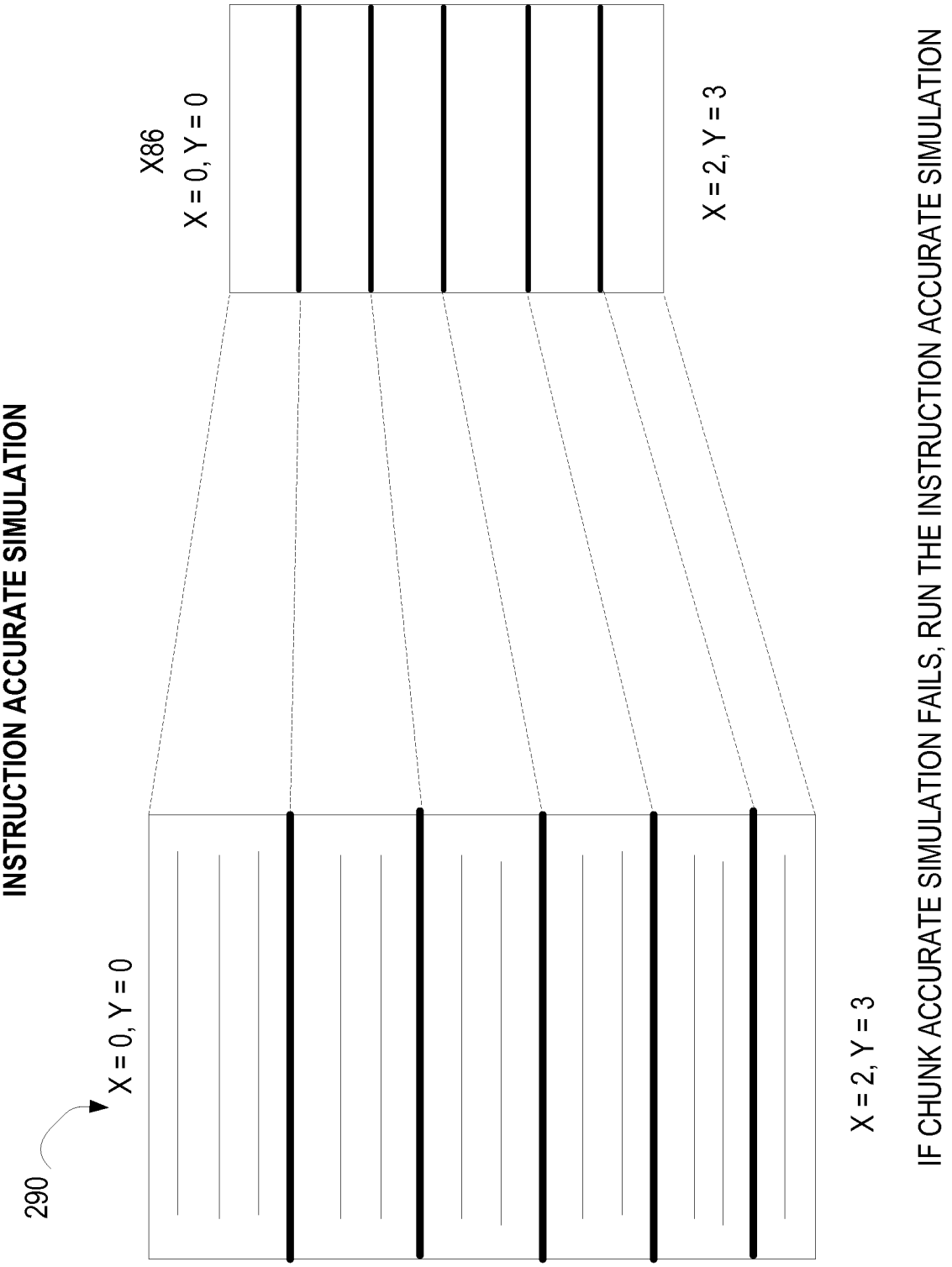

INSTRUCTION ACCURATE SIMULATION
290
X = 0, Y = 0
X = 2, Y = 3
X86
X = 0, Y = 0
X = 2, Y = 3
IF CHUNK ACCURATE SIMULATION FAILS, RUN THE INSTRUCTION ACCURATE SIMULATION

