US 20140189310A1

(19) **United States**
(12) **Patent Application Publication**
Tuck et al.
(10) **Pub. No.: US 2014/0189310 A1**
(43) **Pub. Date: Jul. 3, 2014**

(54) **FAULT DETECTION IN INSTRUCTION TRANSLATIONS**

(71) Applicant: **NVIDIA CORPORATION**, Santa Clara, CA (US)

(72) Inventors: **Nathan Tuck**, Corvallis, OR (US); **David Dunn**, Sammamish, WA (US); **Ross Segelken**, Portland, OR (US); **Madhu Swarna**, Portland, OR (US)

(73) Assignee: **NVIDIA CORPORATION**, Santa Clara, CA (US)

(21) Appl. No.: **13/728,669**

(22) Filed: **Dec. 27, 2012**

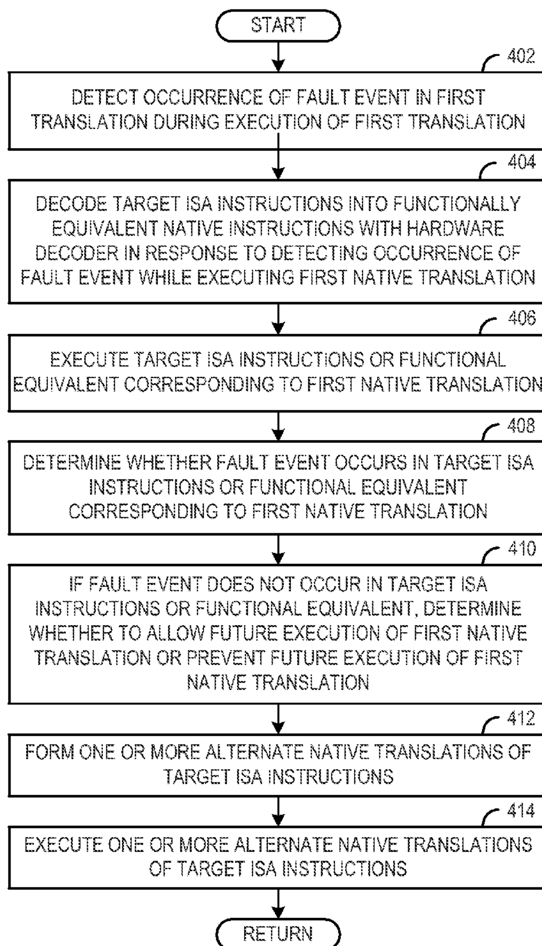
Publication Classification

(51) **Int. Cl.**
G06F 9/30 (2006.01)

(52) **U.S. Cl.**
CPC *G06F 9/30145* (2013.01)
USPC **712/209**

(57) **ABSTRACT**

In one embodiment, a method for identifying and replacing code translations that generate spurious fault events includes detecting, while executing a first native translation of target instruction set architecture (ISA) instructions, occurrence of a fault event, executing the target ISA instructions or a functionally equivalent version thereof, determining whether occurrence of the fault event is replicated while executing the target ISA instructions or the functionally equivalent version thereof, and in response to determining that the fault event is not replicated, determining whether to allow future execution of the first native translation or to prevent such future execution in favor of forming and executing one or more alternate native translations.



