

(19) World Intellectual Property Organization
International Bureau



(43) International Publication Date
18 June 2009 (18.06.2009)

PCT

(10) International Publication Number
WO 2009/076324 A2

(51) International Patent Classification:

G06F 9/46 (2006.01) **G06F 15/16** (2006.01)

G06F 9/44 (2006.01) **G06F 13/00** (2006.01)

(21) International Application Number:

PCT/US2008/085990

(22) International Filing Date:

8 December 2008 (08.12.2008)

(25) Filing Language:

English

(26) Publication Language:

English

(30) Priority Data:

61/012,741 10 December 2007 (10.12.2007) US

(71) Applicant (for all designated States except US): **STRANDERA CORPORATION** [US/US]; 530 Showers Drive, Suite 7-352, Mountain View, CA 94040 (US).

(72) Inventor; and

(75) Inventor/Applicant (for US only): **YOURST, Matt** [US/US]; 530 Showers Drive, Suite 7-352, Mountain View, CA 94040 (US).

(74) Agent: **VAN DYKE, Korbin, S**; 3343 Little Valley Rd., Sunol, CA 94586 (US).

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

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

Declaration under Rule 4.17:

— of inventorship (Rule 4.17(iv))

Published:

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

(54) Title: STRAND-BASED COMPUTING HARDWARE AND DYNAMICALLY OPTIMIZING STRANDWARE FOR A HIGH PERFORMANCE MICROPROCESSOR SYSTEM

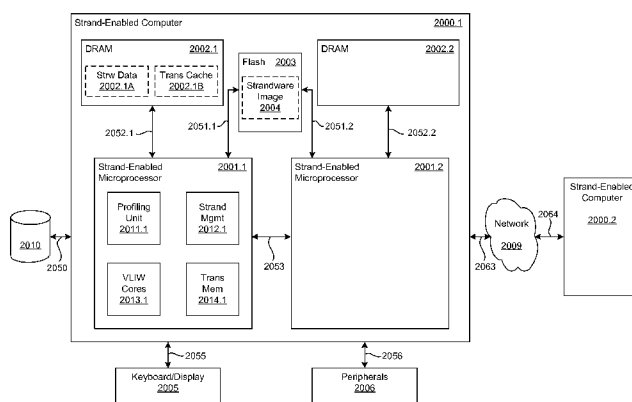


Fig. 1A

(57) Abstract: Strand-based computing hardware and dynamically optimizing strandware are included in a high performance microprocessor system. The system operates in real time automatically and unobservably to parallelize single-threaded software into a plurality of parallel strands for execution by cores implemented in a multi-core and/or multi-threaded microprocessor of the system. The microprocessor executes a native instruction set tailored for speculative multithreading. The strandware directs hardware of the microprocessor to collect dynamic profiling information while executing the single-threaded software. The strandware analyzes the profiling information for the parallelization, and uses binary translation and dynamic optimization to produce native instructions to store in a translation cache later accessed to execute the produced native instructions instead of some of the single-threaded software. The system is capable of parallelizing a plurality of single-threaded software applications (e.g. application software, device drivers, operating system routines or kernels, and hypervisors).

1 STRAND-BASED COMPUTING HARDWARE AND DYNAMICALLY OPTIMIZING
2 STRANDWARE FOR A HIGH PERFORMANCE MICROPROCESSOR SYSTEM
3
4
5

6 BACKGROUND
7

8 [0001] Field: Advancements in computer processing are needed to provide
9 improvements in performance, efficiency, and utility of use.
10

11 [0002] Related Art: Unless expressly identified as being publicly or well known,
12 mention herein of techniques and concepts, including for context, definitions, or comparison
13 purposes, should not be construed as an admission that such techniques and concepts are
14 previously publicly known or otherwise part of the prior art. All references cited herein (if any),
15 including patents, patent applications, and publications, are hereby incorporated by reference in
16 their entireties, whether specifically incorporated or not, for all purposes.
17
18

19 OVERVIEW
20

21 [0003] The invention may be implemented in numerous ways, including as a process,
22 an article of manufacture, an apparatus, a system, and a computer readable medium (e.g. media
23 in an optical and/or magnetic mass storage device such as a disk, or an integrated circuit having
24 non-volatile storage such as flash storage). In this specification, these implementations, or any
25 other form that the invention may take, may be referred to as techniques. The Detailed
26 Description provides an exposition of one or more embodiments of the invention that enable
27 improvements in performance, efficiency, and utility of use in the field identified above. As is
28 discussed in more detail in the Conclusions, the invention encompasses all possible
29 modifications and variations within the scope of the issued claims.
30

Brief Description of Drawings

[0004] Fig. 1A illustrates a system with strand-enabled computers each having one or more strand-enabled microprocessors with access to a strandware image, memory, non-volatile storage, input/output devices, and networking.