FIG. 5

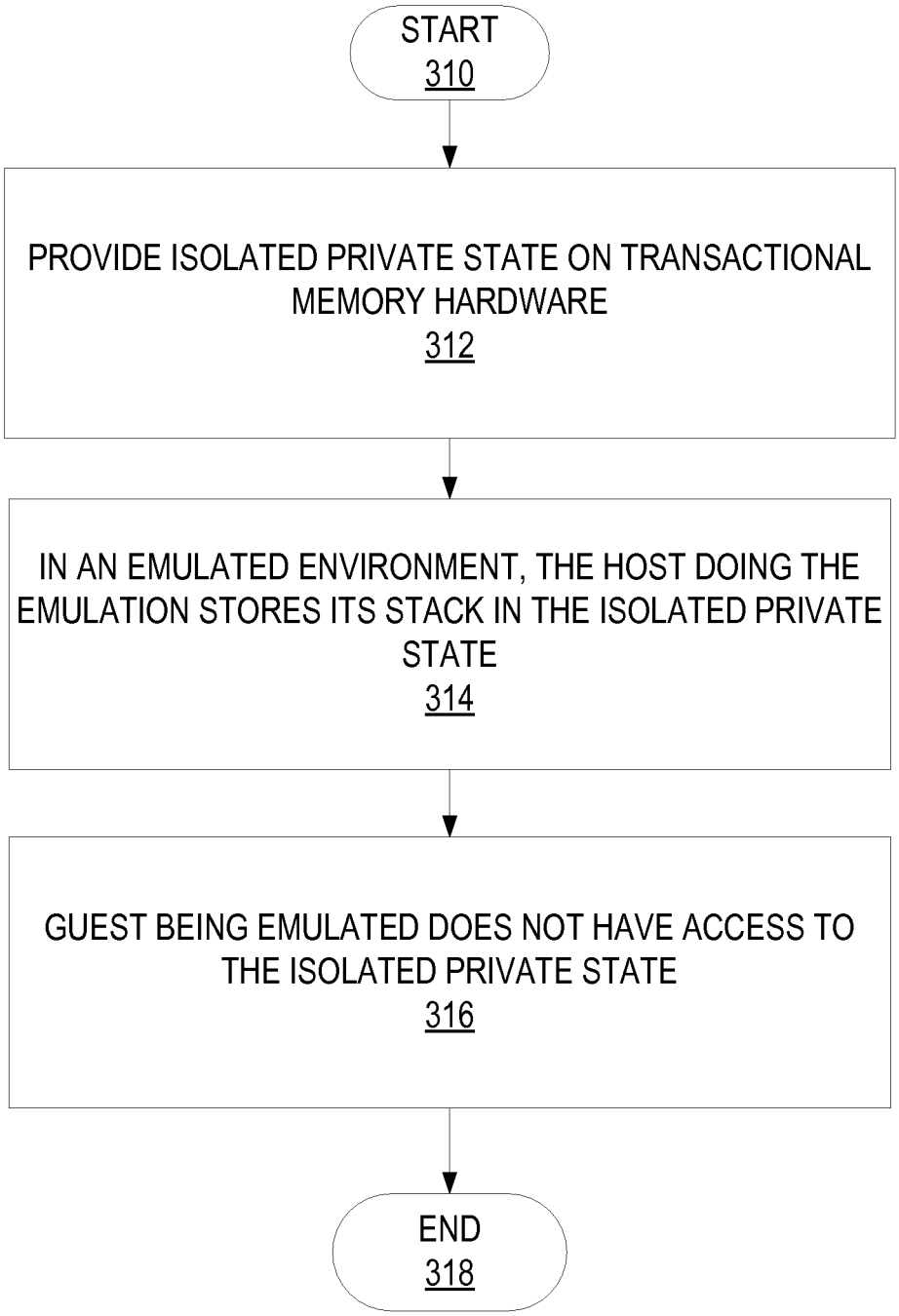

START
310
PROVIDE ISOLATED PRIVATE STATE ON TRANSACTIONAL MEMORY HARDWARE
312
IN AN EMULATED ENVIRONMENT, THE HOST DOING THE EMULATION STORES ITS STACK IN THE ISOLATED PRIVATE STATE
314
GUEST BEING EMULATED DOES NOT HAVE ACCESS TO THE ISOLATED PRIVATE STATE
316
END
318