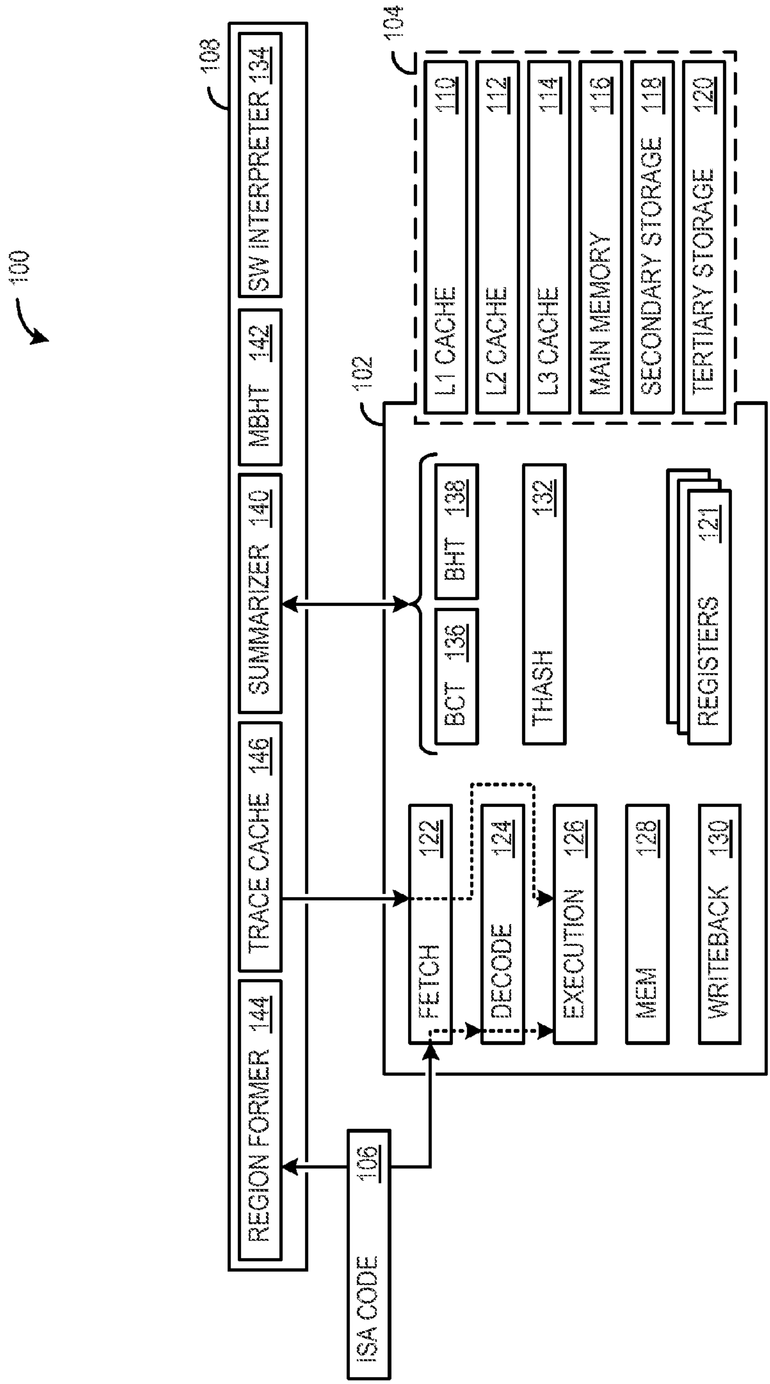


FIG. 1

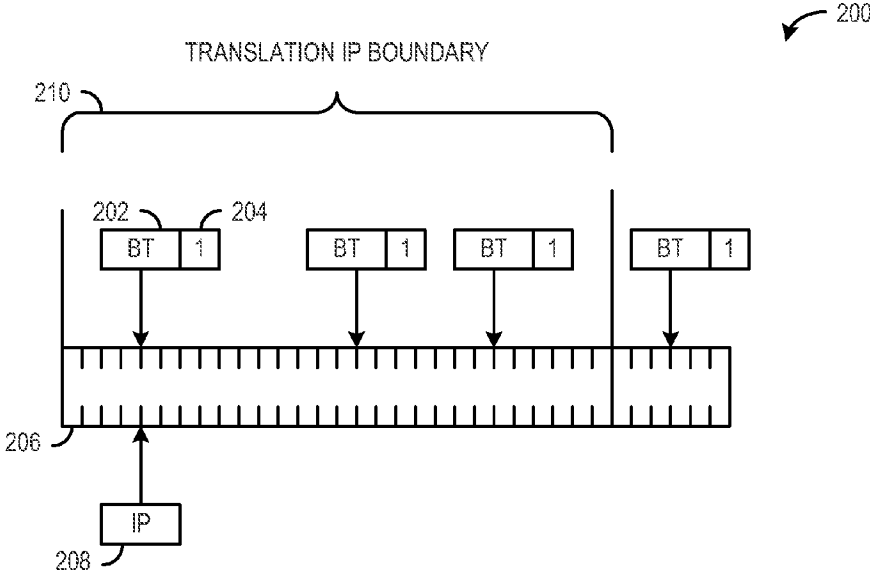


FIG. 2

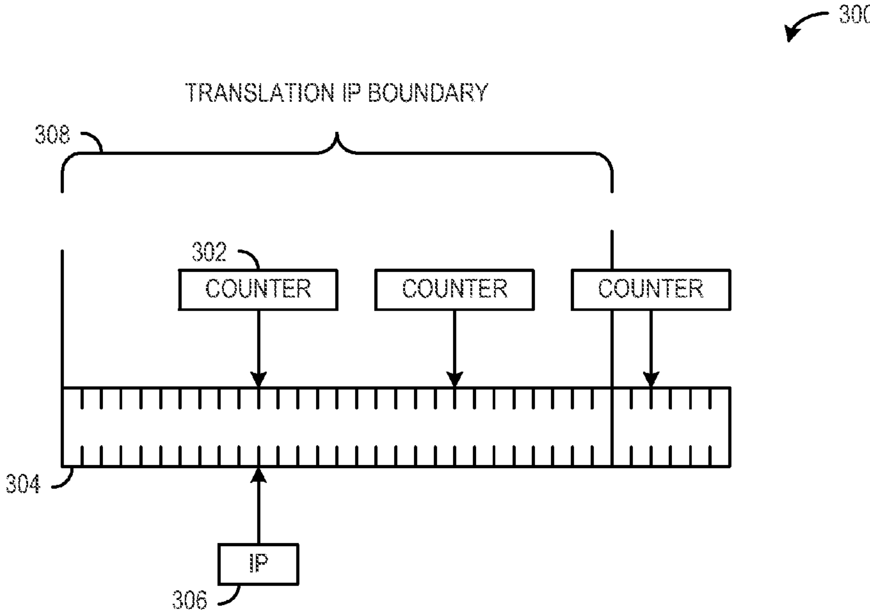


FIG. 3

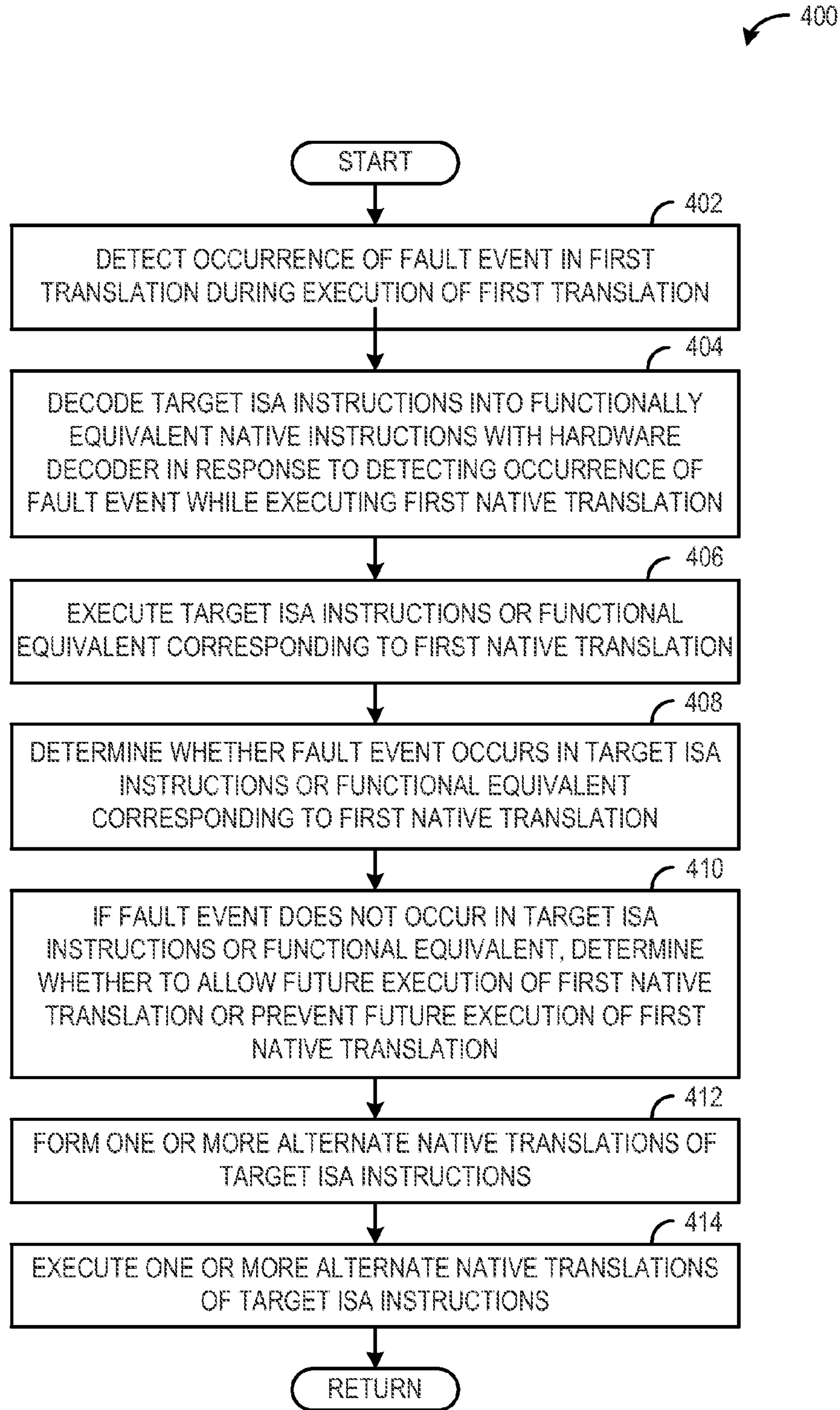


FIG. 4

FAULT DETECTION IN INSTRUCTION TRANSLATIONS

BACKGROUND

[0001] Some computing systems implement translation software to translate portions of target instruction set architecture (ISA) instructions into native instructions that may be executed more quickly and efficiently through various optimization techniques such as combining, reorganizing, and eliminating instructions. More particularly, in transactional computing systems that have the capability to speculate and rollback operations, translations may be optimized in ways that potentially violate the semantics of the target ISA. Due to such optimizations, once a translation has been generated, it can be difficult to distinguish whether events (e.g., architectural fault such as a page violation) encountered while executing a translation are architecturally valid or are spuriously created by over-optimization of the translation.

BRIEF DESCRIPTION OF THE DRAWINGS

[0002] FIG. **1** schematically shows an example computing system in accordance with an embodiment of the present disclosure.

[0003] FIG. **2** shows an example of a trap mechanism for pausing execution in order to determine whether a fault event is spuriously created by a translation.

[0004] FIG. **3** shows an example of a counter mechanism for pausing execution in order determine whether a fault event is spuriously created by a translation.

[0005] FIG. **4** shows an example of a method for identifying and replacing code translations that generate spurious fault events in accordance with an embodiment of the present disclosure.

DETAILED DESCRIPTION

[0006] The present disclosure provides a mechanism for optimizing native translations of corresponding non-native code portions, such as target instruction set architecture (ISA) code portions. The intelligent generation of translations, and the optimization thereof, may be handled by translation software, which may be included as part of a software layer that provides an interface between an ISA and a processor core. More particularly, the present disclosure provides a fault narrowing mechanism that identifies and replaces code translations that generate spurious fault events (e.g., architectural faults). As discussed above, in some cases, a translation may be aggressively or overly optimized such that the translation generates spurious fault events. Note that "spurious" means that if the corresponding target ISA code or a functional equivalent thereof were executed, then the fault event would not occur. In other cases, a fault event may be generated by the target ISA code. The mechanism determines whether a fault event encountered in a translation is generated spuriously by the translation, for example due to over-optimization of the translation, and if it is determined that the fault event was spuriously caused by the translation, it generates a different translation.

