US 20100205408A1

(19) **United States**

(12) **Patent Application Publication**

Chung et al.

(10) **Pub. No.: US 2010/0205408 A1**

(43) **Pub. Date: Aug. 12, 2010**

(54) **SPECULATIVE REGION: HARDWARE SUPPORT FOR SELECTIVE TRANSACTIONAL MEMORY ACCESS ANNOTATION USING INSTRUCTION PREFIX**

(75) Inventors: **Jaewoong Chung**, Bellevue, WA (US); **David S. Christie**, Austin, TX (US); **Michael P. Hohmuth**, Dresden (DE); **Stephan Diestelhorst**, Dresden (DE); **Martin Pohlack**, Dresden (DE)

Correspondence Address:
MEYERTONS, HOOD, KIVLIN, KOWERT & GOETZEL (AMD)
P.O. BOX 398
AUSTIN, TX 78767-0398 (US)

(73) Assignee: **ADVANCED MICRO DEVICES**, Sunnyvale, CA (US)

(21) Appl. No.: **12/764,024**

(22) Filed: **Apr. 20, 2010**

Related U.S. Application Data

(63) Continuation-in-part of application No. 12/510,884, filed on Jul. 28, 2009.

(60) Provisional application No. 61/084,008, filed on Jul. 28, 2008.

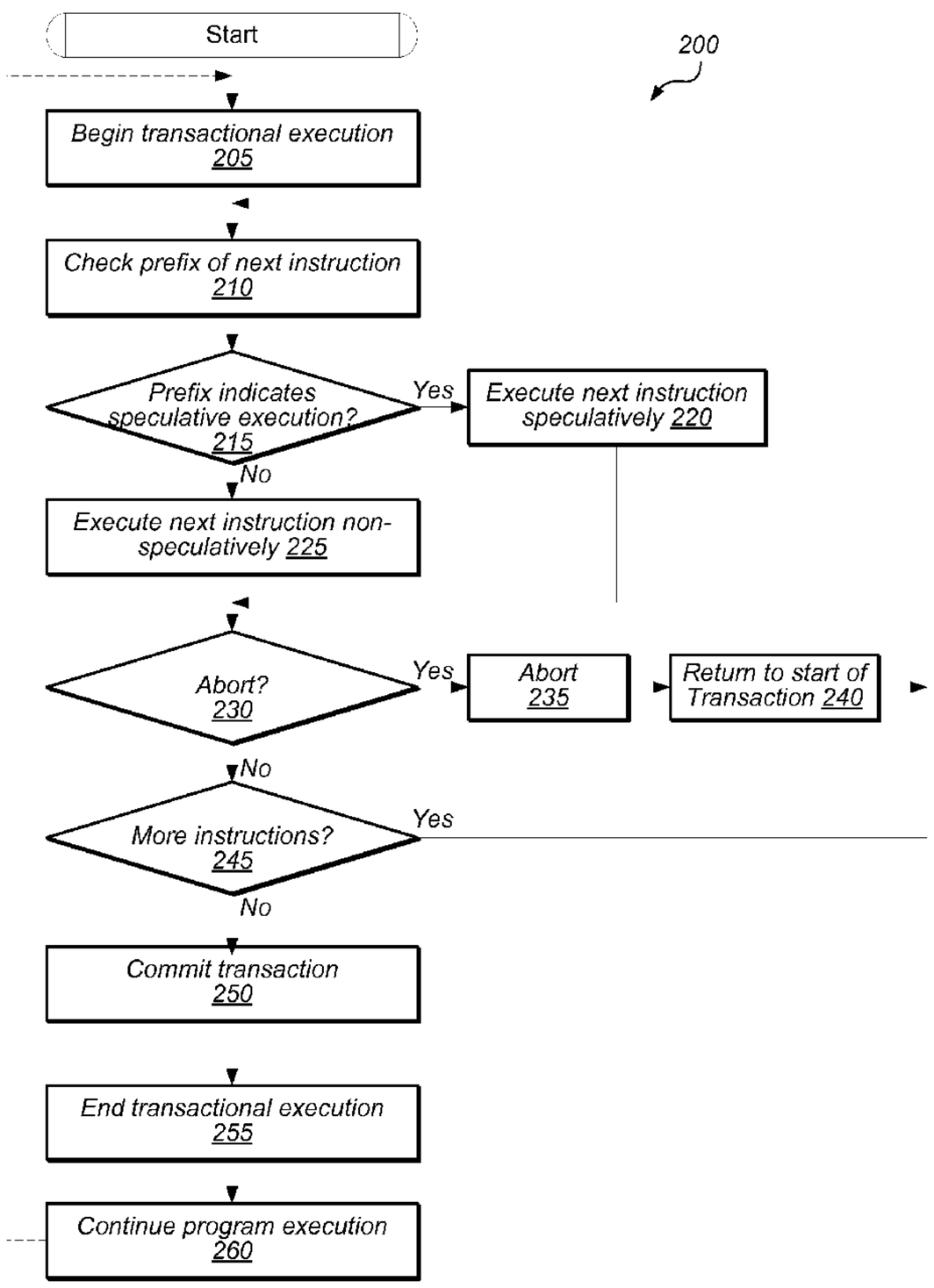
Publication Classification

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

(52) **U.S. Cl.** **712/216**; 712/E09.028

(57) **ABSTRACT**

A computer system and method is disclosed for executing selectively annotated transactional regions. The system is configured to determine whether an instruction within a plurality of instructions in a transactional region includes a given prefix. The prefix indicates that one or more memory operations performed by the processor to complete the instruction are to be executed as part of an atomic transaction. The atomic transaction can include one or more other memory operations performed by the processor to complete one or more others of the plurality of instructions in the transactional region.