[0005] Figs. 1B and 1C collectively illustrate conceptual hardware, strandware (software), and target software layers (e.g. subsystems) relating to a strand-enabled microprocessor.

[0006] Figs. 2A, 2B, and 2C collectively illustrate an example of hardware executing a skipahead strand (such as synthesized by strandware), plotted against time in cycles versus core or interconnect. Sometimes the description refers to Figs. 2A, 2B, and 2C as Fig. 2.

DETAILED DESCRIPTION

[0007] A detailed description of one or more embodiments of the invention is provided below along with accompanying figures illustrating selected details of the invention. The invention is described in connection with the embodiments. The embodiments herein are understood to be merely exemplary, the invention is expressly not limited to or by any or all of the embodiments herein, and the invention encompasses numerous alternatives, modifications, and equivalents. To avoid monotony in the exposition, a variety of word labels (including but not limited to: first, last, certain, various, further, other, particular, select, some, and notable) may be applied to separate sets of embodiments; as used herein such labels are expressly not meant to convey quality, or any form of preference or prejudice, but merely to conveniently distinguish among the separate sets. The order of some operations of disclosed processes is alterable within the scope of the invention. Wherever multiple embodiments serve to describe variations in process, method, and/or program instruction features, other embodiments are contemplated that in accordance with a predetermined or a dynamically determined criterion perform static and/or dynamic selection of one of a plurality of modes of operation corresponding respectively to a plurality of the multiple embodiments. Numerous specific details are set forth in the following description to provide a thorough understanding of the invention. These details are provided for the purpose of example and the invention may be practiced according to the claims without some or all of the details. For the purpose of clarity, technical material that is known in the technical fields related to the invention has not been described in detail so that the invention is not unnecessarily obscured.

1 OVERVIEW

2

3 Terms

4

5 **[0008]** The disclosure herein uses various terms. Examples of at least some of the
6 terms follow.

7

8 **[0009]** An example of a thread is a software abstraction of a processor, e.g. a dynamic
9 sequence of instructions that share and execute upon the same architectural machine state (e.g.
10 software visible state). Some (so-called single-threaded) processors are enabled to execute one
11 sequence of instructions on one architectural machine state at a time. Some (so-called
12 multithreaded) processors are enabled to execute N sequences of instructions on N architectural
13 machine states at a time. In some systems, an operating system creates, destroys, and schedules
14 threads on available hardware resources.

15

16 **[0010]** An example of a strand is an abstraction of processor hardware, e.g. a dynamic
17 sequence of uops (e.g. micro-operations directly executable by the processor hardware) that
18 share and execute upon the same machine state. For some strands the machine state is
19 architectural machine state (e.g. architectural register state), and for some strands the machine
20 state is not visible to software (e.g. renamed register state, or performance analysis registers). In
21 some embodiments, a strand is visible to an operating system if machine state of the strand
22 includes all architectural machine state of a thread (e.g. general-purpose registers, software
23 accessible machine state registers, and memory state). In some embodiments, a strand is not
24 visible to an operating system, even if machine state of the strand includes all architectural
25 machine state of a thread.

26

27 **[0011]** An example of an architectural strand is a strand that is visible to an operating
28 system and corresponds to a thread. An example of a speculative strand (e.g. a successor strand)
29 is a strand that is not visible to the operating system. Certain strands contain only hidden
30 machine state (e.g. prefetch or profiling strands).

31

32 **[0012]** In some embodiments, strandware and/or processor hardware create, destroy,
33 and schedule strands. In some embodiments, forks create strands. Some forks are in response to
34 a uop (of a parent strand) that specifies a target address (for the strand created by the fork) and
35 optionally specifies other information (e.g., data to be inherited as machine state). When the uop
36 of the (parent) strand is executed, a speculative successor strand is optionally created.