FIG. 6

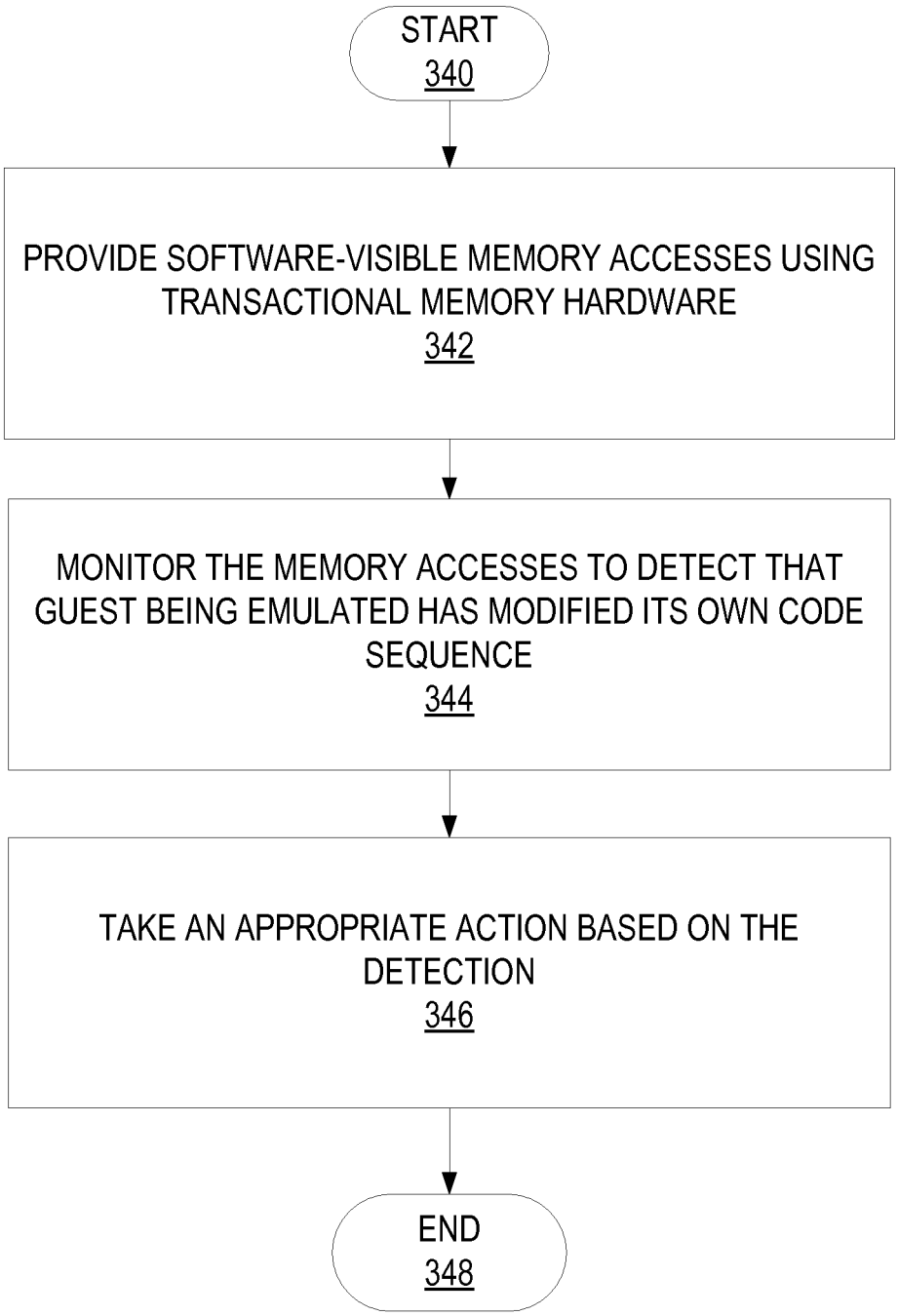

START
340
PROVIDE SOFTWARE-VISIBLE MEMORY ACCESSES USING TRANSACTIONAL MEMORY HARDWARE
342
MONITOR THE MEMORY ACCESSES TO DETECT THAT GUEST BEING EMULATED HAS MODIFIED ITS OWN CODE SEQUENCE
344
TAKE AN APPROPRIATE ACTION BASED ON THE DETECTION
346
END
348