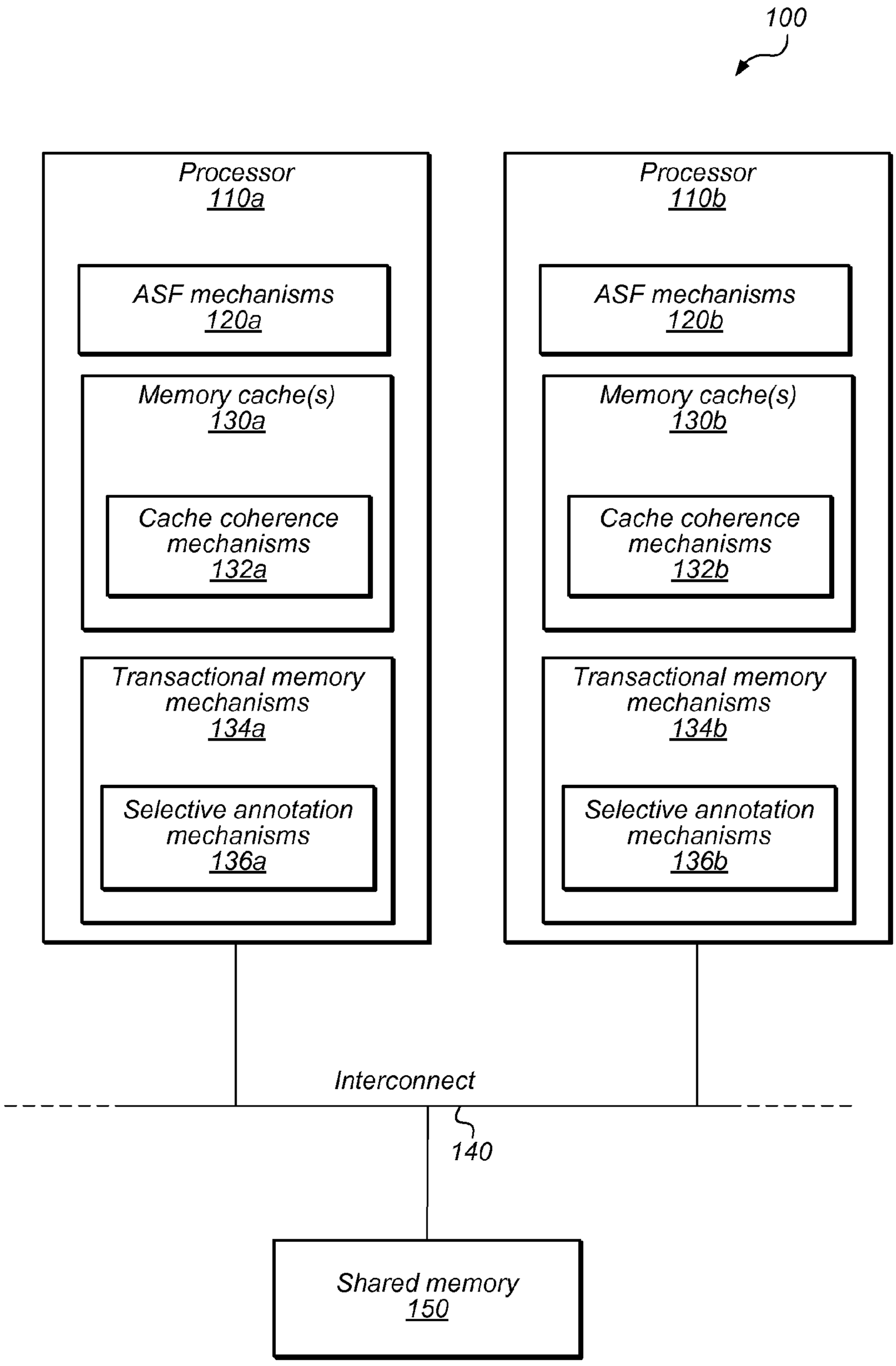

100
Processor
110a
ASF mechanisms
120a
Memory cache(s)
130a
Cache coherence mechanisms
132a
Transactional memory mechanisms
134a
Selective annotation mechanisms
136a
Processor
110b
ASF mechanisms
120b
Memory cache(s)
130b
Cache coherence mechanisms
132b
Transactional memory mechanisms
134b
Selective annotation mechanisms
136b
Interconnect
140
Shared memory
150

FIG. 1

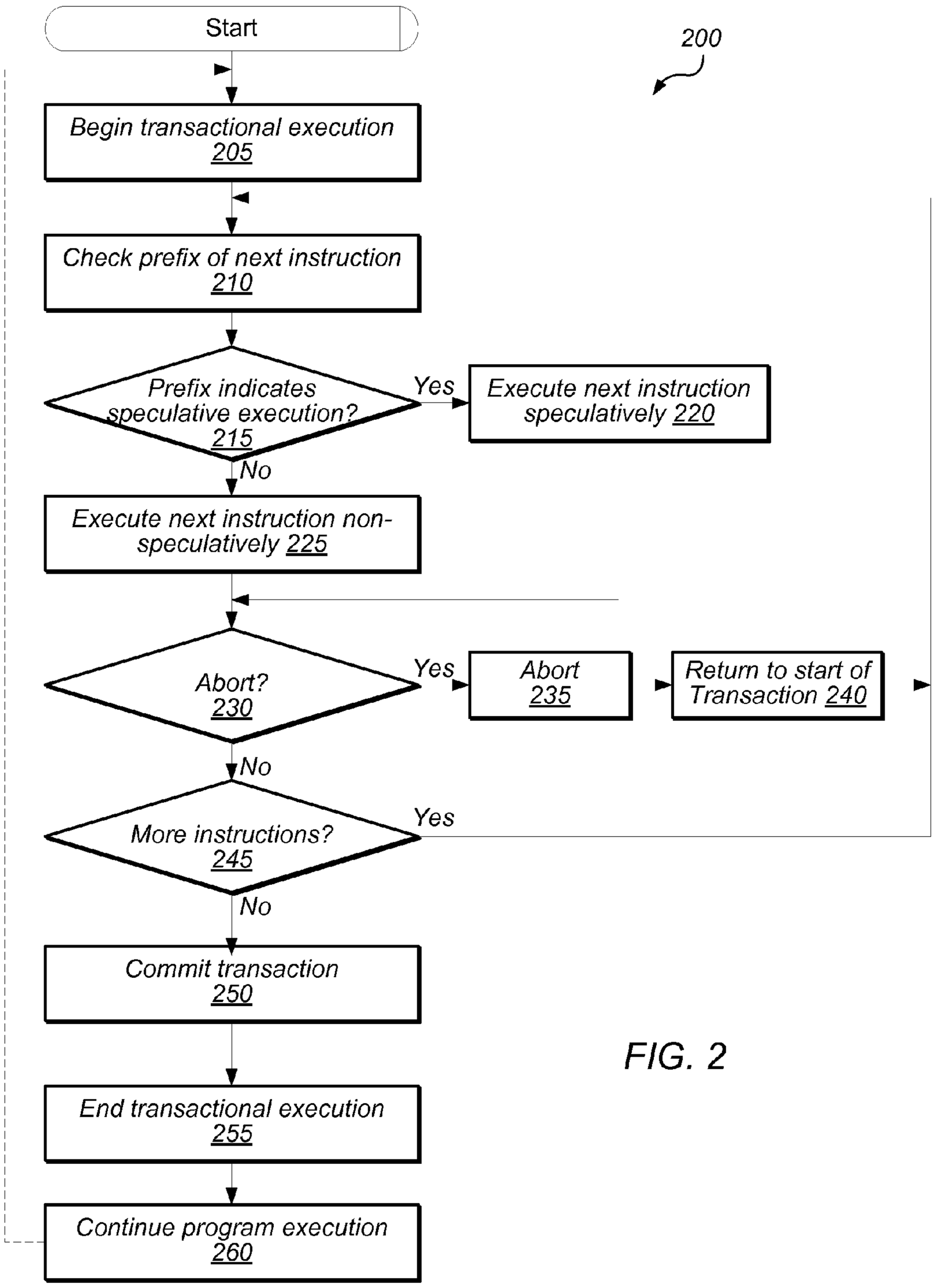

200
Start
Begin transactional execution 205
Check prefix of next instruction 210
Prefix indicates speculative execution? 215
Yes
Execute next instruction speculatively 220
No
Execute next instruction non-speculatively 225
Abort? 230
Yes
Abort 235
Return to start of Transaction 240
No
More instructions? 245
Yes
No
Commit transaction 250
End transactional execution 255
Continue program execution 260