[0007] In one example, the translation software redirects execution to an instruction pointer (IP) of a native translation in lieu of corresponding target ISA code by the processor core. The native translation may be executed without using a hardware decoder located on the processor core. Note that when this disclosure refers to execution "without using the hardware decoder," that language may still encompass minor or trivial uses of decode logic in hardware while a translation is being executed. Circumventing the hardware decoder (i.e., by executing a translation) in many cases will improve speed of execution, reduce power consumption, and provide various other benefits. During execution of the native translation, a fault may be encountered. At this point, it is unknown whether the fault is an actual architectural event or if it is an artifact of the way that the code has been optimized in the translation. As such, execution is rolled back to a committed state (e.g., through a checkpoint mechanism), and a different version of code corresponding to the translation that does not produce the artifact event is executed. In one example, the alternate version of the code corresponding to the translation is target ISA code that is decoded by a hardware decoder into native instructions. If the fault is encountered during execution of the alternate code, then it is concluded that the translation itself was not the cause of the fault. If the fault is not encountered during execution of the alternate code, then it is concluded that the translation generated the artifact, and it is determined whether to allow future execution of the native translation or to prevent such future execution in favor of forming and executing one or more alternate native translations. In some embodiments, the translation is reformed in a different way, and the reformed translation is executed subsequently. In one example, the translation is reformed with fewer optimizations so as not to cause the fault during execution.

[0008] By using this mechanism, a translation can be aggressively over-optimized, then quickly narrowed if necessary using the hardware decoder to get to a translation that is suitably optimized to be executed without generating fault events. Implementations without this mechanism would find the overhead of narrowing or re-optimizing to be high enough that translations would tend to be overly conservative or under-optimized to avoid the narrowing process. For example, a software interpreter may be adequate to isolate an architectural event or lack thereof, but would be obtrusively slow for narrowing and re-optimizing as the software interpreter can require hundreds of native instructions to emulate a single target ISA instruction.

[0009] FIG. **1** shows aspects of an example micro-processing and memory system **100** (e.g., a central processing unit or graphics processing unit of a personal computer, game system, smartphone, etc.) including processor core **102**. Although the illustrated embodiment includes only one processor core, it will be appreciated that the micro-processing system may include additional processor cores in what may be referred to as a multi-core processing system. Microprocessor core/die **102** variously includes and/or may communicate with various memory and storage locations **104**. In some cases, it will be desirable to allocate a portion (sometimes referred to as a "carveout") of memory as secure and private such that it is invisible to the user and/or instruction set architecture (ISA) code **106**. Various data and software may run from, and/or be stored in said allocation, such as software layer **108** and related software structures. As will be discussed in greater detail below, the software layer may be configured to generate, optimize, and store translations of ISA code **106**, and further to manage and interact with related hardware on core **102** to determine whether translations are suitably optimized (e.g., the translations do not generate faults or other artifacts).