1
2 **[0013]** In various embodiments and/or usage scenarios, strands are destroyed in
3 response to one or more of a kill uop, an unrecoverable error, and completion of the strand (e.g.
4 via a join). In some embodiments and/or usage scenarios, strands are joined in response to a join
5 uop. In some embodiments and/or usage scenarios, strands are joined in response to a set of
6 hardware-detected conditions (e.g. a current execution address matching a starting address of a
7 successor strand). In various embodiments, strands are destroyed by any combination of
8 strandware and/or hardware (e.g. in response to processing a uop or automatically in response to
9 a predetermined or programmatically specified condition). In some usage scenarios, strands are
10 joined by merging some machine state of a parent architectural strand with machine state of a
11 successor strand of the parent; then the parent is destroyed and the child strand optionally
12 becomes an architectural strand.

13
14 **[0014]** An example of a Virtual Central Processing Unit (VCPU) is a software visible
15 execution context that is enabled for an operating system to schedule one thread onto at any
16 particular time. In some embodiments, a computer system presents one or more VCPUs to the
17 operating system. Each VCPU implements a register portion of the architectural machine state,
18 and in some embodiments, architectural memory state is shared between one or more VCPUs.
19 Conceptually each VCPU comprises one or more strands dynamically created by strandware
20 and/or hardware. For each VCPU, the strands are arranged into a first-in first-out (FIFO) queue,
21 where the next strand to commit is the architectural strand of the VCPU, and all other strands are
22 speculative.

23 24 25 MULTI-CORE, MULTITHREADING, AND SPECULATION

26 27 Microprocessors, Multi-Core, and Multithreading

28
29 **[0015]** Performance of microprocessors has grown since introduction of the first
30 microprocessor in the 1970s. Some microprocessors have deep pipelines and/or operate at
31 multi-GHz clock frequencies to extract performance with a single processor out of sequential
32 programs. Software engineers write some programs as a sequence of instructions and operations
33 that a microprocessor is to execute sequentially and/or in order. Various microprocessors
34 attempt to increase performance of the programs by operating at an increased clock frequency,
35 executing instructions out-of-order (OOO), executing instructions speculatively, or various
36 combinations thereof. Some instructions are independent of other instructions, thus providing

1 instruction level parallelism (ILP), and therefore are executable in parallel or OOO. Some
2 microprocessors attempt to exploit ILP to improve performance and/or increase utilization of
3 functional units of the microprocessor.

4
5 **[0016]** Some microprocessors (sometimes referred to as multi-core microprocessors)
6 have more than one “core” (e.g. processing unit). Some single chip implementations have an
7 entire multi-core microprocessor, in some instances with shared cache memory and/or other
8 hardware shared by the cores. In some circumstances, an agent (e.g. strandware) partitions a
9 computing task into threads, and some multi-core microprocessors enable higher performance by
10 executing the threads in parallel on the of cores of the microprocessor. Some microprocessors
11 (such as some multi-core microprocessors) have cores that enable simultaneous multithreading
12 (SMT).

13
14 **[0017]** Some microprocessors that are compatible with an x86 instruction set (such as
15 some microprocessors from Intel and AMD) have a relatively few replications of (relatively
16 complex) OOO cores. Some microprocessors (such as some microprocessors from Sun and
17 IBM) have relatively many replications of (relatively simple) in-order cores. Some server and
18 multimedia applications are multithreaded, and some microprocessors with relatively many
19 cores perform relatively well on the multithreaded software.

20
21 **[0018]** Some multi-core microprocessors perform relatively well on software that has
22 relatively high thread level parallelism (TLP). However, in some circumstances, some resources
23 of some multi-core microprocessors are unused, even when executing software that has
24 relatively high TLP. Software engineers striving to improve TLP use mechanisms that
25 coordinate access to shared data to avoid collisions and/or incorrect behavior, mechanisms that
26 ensure smooth and efficient parallel interlocking by reducing or avoiding interlocking between
27 threads, and mechanisms that aid debugging of errors that appear in multithreaded
28 implementations.

29
30 **[0019]** With respect to some problem domains, some compilers automatically
31 recognize seemingly sequential operations of a thread as divisible into parallel threads of
32 operations. Some sequences of operations are indeterminate with respect to independence and
33 potential for parallel execution (e.g. portions of code produced from some general-purpose
34 programming languages such as C, C++, and Java). Software engineers sometimes use some
35 special-purpose programming languages (or parallel extensions to general-purpose programming
36 languages) to express parallelism explicitly, and/or to program multi-core and/or multithreaded

1 microprocessors or portions thereof (such a graphics processing unit or GPU). Software
2 engineers sometimes express parallelism explicitly for some scientific, floating-point, and media
3 processing applications.