FIG. 2

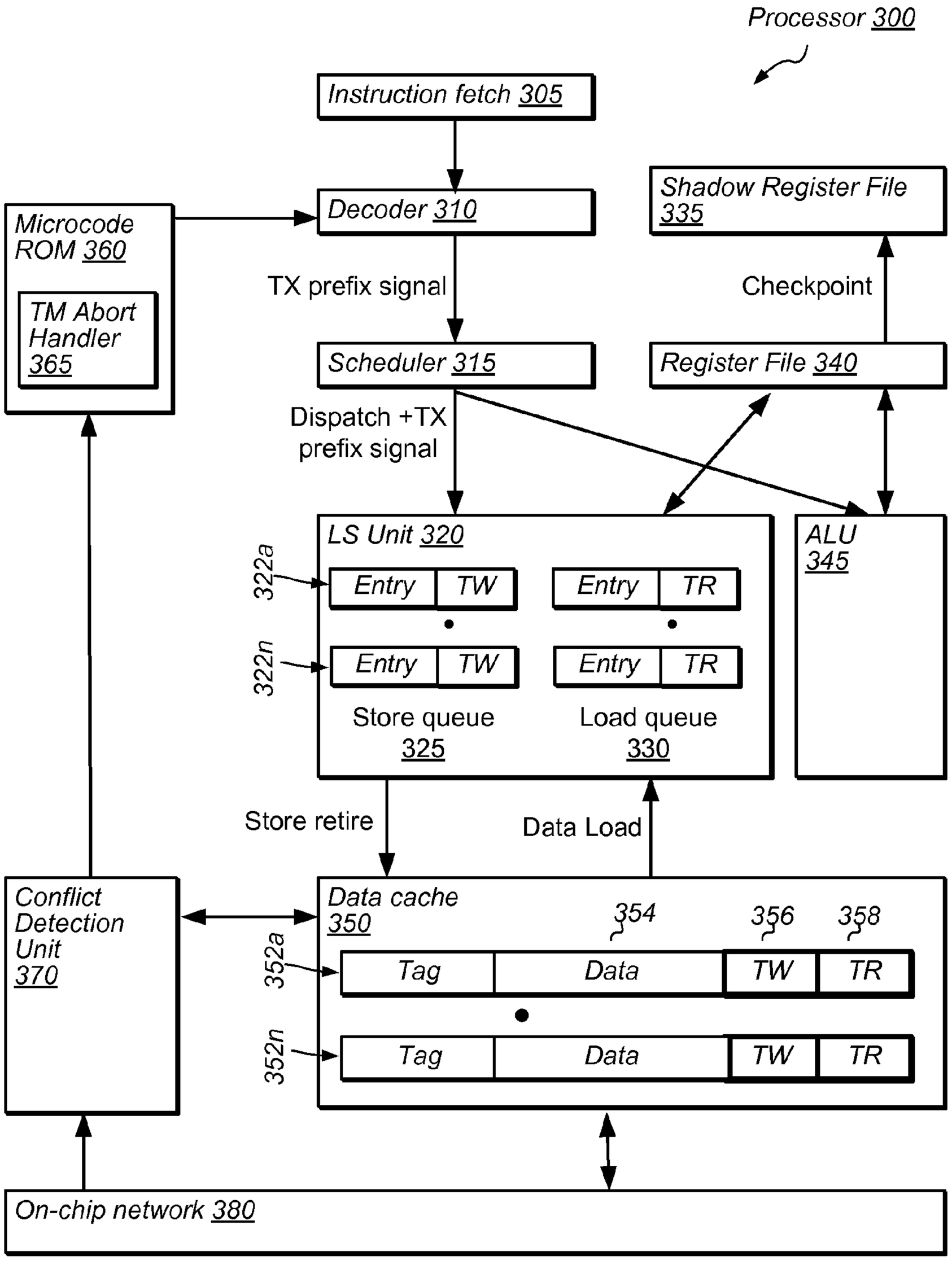

Processor 300
Instruction fetch 305
Decoder 310
Microcode ROM 360
TM Abort Handler 365
TX prefix signal
Scheduler 315
Dispatch +TX prefix signal
Shadow Register File 335
Checkpoint
Register File 340
LS Unit 320
322a
Entry
TW
Entry
TR
322n
Entry
TW
Entry
TR
Store queue 325
Load queue 330
ALU 345
Store retire
Data Load
Conflict Detection Unit 370
Data cache 350
354
356
358
352a
Tag
Data
TW
TR
352n
Tag
Data
TW
TR
On-chip network 380

FIG. 3

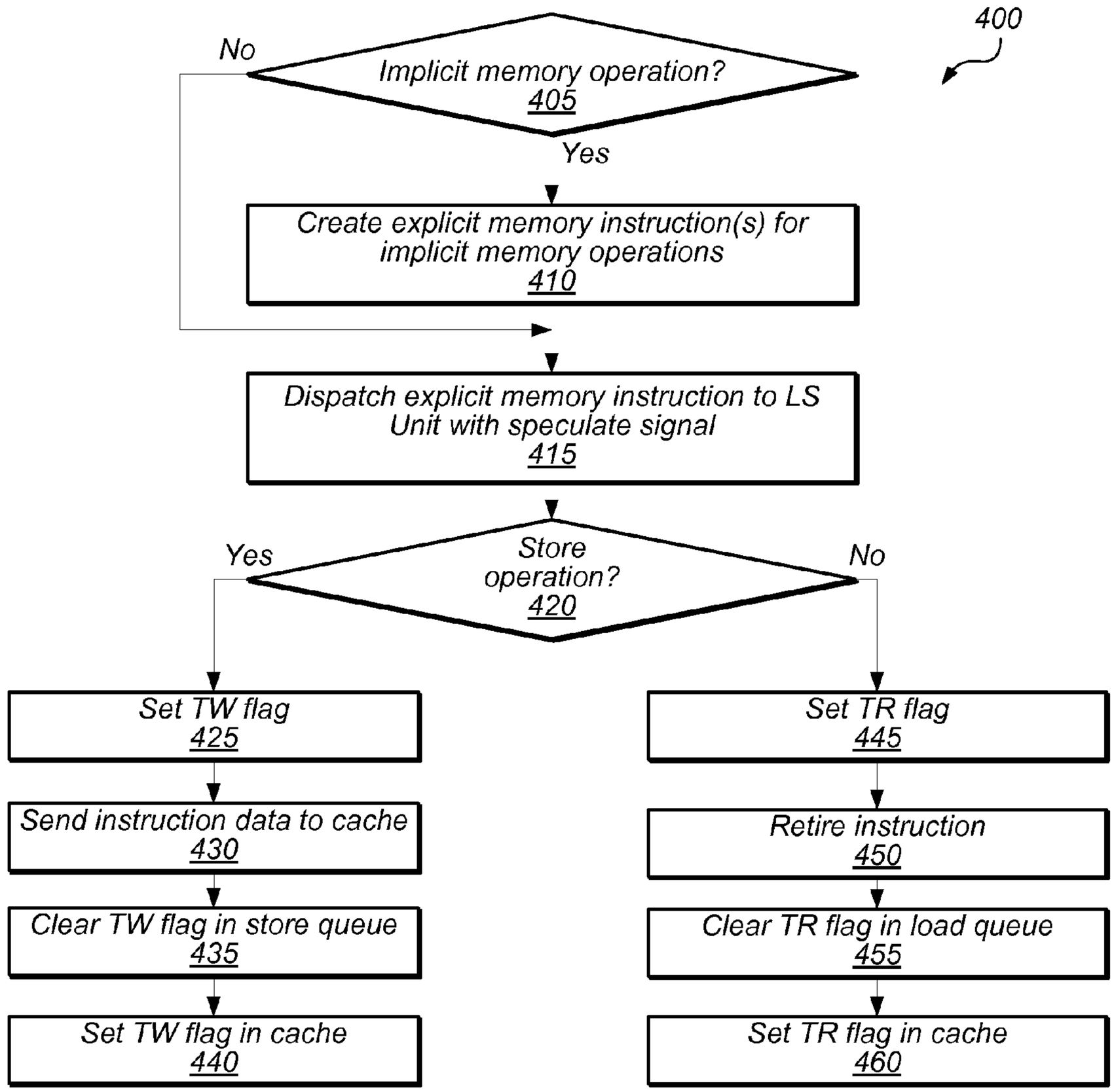

400
Implicit memory operation?
405
No
Yes
Create explicit memory instruction(s) for implicit memory operations
410
Dispatch explicit memory instruction to LS Unit with speculate signal
415
Store operation?
420
Yes
No
Set TW flag
425
Send instruction data to cache
430
Clear TW flag in store queue
435
Set TW flag in cache
440
Set TR flag
445
Retire instruction
450
Clear TR flag in load queue
455
Set TR flag in cache
460