[0010] Memory and storage locations **104** may include L1 processor cache **110**, L2 processor cache **112**, L3 processor cache **114**, main memory **116** (e.g., one or more DRAM chips), secondary storage **118** (e.g., magnetic and/or optical storage units) and/or tertiary storage **120** (e.g., a tape farm). Processor core **102** may further include processor registers **121**. Some or all of these locations may be memory-mapped, though in some implementations the processor registers may be mapped differently than the other locations, or may be implemented such that they are not memory-mapped. It will be understood that the memory/storage components are listed above in increasing order of access time and capacity, though there are possible exceptions. In some embodiments, a memory controller may be used to handle the protocol and provide the signal interface required of main memory **116**, and, typically, to schedule memory accesses. The memory controller may be implemented on the processor die or on a separate die. It is to be understood that the locations set forth above are non-limiting and that other memory/storage locations may be used instead of, or in addition to, those described above without departing from the scope of this disclosure.

[0011] Microprocessor **102** includes a processing pipeline which typically includes one or more of fetch logic **122**, decode logic **124** (referred to herein as a hardware decoder or hardware decode logic (HWD)), execution logic **126**, mem logic **128**, and writeback logic **130**. Note that one or more of the stages in the processing pipeline may be individually pipelined to include a plurality of stages to perform various associated operations. It should be understood that these five stages are somewhat specific to, and included in, a typical RISC implementation. More generally, a microprocessor may include fetch, decode, and execution logic, with mem and writeback functionality being carried out by the execution logic. The present disclosure is equally applicable to these and other microprocessor implementations, including hybrid implementations that may use VLIW instructions and/or other logic instructions.

[0012] Fetch logic **122** retrieves instructions from one or more of memory locations **104** (e.g., unified or dedicated L1 caches backed by L2-L3 caches and main memory). In some examples, instructions may be fetched and executed one at a time, possibly requiring multiple clock cycles.

[0013] Microprocessor **102** is configured to execute instructions, via execution logic **126**. Such instructions are generally described and defined by an ISA that is native to the processor, which may be generated and/or executed in different modes of operation of the microprocessor. A first mode (referred to herein as the "hardware decoder mode") of execution includes utilizing the HWD **124** to receive and decode (e.g., by parsing opcodes, operands, and addressing modes, etc.) target ISA or non-native instructions of ISA code **106** into native instructions for execution via the execution logic. It will be appreciated that the native instructions dispatched by the HWD may be functionally equivalent to the non-native instructions, in that execution of either type of instructions achieves the same final result or outcome.

[0014] A second mode (referred to herein as the "translation mode") of execution includes retrieving and executing native instructions without use of the HWD. A native translation may cover and provide substantially equivalent functionality for any number of portions of corresponding target ISA or non-native ISA code **106**. The corresponding native translation is typically optimized to some extent by the translation software relative to the corresponding non-native code that would be dispatched by the HWD. However, it will be understood that a variety of optimizations and levels of optimization may be employed.

[0015] A third mode (referred to herein as "software interpretation mode") of execution includes utilizing a software interpreter **134** located in the software layer **108** to execute target ISA code one instruction at a time by translating the target ISA instruction into corresponding native instructions.

[0016] Typically, translation mode provides the fastest and most efficient operation out of the above described execution modes. However, there may be substantial overhead costs associated with generating an optimized translation of target ISA instructions. Accordingly, a translation may be generated for portions of target ISA code that are executed frequently or consume substantial processing time, such as frequently used or "hot" loops or functions in order to control such translation overhead. In one example, a translation may be generated for a portion of target ISA code in response to the portion of code being executed a number of times that is greater than a threshold value.

[0017] Hardware decoder mode may be slower or less efficient than translation mode and faster or more efficient than software interpretation mode. For example, hardware decoder mode may be used to execute portions of target ISA code that do not have corresponding translations. As another example, hardware decoder mode may be used to determine whether or not a translation is over-optimized based on encountering a fault during execution of a translation as will be discussed in further detail below.

[0018] Software interpretation mode may be used in corner cases or other unusual/rare circumstances, such as to isolate a fault or lack of a fault. The software interpretation mode may be used least frequently of the above described modes of operation, because the software interpretation mode may be substantially slower than the other modes of operation. For example, software interpretation mode may require hundreds of native instructions to emulate a single target ISA instruction.

[0019] For the sake of clarity, the native instructions output by the HWD in hardware decoder mode will in some cases be referred to as non-translated instructions, to distinguish them from the native translations that are executed in the translation mode without use of the HWD.

[0020] Native translations may be generated in a variety of ways. As discussed above, due to the high overhead of generating translations, in some embodiments, code portions of non-native ISA code may be profiled in order to identify whether and how those code portions should be included in new or reformed translations. When operating in hardware decoder mode, the system may dynamically change and update a code portion profile in response to the use of the HWD to execute a portion of non-native ISA code. For example, profiled code portions may be identified and defined by taken branches. This is but one example, however, and any suitable type of code portion associated definition may be used.

[0021] In certain embodiments, the code portion profile is stored in an on-core micro-architectural hardware structure (e.g., on core **102**), to enable rapid and lightweight profiling of code being processed with the HWD. For example, the system may include a branch count table (BCT) **136** and a branch history table (BHT) **138** each including a plurality of records containing information about code portions of non-native ISA code **106** encountered by the HWD as branch

instructions are processed. In general, the BCT tracks the number of times a branch target address is encountered, while the BHT records information about the taken branch when a branch target address is encountered. Furthermore, the BCT is used to trigger profiling for translation upon saturation of a particular code portion. For example, the BCT may be used to determine whether a code portion has been executed a number of times that exceeds a threshold value, which triggers reforming of a corresponding translation.

[0022] As the code portions of non-native ISA code are processed by HWD, records may be dynamically added to BCT and BHT. For example, as the HWD processes taken branches leading to a branch target address, a record for that branch target address is added to the BCT and an initial value is inserted into a counter associated with the record. Alternatively, if a record already exists for the target address, the counter is incremented or decremented, as appropriate to the implementation. As such, the system may include micro-architectural logic for adding and updating records in the BCT and the BHT. This logic may be a distinct component or distributed within various components of the processing pipeline, though typically this logic will be operatively coupled closely with the HWD since it is the use of the HWD that results in changes to the BCT and the BHT.

[0023] From time to time, the records of BCT and/or BHT may be sampled and processed, for example by a summarizer **140** of software layer **108**. As described above, the software layer may reside in a secure/private memory allocation of storage locations **104** that is accessible by microprocessor **102** during execution of native ISA instructions. In other words, such an allocation may be inaccessible by ISA code.