FIG. 7

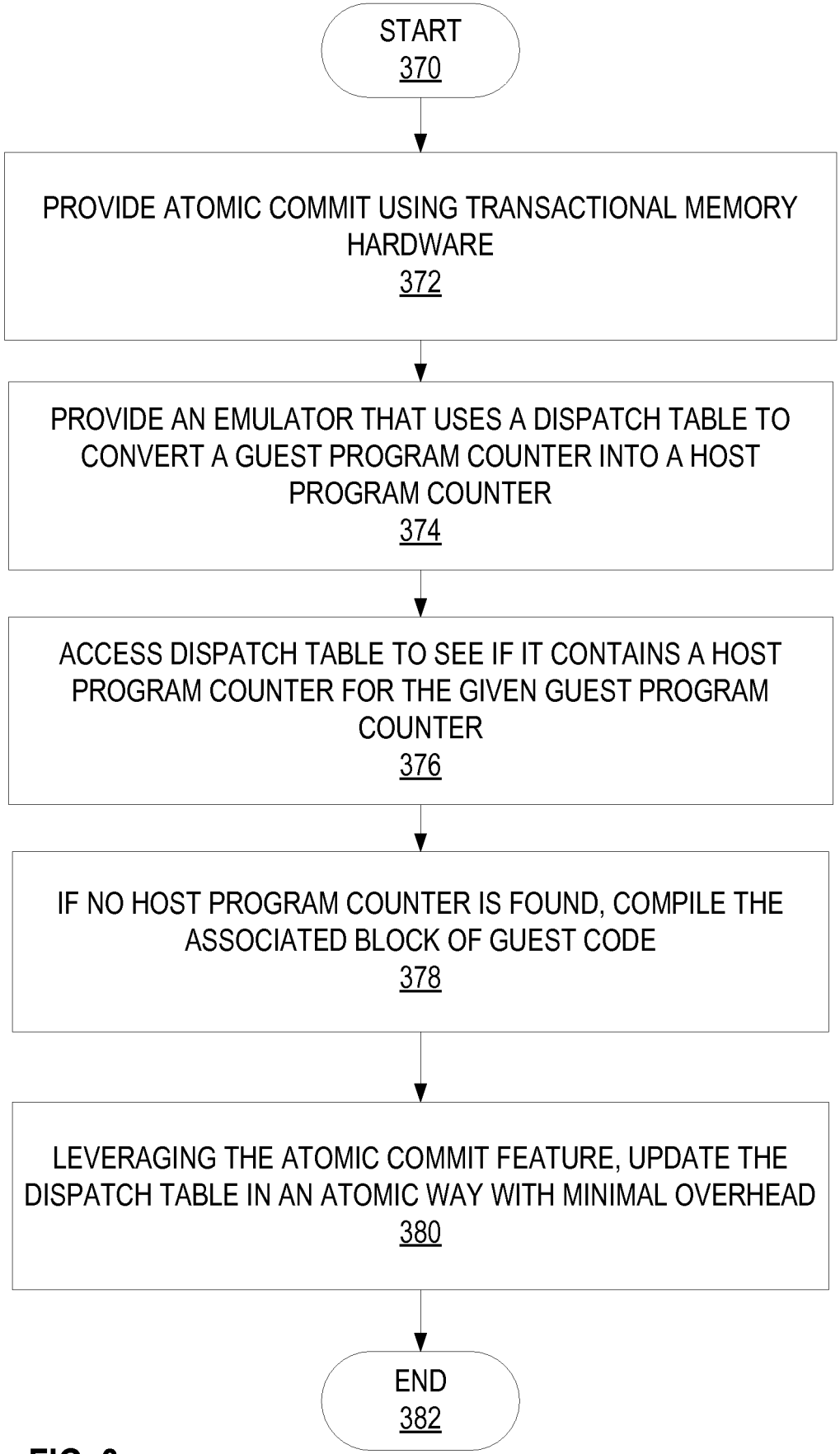

START
370
PROVIDE ATOMIC COMMIT USING TRANSACTIONAL MEMORY HARDWARE
372
PROVIDE AN EMULATOR THAT USES A DISPATCH TABLE TO CONVERT A GUEST PROGRAM COUNTER INTO A HOST PROGRAM COUNTER
374
ACCESS DISPATCH TABLE TO SEE IF IT CONTAINS A HOST PROGRAM COUNTER FOR THE GIVEN GUEST PROGRAM COUNTER
376
IF NO HOST PROGRAM COUNTER IS FOUND, COMPILE THE ASSOCIATED BLOCK OF GUEST CODE
378
LEVERAGING THE ATOMIC COMMIT FEATURE, UPDATE THE DISPATCH TABLE IN AN ATOMIC WAY WITH MINIMAL OVERHEAD
380
END
382