FIG. 4

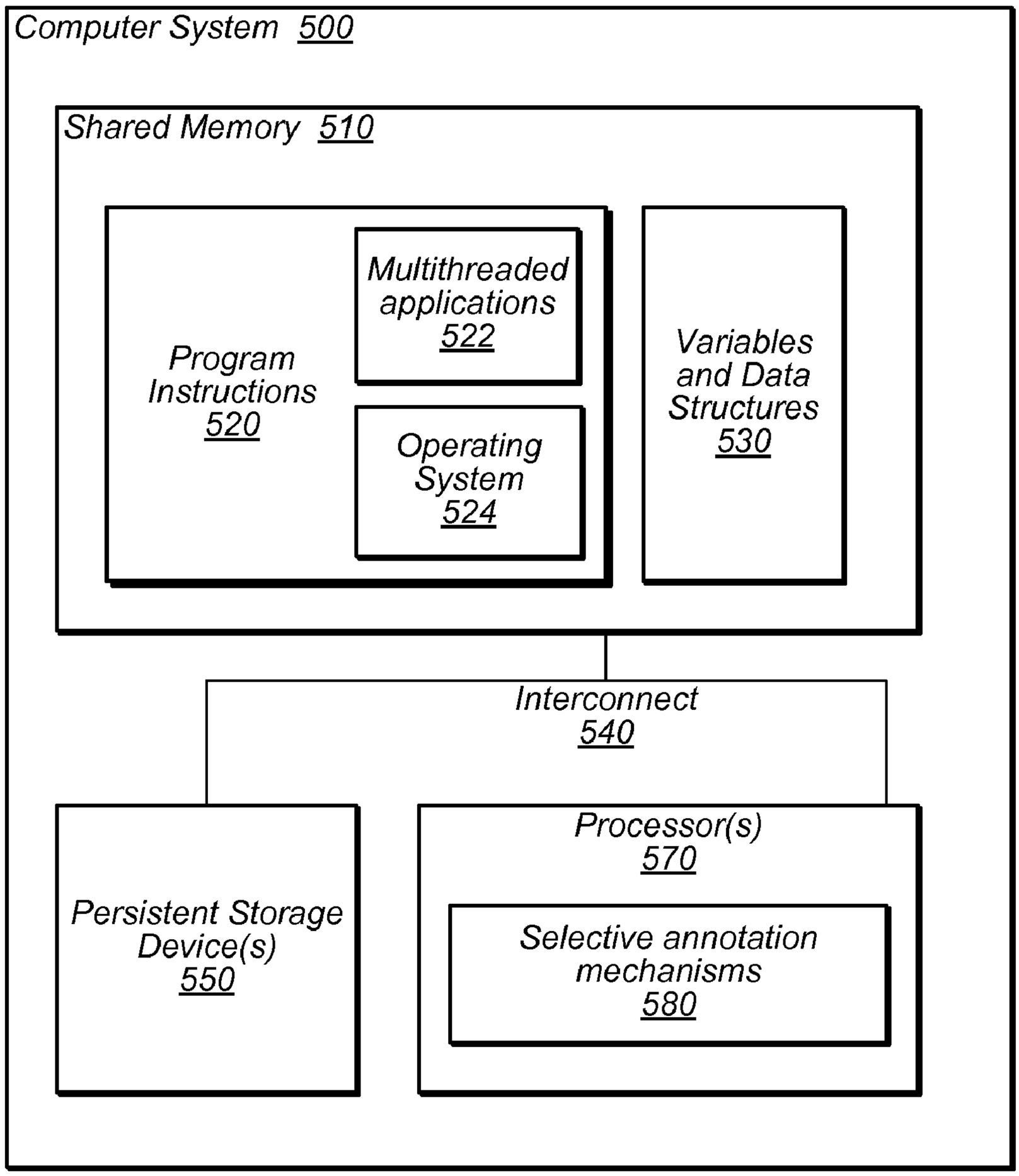

Computer System 500
Shared Memory 510
Program Instructions 520
Multithreaded applications 522
Operating System 524
Variables and Data Structures 530
Interconnect 540
Persistent Storage Device(s) 550
Processor(s) 570
Selective annotation mechanisms 580

FIG. 5

SPECULATIVE REGION: HARDWARE SUPPORT FOR SELECTIVE TRANSACTIONAL MEMORY ACCESS ANNOTATION USING INSTRUCTION PREFIX

[0001] This application is a continuation-in-part of U.S. application Ser. No. 12/510,884, filed Jul. 28, 2009, which claims the benefit of priority to U.S. Provisional Application No. 61/084,008, filed Jul. 28, 2008, both of which are incorporated by reference herein in their entireties.

BACKGROUND

[0002] Shared-memory computer systems allow multiple concurrent threads of execution to access shared memory locations. Unfortunately, writing correct multi-threaded programs is difficult due to the complexities of coordinating concurrent memory access. One approach to concurrency control between multiple threads of execution is transactional memory. In a transactional memory programming model, a programmer may designate a section of code (i.e., an execution path or a set of program instructions) as a "transaction", which a transactional memory system should execute atomically with respect to other threads of execution. For example, if the transaction includes two memory store operations, then the transactional memory system ensures that all other threads may only observe either the cumulative effects of both memory operations or of neither, but not the effects of only one.

[0003] To implement transactional memory, memory accesses are sometimes executed one by one speculatively and committed all at once at the end of the transaction. Otherwise, if an abort condition is detected (e.g., data conflict with another processor), those memory operations that have been executed speculatively may be rolled back or dropped and the transaction may be reattempted. Data from speculative memory accesses may be saved in a speculative data buffer, which may be implemented by various hardware structures, such as an on-chip data cache.

[0004] Various transactional memory systems have been proposed in the past, including those implemented by software, by hardware, or by a combination thereof. However, many traditional implementations are bound by various limitations. For example, hardware-based transactional memory proposals (HTMs) sometimes impose limitations on the size of transactions supported (i.e., maximum number of speculative memory operations that can be executed before the transaction is committed). Often, this may be a product of limited hardware resources, such as the size of one or more speculative data buffers used to buffer speculative data during transactional execution.

SUMMARY

[0005] In various embodiments, a computer processor may be configured to implement a hardware transactional memory system. The system may execute a transactional region of code such that only a subset of the instructions in the transactional region of code (e.g., those including a given instruction prefix) are executed as a single atomic memory transaction while the other instructions in the transactional region (e.g., those lacking the given prefix) are not necessarily executed atomically.

[0006] In some embodiments, a computer system may be configured to determine whether an instruction within a plurality of instructions in a transactional region includes a given prefix. The prefix indicates that one or more memory operations performed by the processor to complete the instruction are to be executed as part of an atomic transaction. The atomic transaction may include memory operations performed by the processor to complete one or more others of the plurality of instructions in the transactional region.

[0007] In some embodiments, the one or more memory operations performed atomically by the processor to complete the instruction may correspond to implicit memory operations. That is, if executing the instruction with the given prefix requires executing multiple implicit memory operations, then the processor may execute these multiple implicit memory operations atomically as party of the atomic transaction.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] FIG. **1** is a block diagram illustrating components of a multi-processor computer system configured to implement selective annotation of transactions, according to various embodiments.

[0009] FIG. **2** is a flow diagram of a method for executing a transaction that uses selective annotation, according to some embodiments.

[0010] FIG. **3** is a block diagram illustrating the hardware structures of a processor configured to implement selective transactional annotation as described herein, according to some embodiments.

[0011] FIG. **4** is a flow diagram illustrating a method by which processor **300** may execute a speculative memory access operation (as in **220**), according to some embodiments.

[0012] FIG. **5** illustrates a computing system configured to implement selective annotation as described herein, according to various embodiments.

[0013] Any headings used herein are for organizational purposes only and are not meant to limit the scope of the description or the claims. As used herein, the word "may" is used in a permissive sense (i.e., meaning having the potential to) rather than the mandatory sense (i.e. meaning must). Similarly, the words "include", "including", and "includes" mean including, but not limited to.

DETAILED DESCRIPTION OF EMBODIMENTS

[0014] In transactional programming models, a programmer may designate a region of code as a transaction, such as by using designated start and end instructions to demarcate the execution boundaries of the transaction. In many implementations, hardware capacity limitations, such as speculative buffer sizes, constrain the number of speculative memory access operations that can be executed together atomically by a hardware transactional memory (HTM) system as part of a single transaction.

[0015] In traditional implementations, all memory accesses that occur within a designated transaction are executed together atomically. However, in some cases, correct program semantics may not strictly require that all memory operations within a given transaction be executed together atomically.

[0016] According to various embodiments, a computer processor may be configured to determine a first non-empty subset of instructions within a transactional region that are

speculative and another non-empty subset of instructions within the transactional region that are non-speculative. The processor may then execute the transactional region such that the speculative subset is executed as a single atomic transaction and the non-speculative subset of instructions is not necessarily executed atomically.