[0024] The summarizer may be implemented as a light-weight event handler that is triggered when a record in the BCT produces an event (e.g., the counter for the record saturates). In other words, the BCT produces an event, and the summarizer handles the event (e.g., by sampling and processing records in the BHT). Each counter maintained in the BCT for a target address is used to control how many times the associated code portion will be encountered before an event is taken for that code portion.

[0025] The summarizer identifies flow into, out of, and/or between code portions when using the hardware decoder. Furthermore, the summarizer identifies one or more non-translated code portions to be included in a new native translation by producing a summarized representation (e.g., a control flow graph) of code portion control flow involving the HWD. For example, the sampling and processing by the summarizer may be used to generate and update a meta branch history table (MBHT) **142** in and between non-native code portions processed by the HWD. It will be appreciated that information about code portions and control flow may be represented in any suitable manner, data structure, etc. The information in the MBHT is subsequently consumed by a region former **144**, which is responsible for forming new translations of non-native ISA code. Once formed, translations may be stored in one or more locations (e.g., a trace cache **146** of software layer **108**). The region former may employ various optimization techniques in creating translations, including, but not limited to, reordering instructions, renaming registers, consolidating instructions, removing dead code, unrolling loops, etc. It will be understood that these translations may vary in length and the extent to which they have been optimized. For example, the region former may vary the aggressiveness at which a translation is optimized in order to strike a balance between increasing performance and generating spurious architectural events or artifacts during execution. It will be appreciated that the structures stored in the software layer may be included in or collectively referred to herein as a translation manager or as translation management software.

[0026] During operation, the existence of a translation may be determined using an on-core hardware redirector **132** (a.k.a., a THASH). The hardware redirector is a micro-architectural structure that includes address information or mappings sufficient to allow the processing pipeline to retrieve and execute a translation or a portion thereof associated with a non-native portion of ISA code via address mapping. Specifically, when the processing pipe branches to a target address of a non-native portion of ISA code, the target address is looked up in the THASH. Over time, translations that are frequently and/or recently requested are indexed by, and incorporated into, the hardware redirector. Each entry in the hardware redirector is associated with a translation, and provides redirection information that enables the microprocessor, during a fetch operation for a selected code portion, to cause execution to be redirected away from that code portion and to its associated translation. In order to save on processor die area and to provide rapid lookups, the hardware redirector may be of limited size, and it is therefore desirable that it be populated with entries providing redirection for the most "valuable" translations (e.g., translations that are more frequently and/or recently used). Accordingly, the hardware redirector may include usage information associated with the entries. This usage information varies in response to the hardware structure being used to redirect execution, and thus the entries are maintained in, or evicted from, the hardware redirector based on this usage information.

[0027] In the event of a hit in the THASH, the lookup returns the address of an associated translation (e.g., translation stored in trace cache **146**), which is then executed in translation mode (i.e., without use of HWD **124**). Alternatively, in the event of a miss in the THASH, the portion of code may be executed through a different mode of operation and one or more of the mechanisms described above may be usable to generate a translation. The THASH lookup may therefore be usable to determine whether to add/update records in BCT and BHT. In particular, a THASH hit means that there is already a translation for the non-native target code portion, and there is thus no need to profile execution of that portion of target code in hardware decoder mode. Note that the THASH is merely one example of a mechanism for locating and executing translations, and it will be appreciate that the processor hardware and/or software may include other mechanisms for locating and executing translations without departing from the scope of the present description.

[0028] Throughout operation, a state of the microprocessor (e.g., registers **121** and/or other suitable states) may be checkpointed or stored to preserve the state of the microprocessor while a non-checkpointed working state version of the microprocessor speculatively executes instructions. For example, the state of the microprocessor may be checkpointed when execution of an instruction (or bundle, code portion of a translation, etc.) is completed without encountering an architectural event, artifact, exception, fault, etc. For example, an architectural event (e.g., a fault event) may include a page violation, a memory alignment violation, a memory ordering violation, a break point, execution of an illegal instruction, etc. If a fault event is encountered during execution, then the

instruction may be rolled back, and state of the microprocessor may be restored to the checkpointed state. Then operation may be adjusted to handle the fault event. For example, the microprocessor may operate in hardware decoder mode and mechanisms for determining whether the encountered event is an artifact of the translation may be employed. In one example, the decode logic is configured to manage checkpointing/rollback/restore operations. Although it will be appreciated that in some embodiments a different logical unit may control such operations. In some embodiments, checkpointing/rollback/restore schemes may be employed in connection with the memory and storage locations **104** in what may be generally referred to as transactional memory. In other words, microprocessor **102** may be a transaction-based system.

[0029] Furthermore, during execution of a native translation, the execution logic may be configured to detect occurrence of a fault event in the native translation. Since at the time of encountering the fault event, it may not be known whether or not the fault event is an artifact generated due to a particular way in which the native translation was formed, the translation manager causes the code portion to be executed differently. For example, the target ISA instructions or a functionally equivalent version thereof may be executed without executing the native translation to determine whether the fault event was a product of the native translation.

[0030] If a fault event is encountered in the translation, then the translation manager may note the IP boundaries of the translation before execution of the translation is rolled back. In some cases, the IP boundaries may include one contiguous portion of target ISA code. In other cases, the IP boundaries may include multiple non-contiguous portions of target ISA code (e.g., if the translation was formed including a target ISA branch that was assumed to be taken when the translation was generated). The IP boundaries may be used during execution of the target ISA instructions or a functionally equivalent version thereof to determine whether a fault event occurs in the code portion corresponding to the native translation.

[0031] In one example, the system may operate in hardware decoder mode to produce a functional equivalent of the target ISA instructions. In particular, the HWD receives target ISA instructions starting at the IP boundary corresponding to the beginning of the native translation, decodes them into native instructions, and dispatches the native instruction to the execution logic for execution. The native instructions may be executed by the execution logic until the fault event is encountered again, or execution leaves the code portion corresponding to the native translation (e.g., the IP is beyond the IP boundary corresponding to the end of the translation). Various mechanisms for determining whether execution has left the code portion corresponding to the native translation may be employed during operation in hardware decoder mode. Several non-limiting examples of such mechanisms are discussed in further detail below with reference to FIGS. **2** and **3**.