FIG. 8

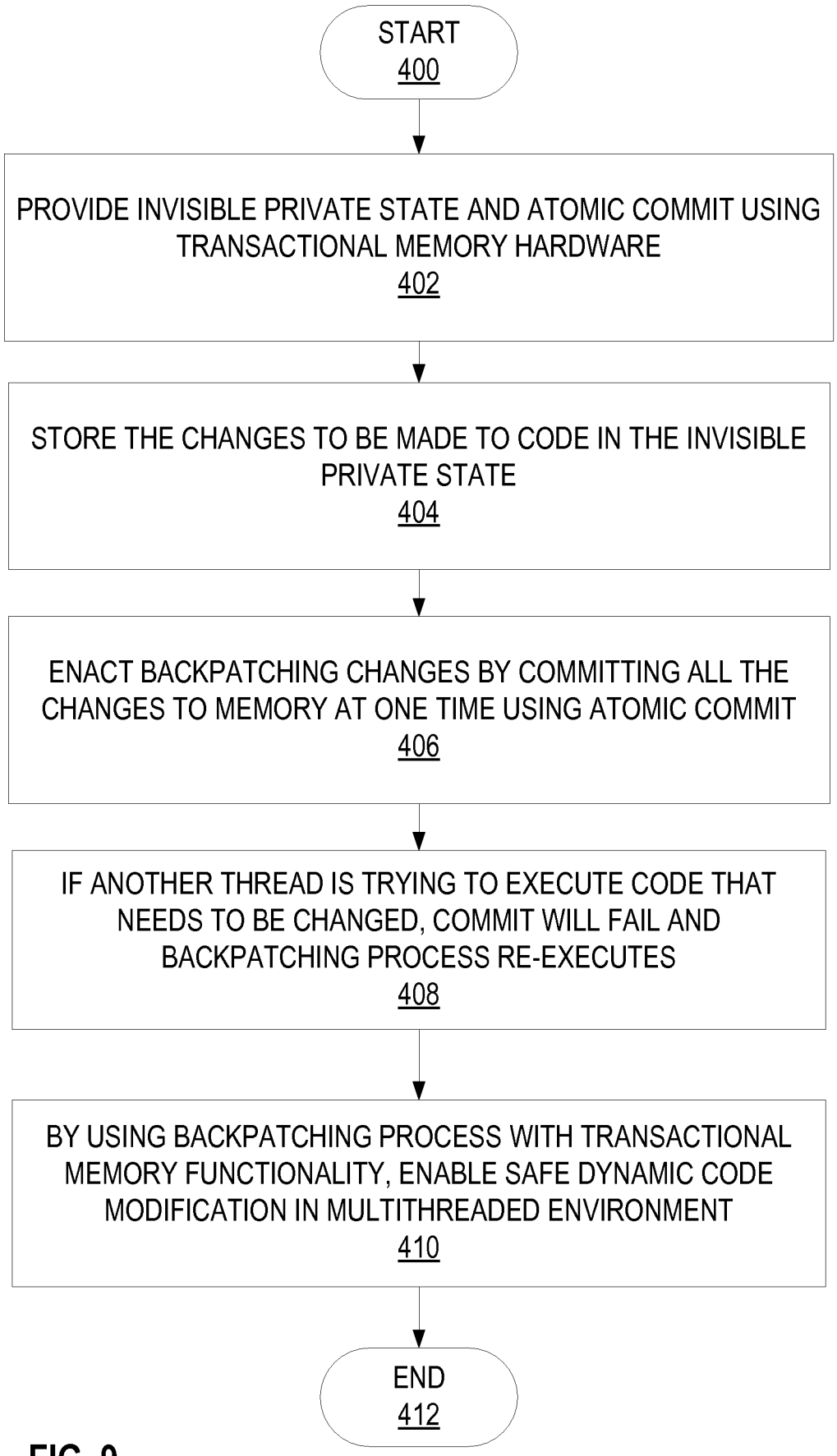

START
400
PROVIDE INVISIBLE PRIVATE STATE AND ATOMIC COMMIT USING TRANSACTIONAL MEMORY HARDWARE
402
STORE THE CHANGES TO BE MADE TO CODE IN THE INVISIBLE PRIVATE STATE
404
ENACT BACKPATCHING CHANGES BY COMMITTING ALL THE CHANGES TO MEMORY AT ONE TIME USING ATOMIC COMMIT
406
IF ANOTHER THREAD IS TRYING TO EXECUTE CODE THAT NEEDS TO BE CHANGED, COMMIT WILL FAIL AND BACKPATCHING PROCESS RE-EXECUTES
408
BY USING BACKPATCHING PROCESS WITH TRANSACTIONAL MEMORY FUNCTIONALITY, ENABLE SAFE DYNAMIC CODE MODIFICATION IN MULTITHREADED ENVIRONMENT
410
END
412