[0017] In some embodiments, a processor may be configured to differentiate between speculative instructions from non-speculative instructions, at least in part based on a predefined speculative-instruction prefix. For example, some instruction set architectures (e.g., x86 and other CISC architectures) include various potentially variable-length instructions that may include optional prefix fields. For instance, x86 instructions include one to five optional prefix bytes, followed by an operation code (opcode) field, and optional addressing mode byte, scale-index-base byte, displacement field, and immediate data field. According to various embodiments, a processor may be configured to determine that a given instruction of a transactional region is speculative dependent at least on the value of the instruction prefix. As used herein, a "prefix" refers to a portion of an instruction that is distinct from the opcode field (as well as from any operands of the instruction), where the opcode field specifies an operation to be performed by a processor. A prefix further specifies the manner in which the processor performs the operation specified by the opcode.

[0018] In various embodiments, the processor may comprise different transactional memory mechanisms for executing the speculative instructions of the transactional region as a single atomic instruction while executing the non-speculative instructions without guarantee of atomicity. This differentiation between speculative and non-speculative execution for different subsets of instructions in a transaction may be referred to herein as selective annotation, and therefore, a processor in various embodiments may be configured to implement selectively annotated transactions.

[0019] For each of the speculative instructions, executing the instruction may include performing one or more explicit or implicit memory access operations. As used herein, an "explicit" memory access operation is one that the processor performs as part of executing an explicit load or store instruction (e.g., an instruction for loading from or writing to a given target memory location specified by an operand, such as the x86 MOV instruction). As used herein, an "implicit" memory access operation is one that is performed by the processor as part of executing a type of instruction other than a load/store instruction, but one which necessitates that the processor perform a memory access to one or more memory locations specified by one or more operands. For example, an "ADD" instruction having an operand that specifies a memory location rather than a register or immediate value is an example of instruction whose execution includes an implicit memory access operation.

[0020] FIG. **1** is a block diagram illustrating components of a multi-processor computer system configured to implement selective annotation of transactions, according to various embodiments. According to the illustrated embodiment, computer system **100** may include multiple processors, such as processors **110***a* and **110***b*. In different embodiments, the processors (e.g., **110**) may be coupled to each other and/or to a shared memory (e.g., **150**) over an interconnect, such as **140**. In various embodiments, different interconnects may be used, such as a shared system bus or a point-to-point network in various topographies (e.g., fully connected, torus, etc.).

[0021] In some embodiments, processors **110** may comprise multiple physical or logical (e.g., SMT) cores, each capable of executing a respective thread of execution concurrently. In some such embodiments, selective annotation may be implemented by a single processor with multiple cores. However, for clarity of explanation, the embodiments outlined herein are described using single-core processors. Those skilled in the art will recognize that the methods and systems described herein apply also to multi-core processors.

[0022] According to the illustrated embodiment, each processor **110** may include one or more levels of memory caches **130**. Levels of memory caches may be hierarchically arranged (e.g., L1 cache, L2 cache, L3 cache, etc.) and may be used to cache local copies of values stored in shared memory **150**.

[0023] In various embodiments, memory caches **130** may include various cache-coherence mechanisms **132**. Cache-coherence mechanisms **132** may, in some embodiments, implement a cache coherence communication protocol among the interconnected processors, which may ensure that the values contained in memory caches **130** of each processor **110** are coherent with values stored in shared memory and/or in the memory caches of other processors. Several such protocols exist (including the MESI (i.e., Illinois protocol) and MOESI protocols), and may be implemented in various embodiments. Cache coherence protocols may define a set of messages and rules by which processors may inform one another of modifications to shared data and thereby maintain cache coherence. For example, according to the MESI protocol, each block stored in a cache must be marked as being in one of four states: modified, exclusive, shared, or invalid. A given protocol defines a set of messages and rules for sending and interpreting those messages, by which processors maintain the proper markings on each block. Depending on the state of a given cache block, a processor may be restricted from performing certain operations. For example, a processor may not execute program instructions that depend on a cache block that is marked as invalid. Cache coherence mechanisms may be implemented in hardware, software, or in a combination thereof, in different embodiments. Cache coherence messages may be may be communicated across interconnect **140** and may be broadcast or point-to-point.

[0024] According to the illustrated embodiment, each processor **110** may also include various transactional memory mechanisms (e.g., **134**) for implementing transactional memory, as described herein. Transactional memory mechanisms **134** may include selective annotation mechanisms **136** for implementing selective annotation using instruction prefixes, as described herein. In various embodiments, more processors **110** may be connected to interconnect **140**, and various levels of cache memories may be shared among multiple ones of such processors and/or among multiple cores on each processor.

[0025] FIG. **2** is a flow diagram of a method for executing a transaction that uses selective annotation, according to some embodiments. Method **200** may be performed by a processor (e.g., **110**) in a multiprocessor system. In one embodiment, method **200** may be performed by processor **110** executing a thread, and is described in this manner below.

[0026] According to the illustrated embodiment, method **200** begins when the thread of execution encounters a transactional region of code, and thus begins transactional execution, as in **205**. In some embodiments, the processor may begin transactional execution (i.e., enter a transactional mode

of execution) in response to executing an explicit instruction that indicates the beginning of a transactional region of code (e.g., TxBegin, SPECULATE, etc.).

[0027] In some embodiments, once the transactional execution begins, the processor examines each instruction encountered with the transactional region of code to determine whether the instruction is speculative. In one embodiment, the processor may be configured to ensure that the set of instructions in the transaction that are determined to be speculative are executed together as a single atomic memory transaction. However, the processor need not make such a guarantee for any instructions in the transactional region of code that are not determined to be speculative.

[0028] According to various embodiments, the processor may determine which of the instructions are speculative based, at least in part, on a prefix (a "speculative instruction prefix") that indicates that a particular instruction is to be executed speculatively (e.g., within a single atomic memory transaction). For example, various Complex Instruction Set Computers (CISC), such as x86, allow each instruction to include (or exclude) a prefix field, which can be used to provide the processor with special directions for executing an instruction specified by the opcode portion of the instruction encoding. In some embodiments, an instruction may include an instruction prefix informing the processor that the instruction is speculative and should be executed atomically as part of the speculative instruction subset in the transaction body. For example, the speculative-instruction prefix may be a one-byte encoding using reserved register D**6** or F**1**. In other embodiments, the prefix may be implemented as a two-bye encoding where the first byte is **0**F (escaping byte) and the second byte is one of the unused encodings available. Various other encodings may be possible.

[0029] According to the illustrated embodiment, for each instruction in the transaction, the processor checks for a speculative-instruction prefix, as in **210**. For example, in one embodiment, the instruction decoding unit of the processor (as shown in FIG. **3**) is configured to determine whether a given instruction includes a speculative instruction prefix.

[0030] If the examined instruction includes the speculative instruction prefix, as indicated by the affirmative exit from **215**, the processor executes the next instruction speculatively, as in **220**. Alternatively, if the next instruction does not include the speculative-instruction prefix, as indicated by the negative exit from **215**, the processor executes the instruction non-speculatively, as in **225**. That is, instructions without the speculative instruction prefix may be executed and committed upon being retired, regardless of whether the transaction has been committed. Executing such non-speculative instructions may consume fewer or no transactional memory hardware resources, such as transactional buffer space.

[0031] As discussed above, different instructions may include one or more explicit or implicit memory access operations. For example, an explicit memory store (e.g., MOV) instruction explicitly instructs the processor to perform a memory access to a given memory location specified in an operand. However, other instructions (e.g., ADD) may include one or more operands that specify memory locations that contain data needed by the processor to execute the full instruction. In such cases, the processor must perform one or more memory read operations as part of executing the instruction, meaning that the instruction contains one or more implicit memory references. For example, if an ADD instruction includes three register operands (two that contain respective locations in memory where the values to be added are stored and a third register operand for storing the summation) then executing the ADD instruction may require the processor to read the respective memory locations before performing the summation. In this case, the ADD instruction implicitly requires two memory read operations.

[0032] In some embodiments, executing a set of speculative instructions as a single atomic transaction includes performing both the explicit and implicit memory operations necessitated by the speculative instructions as a single atomic transaction. For instance, if the ADD instruction from the example above is determined to be speculative, then the two implicit memory operations involved in reading the operands are included in the set of memory operations that are executed by the processor as a single atomic transaction.