[0032] If the event is encountered during execution of the target ISA instruction or their functional equivalent (e.g., in the hardware decoder mode), it can be assumed that the event is an architectural fault that was not created by the translation, and redirection of control flow to the architectural exception vector is performed where control is passed to the translation manager or other architectural event handling logic to correct the architectural event or provide other event handling operations.

[0033] If execution leaves the translation without encountering the fault event, then it can be assumed that the native translation spuriously caused the fault event. In other words, the translation manager determines that the fault event is not replicated during execution of the target ISA instructions or the functionally equivalent version thereof. In response to determining that the fault event is not replicated, the translation manager is configured determine whether to allow future execution of the native translation or to prevent such future execution in favor of forming and executing one or more alternate native translations. Note that a future execution of the native translation may include any execution subsequent to determining that the native translation spuriously caused the fault event. The native translation may be prevented from being executed in order to reduce the likelihood of the fault event from occurring during subsequent executions of the target ISA instructions or the functionally equivalent version thereof. In some embodiments, the determination whether to allow future execution of the native translation or to prevent such future execution may include forming and executing the one or more alternate translations upon determining that a performance cost associated with forming the one or more alternate translations is less than a performance cost associated with continuing to execute the first native translation, executing the target ISA instructions or a functionally equivalent version thereof, without executing the first native translation, or a combination thereof. It will be appreciated that the performance costs may be calculated in any suitable manner without departing from the scope of the present disclosure.

[0034] In some embodiments, the native translation may be prevented from being executed immediately after determining that the native translation spuriously caused the fault event such that the translation is not executed again. For example, when the code portion corresponding to the native translation is encountered subsequent to the determination, the system may operate in hardware decoder mode to execute the code portion instead of executing the native translation. As another example, a different translation may be executed instead of the native translation.

[0035] In some embodiments, the native translation may be executed one or more times subsequent to the determination before the native translation is prevented from being executed. For example, the native translation may be executed subsequently in order to determine if the native translation spuriously causes any different faults. In one particular example, the native translation is not prevented from being executed until a first fault and a second fault are encountered a designated number of times as a result of executing the native translation. In other words, the native translation may be repeatedly executed until it can be assumed with a level of confidence that the native translation is the cause of a number of different faults before execution of the native translation is prevented.

[0036] In some cases, the system may operate in software interpreter mode instead of hardware decoder mode in response to encountering a fault event during execution of the native translation (e.g., to handle of corner cases). As discussed above, hardware decoder mode may be preferred over software interpreter mode for fault narrowing operation because the software interpreter mode may be significantly slower to execute the target ISA instructions. For example, the software interpreter may take over one hundred times longer to execute an instruction than the HWD may take to execute the same instruction.

[0037] In some embodiments, the translation manager may be configured to generate an updated or reformed translation of the target ISA instructions that is optimized differently based on encountering an artifact or fault event in the first translation. In one example, the reformed translation is optimized differently so as not to generate the architectural event. For example, the updated translation may include fewer optimizations than the previous translation, such as less combinations, reorganizations, and/or eliminations of target ISA instructions. Further, the execution logic may be configured to, upon subsequently encountering the code portion of the target ISA instructions, execute the updated or reformed translation instead of the previous translation that spuriously caused the fault event.

[0038] Since there may be substantial overhead costs associated with generating an optimized translation of target ISA instructions, in some embodiments, the translation manager may be configured to track activity related to the translation subsequent to determining that the fault was an artifact of the translation, and determine if or when it would be suitable to update the translation. In one example, the translation manager is configured to increment a counter associated with the native translation subsequent to determining that the fault event is an artifact of the translation. Further, the translation manager may generate the updated translation of the target ISA instructions responsive to the counter saturating or becoming greater than a threshold value. The counter may be employed to track or count a variety of different factors, events, parameters, or execution properties associated with the translation that spuriously caused the fault event. Non-limiting examples of these factors that the counter may track include time, a number of translation executions, a number of translation executions that spuriously cause a fault event, a number of translation execution that spuriously cause a number of different fault events. In some embodiments the counter may include a decision function that includes a combination of these factors.

[0039] It will be appreciated that the counter may be used to track any suitable parameter or event associated with the translation in order to determine if or when to reform the translation. Moreover, it will be appreciated that the counter is merely one example of a tracking mechanism, and any suitable mechanism may be employed to decide when to reform the translation.

[0040] In some embodiments, the translation manager may be configured to reform the translation (or generate a new translation) of only a subset of the target ISA instructions that were represented by the translation that spuriously created the fault. In some embodiments, the translation manager may be configured to generate a plurality of translations that span the target ISA instructions that were represented by the translation that spuriously created the fault.

[0041] FIGS. **2** and **3** show examples of various mechanisms that may be employed to pause execution during operation in hardware decoder mode in order to determine whether the code portion corresponding to the translation is executed without encountering the event, which may be used to determine whether an event is spuriously created by the translation. FIG. **2** shows an example of a mechanism **200** that causes execution to be paused responsive to encountering a target of a branch instruction that is dispatched by the HWD. In particular, when an event is encountered in translation mode, execution is rolled back to the beginning of the IP boundary **210** of the translation. The IP boundary defines the code region of the translation by denoting the IP at the beginning of the translation and the IP at the end of the translation. The translation manager calls the HWD to operate in hardware decoder mode with a particular jump instruction that includes a “sticky bit” **202** that is set based on encountering the event. By setting the sticky bit in the jump instruction that invokes the HWD, each branch causes a field **204** to be set that is associated with the branch target. The set bit is recognized upon execution of the branch target causing execution in hardware decoder mode to be paused. The set bit is cleared and control is passed from the HWD to the translation manager. The translation manager determines whether the IP is within the IP boundary of the code portion corresponding to the translation. If the IP is beyond the IP boundary of the translation, then the event was not encountered in the code portion at issue and it can be assumed that the event was an artifact of the translation, and the translation may need to be reformed in a different manner and the sticky bit is cleared. If the IP is within the IP boundary of the translation, then control is passed back to the HWD and execution in hardware decoder mode continues until another branch target having a set bit is encountered or the event is encountered. If the event is encountered, then it can be assumed that the event is not an artifact of the translation and the translation may not be over-optimized and the sticky bit is cleared.