FIG. 9

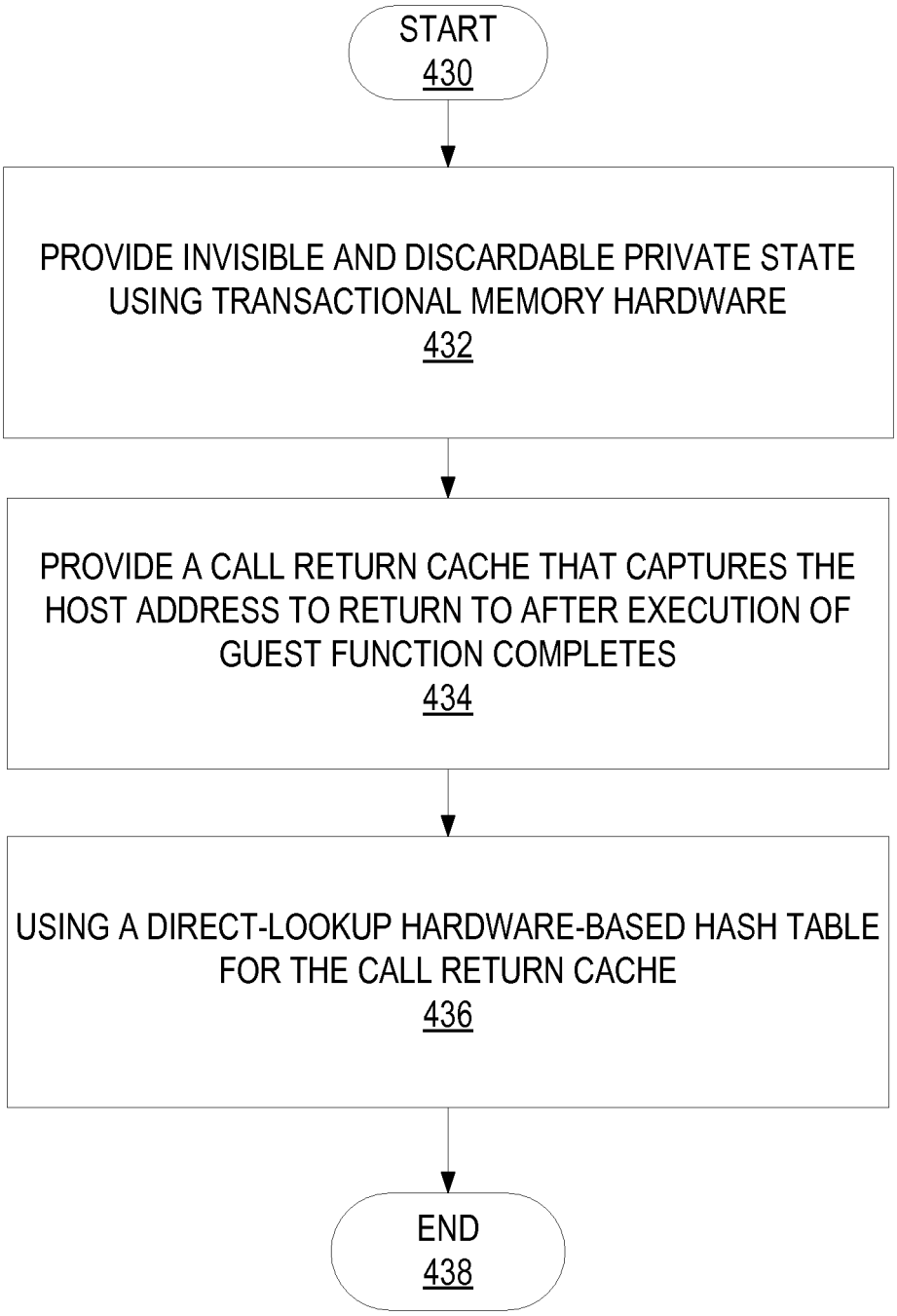

START
430
PROVIDE INVISIBLE AND DISCARDABLE PRIVATE STATE USING TRANSACTIONAL MEMORY HARDWARE
432
PROVIDE A CALL RETURN CACHE THAT CAPTURES THE HOST ADDRESS TO RETURN TO AFTER EXECUTION OF GUEST FUNCTION COMPLETES
434
USING A DIRECT-LOOKUP HARDWARE-BASED HASH TABLE FOR THE CALL RETURN CACHE
436
END
438

FIG. 10