[0033] As shown in FIG. **2**, the processor continues executing instructions of the transaction (as indicated by the feedback loop from **245** to **210**) until either an abort condition is detected (affirmative exit from **230**) or no more instructions exist in the transaction (negative exit from **245**). If all instructions have been executed (as indicated by the negative exit from **245**), then the transaction is committed (as in **250**), and the processor exits the transactional execution mode (as in **255**) and continues normal execution (as in **260**). Normal execution may include executing more transactions, as indicated by the feedback loop from **260** to **205**.

[0034] Otherwise, if an abort condition is detected before the transaction commits (as indicated by the affirmative exit from **230**), then the processor aborts the transaction attempt (as in **235**), rolls back execution to the start of the transaction (as in **240**), and reattempts to execute the transaction (as indicated by the feedback loop from **240** to **210**). In some embodiments, aborting the transaction attempt may include dropping speculative data and/or metadata and/or undoing various speculatively executed memory operations.

[0035] In various embodiments, aborts may be caused by different conditions. For example, if there is insufficient hardware capacity to buffer speculative data in the transaction (e.g., the transaction is too long), then the processor may determine that the transaction attempt must be aborted (as in **235**). Buffering speculative data is discussed in more detail below. In another example, an abort may be caused by memory contention—that is, interference caused by another processor attempting to access one or more memory locations accessed by the processor as part of executing one of the speculative instructions. In various embodiments, the processor may include contention detection mechanisms configured to detect various cache coherence messages (e.g., invalidating and/or non-invalidating probes) sent by other processors. The contention detection mechanisms may determine whether a received probe is relevant to one or more memory areas accessed as part of executing a speculative instruction and to determine whether the probe indicates that a data conflict exists. In response to detecting a data conflict, the processor may abort the transactional attempt, as in **235**.

[0036] According to various embodiments, a transaction may be aborted if an invalidating probe relevant to a speculatively-read memory location is received and/or if a non-invalidating probe relevant to a speculatively-written memory location is received. In one example of detecting a data conflict, consider a first thread executing in transactional mode on a first processor and having an access to a memory location as part of executing a speculative instruction. If a second thread executing on a second processor subsequently

attempts a store to the speculatively-accessed memory location, then the second processor may send an invalidating probe to the first processor in accordance with the particular cache coherence protocol deployed by the system. If the first processor receives the invalidating probe while the memory location is still protected (e.g., before the first thread commits its transaction or otherwise releases the memory location) then a data conflict may exist and the first processor may abort the transaction.

[0037] Once a transaction is committed, as in **250**, all values written either explicitly and/or implicitly by the processor as part of executing the speculative instructions in the transaction become visible to all other threads in the system atomically. However, data values read or written as part of executing the non-speculative instructions are not protected as part of the transactional execution.

[0038] In various embodiments, different mechanisms and/or techniques may be used to implement transactional memory. For example, data accessed by one or more speculative instructions may be marked as speculative in a speculative data buffer, which may be implemented by various processor structures, such as one or more data caches, a load queue, store queue, combined load/store queues, etc.

[0039] Additionally, different transactional memory mechanisms may follow different policies with respect to speculative data values resulting from speculative write operations (implicit and/or explicit). For example, in some embodiments, the processor may implement a redo protocol where speculative data values are kept in a private speculative data buffer until commit time (e.g., **250**), when they are then collectively exposed to other threads in the system. In the case of an abort, the speculative data in the speculative buffer may simply be dropped. In other embodiments, the processor may implement an undo policy, where the processor records a checkpoint at the start of the transaction and restores the checkpoint in the case of an abort, thereby overriding any data modified as part of executing one or more speculative instructions during the transaction. Various other combinations are possible. Such techniques for redoing or undoing modifications to memory in response to a transaction committing or aborting may be referred to herein broadly as "versioning" and the metadata recorded to enable versioning may be referred to as "versioning data."

[0040] In various embodiments, the ISA may support nested transactions so that another transaction can begin within a currently executing transaction. In different embodiments, the HTM implementation may support such nesting in different ways, such as by subsuming the outer transactions (i.e., flattening) or by treating an inner transaction as separate independent transaction from a transaction that contains it.

[0041] FIG. **3** is a block diagram illustrating the hardware structures of a processor configured to implement selective transactional annotation as described herein, according to some embodiments. In the illustrated embodiment, processor **300** includes an instruction fetch unit **305** configured to fetch program instructions to execute, a decoder **310** configured to decode/interpret the fetched instructions, and a scheduler **315** configured to schedule the instructions for execution. In various embodiments, decoder **310** may be configured to recognize various transactional memory instructions, such as TxBegin for starting a transaction, TxEnd for ending a transaction, and the Tx prefix for determining which instructions in a transaction are speculative based on the use of selective annotation.

[0042] In the illustrated embodiment, scheduler **315** is configured to dispatch instructions to the proper execution units, such as Load Store Unit **320** for memory instructions and Arithmetic Logic Unit **345** for arithmetic instructions. Both execution units **320** and **345** are configured to communicate with a register file **340**, which contains operand data and/or various other register values.

[0043] According to the illustrated embodiment, processor **300** may include a shadow register file **335** configured to store a register file checkpoint of register file **340**. For example, as part of executing a TxBegin instruction, processor **300** may take a register checkpoint, such as by storing a backup copy of the current values held in various registers of register file **340** (e.g., program counter register). In the event of a transaction abort, the checkpoint values may be restored from the shadow register file **335** to the register file **340**. For instance, if the transaction is aborted, program control flow may be returned to the start of the transaction by restoring the value of the program counter register stored in shadow register file **335** to register file **340**. In some embodiments, the transaction initiating instruction may accept a parameter indicating an alternative address for the program counter saved in the checkpoint operation, such that in case of an abort and/or rollback operation, the program execution could be made to jump to the alternative address.

[0044] When processor **300** is executing in a transactional mode (e.g., a TxBegin instruction has been executed and no corresponding TxEnd instruction has been executed), it may perform some instructions speculatively and perform other instructions non-speculatively. The processor may be configured to determine which instructions to execute speculatively based at least in part on decoder **310** detecting a speculative instruction prefix. For example, the processor may be configured to execute those instructions that include a given Tx prefix speculatively, while executing those instructions without the given prefix non-speculatively.

[0045] In various embodiments, the processor may be configured to store versioning data in various components, such as data cache **350** and/or load-store unit **320**. For example, in the illustrated embodiment, processor **300** includes data cache **350**, which is configured to store data from recently-accessed memory regions. The data cache **350** may be arranged into multiple cache lines **352***a*-**352***n*, each identified by one or more tags and each storing data (e.g., data **254**) from recently-accessed regions of memory (e.g., shared memory **150** of FIG. **1**).

[0046] In addition to buffering data from recently-accessed regions, data cache **350** may be used to implement a speculative buffer for buffering speculative transactional data. For example, in some embodiments, each cache line **352** in data cache **350** may include versioning data, such as one or more associated transaction flags usable to indicate whether the data in the cache line has been accessed by the processor as part of executing a speculative instruction and/or the nature of such accesses. For example, in the illustrated embodiment, processor **300** includes TW flag **356**, which is usable to indicate whether data in the cache line has been transactionally written (i.e., written as part of executing a speculative instruction) and TR flag **358** usable to indicate whether data in the cache line has been transactionally read. In various embodiments, TW flag **356** and/or TR flag **358** may comprise any suitable number of bits.

[0047] In addition to buffering speculative data in data cache **350**, processor **300** may also buffer speculative data in

a load and/or store queue, such as store queue **325** and load queue **330** in load/store unit **320**. In some embodiments, load queue **330** may hold data indicative of an issued load instruction that has not yet been retired and store queue **325** may hold data indicative of an issued store instruction that has not yet been retired. For example, store queue **325** and load queue **330** may include one or more entries **322***a*-**322***n* for storing such data. In various embodiments, each such entry may comprise a TW flag or TR flag, such as flags **356** and **358**, indicating whether the data is associated with a speculative instruction.