[0042] The above described mechanism may be referred to as a “branch callback trap” because each time a branch target is encountered with a set bit, execution in hardware decoder mode is paused and control is passed to the translation manager. In other words, the sticky bit is the mechanism by which the translation manager gets passed control from hardware decoder mode. Note that when the HWD is called for operation other than when an event is encountered, the sticky bit in the jump instruction may be cleared to suppress the branch callback trap mechanism.

[0043] In some microprocessor implementations that include a hardware redirector or THASH that is accessed by the HWD to check for a translation, access to the THASH by the HWD is disabled or matches in the THASH are inhibited based on the event being encountered. In one example, access to the THASH is disabled when the sticky bit in the jump instruction that calls the HWD is set. By suppressing the lookup of the THASH, execution is not redirected to the translation so that execution in hardware decoder mode may be performed to determine whether the event is generated by the translation. In other words, access to the THASH is disabled when executing the target ISA instructions without executing the native translation.

[0044] FIG. **3** shows an example of a counter mechanism **300** that causes execution to be paused responsive to a counter expiring or elapsing. Similar to the above described example, when the HWD is called based on encountering an event during operation in translation mode, a counter **302** may be set, for example by setting a bit in a particular jump instruction that calls the HWD. During execution in hardware decoder mode, the counter counts down and when the counter expires execution is paused and control is passed to the translation manager. The translation manager determines whether the IP **306** is within the IP boundary **308** of the code portion corresponding to the translation. If the IP is beyond the IP boundary of the translation, then the event was not encountered in the code portion at issue and it can be assumed that the event was an artifact of the translation, and the translation may need to be reformed in a different manner. If the IP is

within the IP boundary of the translation, then control is passed back to the HWD and execution in hardware decoder mode continues until the counter expires again or the event is encountered. If the event is encountered, then it can be assumed that the event is not an artifact of the translation and the translation may not be over-optimized.

[0045] It will be appreciated that the counter may be set to any suitable duration or may track any suitable execution property or parameter. In one example, the counter may be set to for a designated number of clock cycles. In another example, the counter may be set for a designated number of instructions. In yet another example, the counter may expire in response to encountering a branch instruction. In still yet another example, the counter may expire in response to encountering a designated number of branch instructions.

[0046] It will be appreciated that the above described mechanisms may be particularly applicable to operation in hardware decoder mode, because control is passed from hardware (e.g., execution logic) to software (e.g., translation manager) when execution is paused to determine whether execution has left the IP boundary of the code portion at issue. Moreover, such mechanisms may allow for execution to be paused occasionally in order to perform an IP boundary check that allows for faster execution relative to an approach that checks after execution of each instruction.

[0047] FIG. **4** shows an example of a method **400** for optimizing a translation of target ISA instructions in accordance with an embodiment of the present disclosure. The method **400** may be implemented with any suitable software/hardware, including configurations other than those shown in the foregoing examples. In some cases, however, the process flows may reference components and processes that have already been described. For purposes of clarity and minimizing repetition, it may be assumed that these components/processes are similar to the previously described examples.

[0048] At **402**, the method **400** includes, detecting, while executing a first native translation of target ISA instructions, occurrence of a fault event in the first native translation. The first native translation may be executable to achieve substantially equivalent functionality as obtainable via execution of the target ISA instructions. In other words, the first native translation is designed such that execution of the first native translation should produce the same output as the target ISA instructions. In one example, the fault event includes one of a page violation, a memory alignment violation, a memory ordering violation, a break point, and execution of an illegal instruction.

[0049] At **404**, the method **400** includes decoding the target ISA instructions into functionally equivalent native instructions with a hardware decoder in response to detecting occurrence of the fault event while executing the first native translation;

[0050] At **406**, the method **400** includes executing the target ISA instructions or a functionally equivalent version thereof, where such execution is performed without executing the first native translation.

[0051] At **408**, the method **400** includes determining whether occurrence of the fault event is replicated while executing the target ISA instructions or the functionally equivalent version thereof.

[0052] At **410**, the method **400** includes in response to determining that the fault event is not replicated, determining whether to allow future execution of the first native translation or to prevent such future execution in favor of forming and executing one or more alternate native translations.

[0053] At **412**, the method **400** may optionally include in response to determining that the fault event is not replicated, forming one or more alternate native translations of the target ISA instructions. The one or more alternate native translations may be executable to achieve substantially equivalent functionality as obtainable via execution of the target ISA instructions. In some cases, the one or more alternative native translations are optimized differently than the first native translation so as to avoid occurrence of the fault event that was encountered during execution of the first native translation. In some cases, the one or more alternative native translations may include fewer optimizations than employed in the first native translation

[0054] At **414**, the method **400** may optionally include executing the one or more alternate native translations upon subsequently encountering the target ISA instructions.

[0055] While the depicted method may be performed in connection with any suitable hardware configuration, it will be appreciated that modifications, additions, omissions, and refinements may be made to these steps in accordance with method descriptions included above and described with references to the mechanisms, hardware, and systems shown in FIG. **1-3**.

[0056] This written description uses examples to disclose the invention, including the best mode, and also to enable a person of ordinary skill in the relevant art to practice the invention, including making and using any devices or systems and performing any incorporated methods. The patentable scope of the invention is defined by the claims, and may include other examples as understood by those of ordinary skill in the art. Such other examples are intended to be within the scope of the claims.

1. A method for identifying and replacing code translations that generate spurious fault events, comprising:
- detecting, while executing a first native translation of target instruction set architecture (ISA) instructions, occurrence of a fault event, the first native translation being executable to achieve substantially equivalent functionality as obtainable via execution of the target ISA instructions;
- decoding the target ISA instructions into functionally equivalent native instructions with a hardware decoder in response to detecting occurrence of the fault event while executing the first native translation;
- executing the target ISA instructions or a functionally equivalent version thereof, where such execution is performed without executing the first native translation;
- determining whether occurrence of the fault event is replicated while executing the target ISA instructions or the functionally equivalent version thereof; and
- in response to determining that the fault event is not replicated, determining whether to allow future execution of the first native translation or to prevent such future execution in favor of forming and executing one or more alternate native translations.

2. The method of claim **1**, where determining whether to allow or prevent future execution of the first native translation includes forming and executing the one or more alternate translations upon determining that a performance cost associated with forming the one or more alternate translations is less than a performance cost associated with continuing to execute the first native translation.

3. The method of claim **1**, further comprising incrementing a counter associated with the first native translation, and where determining whether to allow or prevent future execution of the first native translation includes preventing such execution and forming and executing the one or more alternate translations in response to saturating the counter.

4. The method of claim **3**, where the counter is incremented in response to determining that a fault event occurring during execution of the first native translation is not replicated when executing target ISA instructions or a functionally equivalent version thereof.

5. The method of claim **1**, further comprising forming one or more alternate native translations to be executed instead of the first native translation, where the one or more alternative native translations are optimized differently than the first native translation so as to avoid occurrence of the fault event that was encountered during execution of the first native translation.

6. The method of claim **5**, where the one or more alternative native translations include fewer optimizations than employed in the first native translation.

7. The method of claim **1**, further comprising:

pausing execution of the target ISA instructions or the functionally equivalent version thereof responsive to encountering a target of a branch instruction;

determining whether an instruction pointer is within an instruction pointer boundary corresponding to the first native translation when execution is paused; and

if the instruction pointer is beyond the instruction pointer boundary when execution is paused, determining that occurrence of the fault event is not replicated while executing the target ISA instructions or the functionally equivalent version thereof.

8. The method of claim **7**, where the target of the branch instruction includes a field having a bit that is set responsive to detecting occurrence of the fault event while executing the first native translation, and execution is paused responsive to encountering the set bit.

9. The method of claim **8**, where the microprocessor includes a hardware redirector that is accessed by a hardware decoder to check for a native translation corresponding to a portion of target ISA instructions, and where access to the hardware redirector by the hardware decoder is disabled when executing the target ISA instructions or the functionally equivalent version thereof without executing the first native translation

10. The method of claim **1**, further comprising:

setting a counter for execution of the target ISA instructions or the functionally equivalent version thereof based on the fault event,

pausing execution of the target ISA instructions or the functionally equivalent version thereof responsive to the counter expiring;

determining whether an instruction pointer is within an instruction pointer boundary corresponding to the first native translation when execution is paused; and

if the instruction pointer is beyond the instruction pointer boundary when execution is paused, determining that occurrence of the fault event is not replicated while executing the target ISA instructions or the functionally equivalent version thereof.

11. A micro-processing and memory system comprising:

memory configured to store target ISA instructions and a first native translation executable to achieve substantially equivalent functionality as obtainable via execution of the target ISA instructions;

a microprocessor including, execution logic configured to (1) detect, during execution of the first native translation, occurrence of a fault event, (2) roll back execution of the first native translation in response to detecting occurrence of the fault event while executing the first native translation;

a hardware decoder configured to decode the target ISA instructions into functionally equivalent native instructions in response to detecting occurrence of the fault event while executing the first native translation, where the execution logic is configured to execute the target ISA instructions or a functionally equivalent version thereof, where such execution is performed without executing the first native translation; and

a translation manager configured to (1) determine whether occurrence of the fault event is replicated while executing the target ISA instructions or the functionally equivalent version thereof, and (2) in response to determining that the fault event is not replicated, determine whether to allow future execution of the first native translation or to prevent such future execution in favor of forming and executing one or more alternate native translations.

12. The system of claim **11**, where determining whether to allow or prevent future execution of the first native translation includes forming and executing the one or more alternate translations upon determining that a performance cost associated with forming the one or more alternate translations is less than a performance cost associated with continuing to execute the first native translation.

13. The system of claim **11**, where the execution logic is configured to increment a counter associated with the first native translation, and where determining whether to allow or prevent future execution of the first native translation includes preventing such execution and forming and executing the one or more alternate translations in response to saturating the counter.

14. The system of claim **12**, where the translation manager is configured to form one or more alternate native translations to be executed instead of the first native translation, where the one or more alternative native translations are optimized differently than the first native translation so as to avoid occurrence of the fault event that was encountered during execution of the first native translation.

15. The system of claim **11**, where the execution logic is configured to pause execution of the target ISA instructions or the functionally equivalent version thereof responsive to encountering a target of a branch instruction, and where the translation manager is configured to (1) determine whether an instruction pointer is within an instruction pointer boundary corresponding to the first native translation when execution is paused, and (2) if the instruction pointer is beyond the instruction pointer boundary when execution is paused, determine that occurrence of the fault event is not replicated while executing the target ISA instructions or the functionally equivalent version thereof.

16. The system of claim **15**, where the target of the branch instruction includes a field having a bit that is set responsive to detecting occurrence of the fault event while executing the first native translation, and execution is paused responsive to encountering the set bit

17. The system of claim **11**, where the translation manager is configured to set a counter for execution of the target ISA instructions or the functionally equivalent version thereof based on detection of the fault event, where the execution logic is configured to pause execution of the target ISA instructions or the functionally equivalent version thereof responsive to the counter expiring, where the translation manager is configured to determine whether an instruction pointer is within an instruction pointer boundary corresponding to the first native translation when execution is paused, and if the instruction pointer is beyond the instruction pointer boundary when execution is paused, determine that occurrence of the fault event is not replicated while executing the target ISA instructions or the functionally equivalent version thereof.

18. A method for identifying and replacing code translations that generate spurious fault events, comprising:

detecting, while executing a first native translation of target instruction set architecture (ISA) instructions, occurrence of a fault event, the first native translation being executable to achieve substantially equivalent functionality as obtainable via execution of the target ISA instructions;

rolling back execution of the first native translation in response to detecting the fault event;

decoding the target ISA instructions into functionally equivalent native instructions with a hardware decoder in response to detecting occurrence of the fault event while executing the first native translation, where targets of branch instructions decoded by the hardware decoder include a field having a bit that is set responsive to encountering the fault event;

executing the native instructions dispatched by the hardware decoder;

pausing execution of the native instructions responsive to encountering a set bit in the field of a target of a branch instruction;

determining whether an instruction pointer is within an instruction pointer boundary corresponding to the first native translation when execution is paused;

if the instruction pointer is beyond the instruction pointer boundary when execution is paused, determining that occurrence of the fault event is not replicated while executing the target ISA instructions or the functionally equivalent version thereof; and

in response to determining that the fault event is not replicated, forming and executing one or more alternate translations upon determining that a performance cost associated with forming the one or more alternate translations is less than a performance cost associated with continuing to execute the first native translation.

19. The method of claim **18**, where the one or more alternative native translations are optimized differently than the first native translation so as to avoid occurrence of the fault event that was encountered during execution of the first native translation.

20. The method of claim **19**, where the one or more alternative native translations include fewer optimizations than employed in the first native translation.

* * * * *