[0048] In different embodiments, the speculative data buffer may be implemented by one or more data caches (e.g., **350**), in load and store queues (e.g., **330** and **325** respectively), a combined load/store queue, or any combination thereof. For example, in some embodiments, speculative data from retired instructions may be moved from store queue **325** or load queue **330** into data cache **350**. In some embodiments, LS unit **320** may be configured to detect whether such a transfer would overflow the capacity of data cache **350** to buffer all the speculative data of an active transaction, and in response, to delay flushing the speculative data and instead maintain it in the load or store queues.

[0049] According to the illustrated embodiment, processor **300** also includes an on-chip network **380** usable by multiple processors (and/or processing cores) to communicate with one another. In some embodiments, on-chip network **380** may be analogous to interconnect **140** of FIG. **1**, and may implement various network topologies.

[0050] Processor **300** may be configured to detect cache coherency probes sent from other processors via on-chip network **380**, such as by using conflict detection unit **370**. Conflict detection unit **370** may receive a cache coherency probe (e.g., sent as part of a cache coherency protocol, such as MESI) and in response, check the speculative buffer implemented by data cache **350** and/or LS Unit **320** to determine if a data conflict exists. For example, conflict detection unit **370** may check the tag of each cache line **352** to determine if the received probe matches the cache line tag and check the TW flag **356** and/or TR flag **358** to determine whether the data contained in the cache line is speculative. In some embodiments, based on these determinations, the conflict detection unit **370** may determine whether the probe indicates a data conflict.

[0051] In some embodiments, a data conflict occurs if two processors have accessed a location in shared memory and at least one processor has written to it. Therefore, conflict detection unit **370** may detect a conflict if a received probe matches an entry in data cache **350** or LS Unit **320** and the entry has the TW flag set. Also, conflict detection unit **370** may detect a conflict if a received probe matches an entry and the probe indicates a write operation (e.g., an invalidating probe). In one example, if the probe indicates that the sending processor has read data from a memory location that matches the tag of **352***a* in data cache **350**, and TW flag **356** indicates that processor **300** has modified that data speculatively within an active (i.e., not yet committed) transaction, then conflict detection unit **370** may determine that the probe indicates a data conflict and therefore signal an abort condition.

[0052] In response to detecting a conflict, conflict detection unit **370** may invoke the microcoded transaction abort handler **365** in microcode ROM **360**, which may invalidate the cache entries with the TW bits, clear all TW/TR bits, restore the register checkpoint taken when the transaction began, and/or flush the instruction pipeline. Since the checkpoint has been restored, including the old program counter, the execution flow then returns to the start of the transaction. Alternatively, if the transaction reaches TxEnd, it may be committed, which may include clearing all TW/TR bits and discarding the register checkpoint.

[0053] In some embodiments, processor **300** may implement transaction workflow **200** of FIG. **2**. For example, processor **300** may begin a transaction (i.e., enter transactional execution mode) as in **205** by executing a TxBegin instruction recognized by decoder **310**. Executing the TxBegin instruction may include storing a checkpoint in shadow register file **335**. This checkpoint may include the values held in various register in register file **340**, including the program counter value that can be used to roll back the transaction in case of an abort.

[0054] For each instruction in the transaction, the decoder **310** may determine if the instruction includes the speculative instruction prefix (e.g., TX), as in **210**. If the instruction includes the prefix, then decoder **310** determines that the instruction should be executed speculatively as part of the transaction, as indicated by the affirmative exit from **215**. As in method **200** of FIG. **2**, each instruction determined to be speculative is executed speculatively, as in **220**.

[0055] FIG. **4** is a flow diagram illustrating a method by which processor **300** may execute a speculative memory access operation (as in **220**), according to some embodiments.

[0056] Method **400** begins when the decoder (and/or other component(s) of the processor) determines (as in **405**) whether executing the given prefixed instruction necessitates executing one or more implicit memory operations. If so, as indicated by the affirmative exit from **405** to **410**, the processor may split the instruction into multiple simpler instructions (e.g., RISC-style instructions), which may include a respective explicit memory access instruction (e.g., MOV) for each of the implicit memory operations that executing the prefixed instruction requires. However if the prefixed instruction is already an explicit memory access instruction, then the processor may skip step **410**, as indicated by the negative exit from **405**.

[0057] In the illustrated embodiment, method **400** then includes dispatching the explicit memory instruction(s) to load/store unit **320** with a speculate signal, as in **415**. The speculate signal may indicate to LS unit **320** that the instruction is to be executed speculatively. This dispatching may be performed via an instruction scheduler, such as scheduler **315**.

[0058] According to method **400**, if the instruction is a store operation, as indicated by the affirmative exit from **420**, then it may be transferred to store queue **325** for execution. Thus, the store operation data may be stored in store queue **325** in one of entries **322**, and the TW flag of the entry may be set, as in **425**. Once the instruction is executed, the data in the respective entry **322** may be sent to data cache **350** for buffering, as in **430**, the TW flag in the store queue may be cleared as in **435**, and the TW flag of the new entry in the data cache may be set, as in **440**.

[0059] According to method **400**, if the explicit memory access operation is not a store operation, as indicated by the negative exit from **420**, then the instruction may be a load operation and may be transferred to load queue **330** for execution. Thus, the load operation may be stored in load queue **330** in one of entries **322**, and the TR flag of the entry may be set,

as in **445**. Once the instruction is executed, the instruction is retired (as in **450**), the TR flag of the respective load queue entry is cleared (as in **455**), and the TR flag of the data cache entry for the loaded data is set (as in **460**).

[0060] For instructions that do no have the speculative instruction prefix (e.g., negative exit from **215** in FIG. **2**), decoder **310** does not send a speculate signal to LS unit **320**. Thus, such instructions may be executed non-speculatively, as in **225**. When an instruction is executed non-speculatively, the processor is configured to not record versioning data, such as the TR and/or TW flags, for the instruction.

[0061] If conflict detection unit **370** detects an abort condition (as in **230**), then it may invoke abort handler **365** to perform an abort, as in **235**. In some embodiments, performing the abort may include invalidating entries of data cache **350** and/or of LS unit **320** whose TW flag is set and then clearing all TR and/or TW flags. In performing the abort, abort handler **365** may then restore the register checkpoint taken at the start of the transaction, including the old program counter value. Thus, the abort procedure may return program control to the start of the transaction, as in **240**, allowing the processor to reattempt execution.

[0062] In some instances, it may be desirable for every instruction in a transaction to be treated as speculative. For example, if an application invokes a function implemented in legacy code that does not use transactions, but correct program semantics dictate that the legacy function should be executed transactionally, then it may be desirable to indicate the system that all explicit and/or implicit memory access operations performed by the legacy function should be treated as speculative. To accommodate such use cases, in some embodiments, decoder **310** may be configured to detect whether an instruction that initiates transactional execution (e.g., FxBegin) includes the speculative instruction prefix (e.g., FX). In such embodiments, if the transaction-initiating instruction (e.g., FxBegin) includes the speculative instruction prefix, the processor is configured to treat every explicit and implicit memory access instruction within the transaction as speculative.

[0063] FIG. **5** illustrates a computing system configured to implement selective annotation as described herein, according to various embodiments. The computer system **500** may be any of various types of devices, including, but not limited to, a personal computer system, desktop computer, laptop or notebook computer, mainframe computer system, handheld computer, workstation, network computer, a consumer device, application server, storage device, a peripheral device such as a switch, modem, router, etc, or in general any type of computing device.

[0064] Computer system **500** may include one or more processors **570**, each of which may include multiple cores, any of which may be single or multi-threaded. The processor may be manufactured by configuring a semiconductor fabrication facility through the use of various mask works. These mask works may be created/generated by the use of netlists, HDL, GDS data, etc.

[0065] The computer system **500** may also include one or more persistent storage devices **550** (e.g. optical storage, magnetic storage, hard drive, tape drive, solid state memory, etc) and one or more memories **510** (e.g., one or more of cache, SRAM, DRAM, RDRAM, EDO RAM, DDR 12 RAM, SDRAM, Rambus RAM, EEPROM, etc.). Various embodiments may include fewer or additional components not illustrated in FIG. **5** (e.g., video cards, audio cards, additional network interfaces, peripheral devices, a network interface such as an ATM interface, an Ethernet interface, a Frame Relay interface, etc.)

[0066] The one or more processors **570**, the storage device (s) **550**, and the system memory **510** may be coupled to the system interconnect **540**. One or more of the system memories **510** may contain program instructions **520**. Program instructions **520** may include program instructions executable to implement one or more multithreaded applications **522** and operating systems **524**. Program instructions **520** may be encoded in platform native binary, any interpreted language such as Java™ byte-code, or in any other language such as C/C++, Java™, etc or in any combination thereof.

[0067] Any number of program instructions **520** may include a speculative instruction prefix as described herein for selective annotation of speculative regions. Each processor **570** may include a decoder unit for recognizing instructions of program instructions **520** usable to signal the start of a transactional region (e.g., TxBegin), the end of a transactional region (e.g., TxEnd), and/or a speculative-instruction prefix (e.g., TX), as described herein.

[0068] Program instructions **520**, such as those used to implement multithreaded applications **522** and/or operating system **524**, may be provided on a computer readable storage medium. The computer-readable storage medium may include any tangible (non-transitory) mechanism for storing information in a form (e.g., software, processing application) readable by a machine (e.g., a computer). The computer-readable storage medium may include, but is not limited to, magnetic storage medium (e.g., floppy diskette); optical storage medium (e.g., CD-ROM); magneto-optical storage medium; read only memory (ROM); random access memory (RAM); erasable programmable memory (e.g., EPROM and EEPROM); flash memory; electrical, or other types of medium suitable for storing program instructions.

[0069] A computer-readable storage medium as described above can be used in some embodiments to store instructions read by a program and used, directly or indirectly, to fabricate the hardware comprising system processor **570**. For example, the instructions may describe one or more data structures describing a behavioral-level or register-transfer level (RTL) description of the hardware functionality in a high level design language (HDL) such as Verilog or VHDL. The description may be read by a synthesis tool, which may synthesize the description to produce a netlist. The netlist may comprise a set of gates (e.g., defined in a synthesis library), which represent the functionality of processor **570**. The netlist may then be placed and routed to produce a data set describing geometric shapes to be applied to masks. The masks may then be used in various semiconductor fabrication steps to produce a semiconductor circuit or circuits corresponding to processor **570**. Alternatively, the database may be the netlist (with or without the synthesis library) or the data set, as desired.

[0070] The scope of the present disclosure includes any feature or combination of features disclosed herein (either explicitly or implicitly), or any generalization thereof, whether or not it mitigates any or all of the problems addressed herein. Accordingly, new claims may be formulated during prosecution of this application (or an application claiming priority thereto) to any such combination of features. In particular, with reference to the appended claims, features from dependent claims may be combined with those of the independent claims and features from respective inde-

pendent claims may be combined in any appropriate manner and not merely in the specific combinations enumerated in the appended claims.

[0071] Although the embodiments above have been described in considerable detail, numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

What is claimed:

1. An apparatus, comprising:

a computer processor configured to determine whether an instruction within a plurality of instructions in a transactional region of code includes a prefix indicating that one or more memory operations performed by the computer processor to complete the instruction are to be executed as part of an atomic transaction that includes memory operations performed by the computer processor to complete at least one other of the plurality of instructions.

2. The apparatus of claim **1**, wherein the computer processor is further configured to determine that at least some other of the plurality of instructions do not include the prefix and in response, to execute those instructions non-atomically.

3. The apparatus of claim **1**, wherein the one or more memory operations performed by the computer processor to complete the instruction are implicit memory operation of the instruction.

4. The apparatus of claim **1**, wherein execution of the one or more memory operations includes buffering versioning data for the instruction in a data cache of the processor.

5. The apparatus of claim **1**, wherein the computer processor comprises a decoder unit and at least one execution unit, wherein the decoder is configured to determine that the instruction includes the prefix and to send the instruction to the at least one execution unit with an indication that the instruction is speculative.

6. The apparatus of claim **1**, wherein the computer processor is configured to execute all of the plurality of instructions as speculative instructions in response to an opcode portion of the instruction indicating the start of the transactional region of code.

7. The apparatus of claim **1**, wherein at least one other of the plurality of instructions also includes the prefix indicating that one or more memory operations performed by the computer processor to complete the at least one other instruction are to be executed as part of the atomic transaction.

8. The apparatus of claim **1**, wherein the computer processor is configured to detect an abort condition while executing the transactional region of code, and, in response thereto, to abort execution of the transactional region of code, at least by undoing modifications to values stored in memory as a result of executing one or more speculative instructions within the plurality of instructions without undoing modifications to one or more other values stored in memory as a result of executing one or more non-speculative instructions within the plurality of instructions.

9. A method comprising:

a computer processor detecting a transactional region of code having a plurality of instructions; and

the computer processor determining that an instruction within the transactional region includes a prefix indicating that the instruction is to be executed as part of an atomic memory transaction that includes one or more other instructions in the transactional region.

10. The method of claim **9**, further comprising:

the computer processor determining that at least some other of the plurality of instructions are not to be executed as part of the atomic memory transaction; and

executing the at least some other instructions non-atomically.

11. The method of claim **9**, further comprising:

determining that execution of the instruction includes at least one implicit memory operation and in response, executing the at least one implicit memory operation as part of the atomic memory transaction.

12. The method of claim **9**, further comprising: executing the instruction as part of the atomic memory transaction, wherein said executing includes buffering versioning data for the instruction.

13. The method of claim **9**, wherein the instruction indicates the start of the transactional region of code.

14. The method of claim **13**, further comprising: in response to the instruction indicating the start of the transactional region of code, determining that all of the plurality of instructions are to be executed as part of the atomic memory transaction.

15. The method of claim **9**, wherein the one or more other instructions in the transactional region included in the atomic transaction also include the prefix.

16. The method of claim **9**, further comprising:

attempting to execute the atomic memory transaction, wherein said attempting includes:

detecting an abort condition; and

in response to detecting the abort condition, aborting execution of the transactional region of code at least by undoing memory effects of one or more instructions within the transactional region that include the prefix without undoing memory effects of one or more instructions within the transactional region that do not include the prefix; and

reattempting to execute the transactional region of code.

17. A computer-readable storage medium having stored thereon program instructions executable by a processor, wherein the program instructions comprise:

a plurality of instructions in a transactional region of code, the instructions executable by the processor in a transactional mode of execution;

wherein at least some of the instructions in the transactional region include a prefix that indicates to the processor that memory operations performed by the processor as part of executing the instructions that include the prefix are to be performed as a single atomic memory transaction.

18. The computer-readable storage medium of claim **17**, wherein the plurality of instructions include:

a transaction-initiating instruction executable by the processor to begin the transactional mode of execution;

a transaction-terminating instruction executable by the processor to exit the transactional mode of execution;

wherein the processor is configured to determine that the transaction-initiating instruction includes the prefix and in response, to execute all memory operations performed as part of executing the plurality of instructions in the transactional region as part of the atomic memory transaction.

19. The computer-readable storage medium of claim **17**, wherein the plurality of instructions include:

a transaction-initiating instruction executable by the processor to begin the transactional mode of execution;

a transaction-terminating instruction executable by the processor to exit the transactional mode of execution;

intermediate instructions appearing between the transaction-initiating instruction and transaction-terminating instruction in program execution order;

wherein each of two or more of the intermediate instructions includes a prefix indicating to the processor that the two or more intermediate instructions are to be executed together as part of the atomic memory transaction, and wherein at least one of the intermediate instructions does not include the prefix.

20. The computer-readable storage medium of claim **17**, wherein the transaction-initiating instruction includes an operand indicating a memory address to which execution should jump in the event that an attempt to atomically execute two or more intermediate instruction appearing between the transaction-initiating instruction and transaction-terminating instruction in program execution order is aborted.

21. A computer readable storage medium comprising a data structure that is operated upon by a program executable on a computer system, the program operating on the data structure to perform a portion of a process to fabricate an integrated circuit including circuitry described by the data structure, the circuitry described in the data structure including:

a computer processor configured to determine whether an instruction within a plurality of instructions in a transactional region of code includes a prefix indicating that the instruction is to be executed speculatively within a single atomic memory transaction.

* * * * *