4 5 6 Speculative Multithreading Fundamentals

7
8 **[0020]** In some usage scenarios and/or embodiments, speculative multithreading, thread
9 level speculation, or both enable more efficient automatic parallelization. In a speculative
10 multithreading microprocessor system, compiler software, strandware, firmware, microcode, or
11 hardware units of the microprocessor, or any combination thereof, conceptually insert one or
12 more instances of a selected one of a plurality of types of fork instructions into various locations
13 of a program. Conceptually, the system begins executing a (new) successor strand at a target
14 address inside the program, and manages propagation of register values (and optionally memory
15 stores) to the successor strand from the (parent) strand the successor strand was forked from.
16 The propagation is either via stalling the successor strand until the values arrive, or by predicting
17 the values and later comparing the predicted values with values generated by the parent strand.
18 The system creates the successor strand as a subset of a thread (e.g., the successor strand
19 receives a subset of architectural state from the thread and/or the successor strand executes a
20 subset of instructions of the thread). The fork instruction specifies the target address as a
21 Register for Instruction Pointer (RIP). The system implements strand management functions
22 (e.g. forking and joining) in various embodiments via various hardware elements (such as logic
23 units, finite state machines, micro-coded engines, and other circuitry), various software elements
24 (such as instructions executable by a core, firmware, microcode, strandware, and other software
25 agents), or various combinations thereof.

26
27 **[0021]** The speculative multithreading microprocessor system processes join operations
28 in (original) program order. Consider a parent strand that forks a successor strand to a target
29 address. A join occurs when the parent strand executes up to the target address (sometimes
30 referred to as an intersection). In some circumstances, the successor strand has completed (in
31 parallel with the parent strand), and the successor strand is immediately ready to join. At a join
32 point, the system performs various consistency checks, such as ensuring (potentially predicted)
33 live-out register values the parent strand propagated to the successor strand match actual values
34 of the parent strand at the join point. The checks guarantee that execution results with the forked
35 strand are identical to results without the forked strand. If any of the checks fail, then the system
36 takes appropriate action (such as by discarding results of the forked strand). After a join of

parent and successor strands, the parent strand terminates. The system then makes the context of the parent strand available for reuse. The successor strand becomes the architecturally visible instance of the thread that the system created the strand for. The system makes current architectural state of the successor strand (e.g. registers and memory) observable to other threads within the microprocessor (such as a thread on another core), other agents of the microprocessor (such as DMA), and devices outside the microprocessor.

[0022] Some speculative multithreading systems implement a nested strand model. For example, a parent strand P forks a primary successor strand S, and recursively forks sub-strands P1, P2, and P3. The system nests the sub-strands within the parent strand. The sub-strands execute independently of S and each other. P joins with S conditionally upon completion all of the sub-strands of P. In contrast, other speculative multithreading systems implement a strictly program ordered non-nested speculative multithreading model. For example, each parent strand P has at most one forked successor strand S outstanding at any time. P forks no more strands until either P intersects with S (resulting in a join) or S no longer executes. In some circumstances, implementing a non-nested model uses less and/or simpler hardware than implementing a nested model. Some usage scenarios with unmodified sequential programs are suitable for use with a non-nested model implementation.

[0023] Some speculative multithreading systems use memory versioning. For example, a successor strand that (speculatively) stores to a particular memory location uses a private version of the location, observable to strands that are later in program order than the successor strand, but not observable to other strands (that are earlier in program order than the successor strand). The system makes the speculative stores observable (in an atomic manner) to other agents when joining the successor and the parent strands. The other agents include strands other than the successor (and later) strands, other threads or units (such as DMA) of the microprocessor, devices external to the microprocessor, and any element of the system that is enabled to access memory. In some circumstances, the system accumulates several kilobytes of speculative store data before a join. Consider a situation where a parent strand (later in program order) is to write a memory location and a successor strand of the parent strand is to read the memory location. If the successor strand reads the memory location before the parent strand writes the memory location, then the system aborts the successor strand. The disclosure sometimes refers to the aforementioned situation as cross-strand memory aliasing. In some scenarios, the system reduces (or avoids) occurrences of cross-strand memory aliasing by choosing fork points resulting in little (or no) cross-strand memory aliasing.

1 **[0024]** Conceptually, the system arranges the strands belonging to a particular thread in
2 a program ordered queue, similar to individual instructions of a reorder buffer (ROB) in an out-
3 of-order processor. The system processes strand forks and joins in program order. The strand at
4 the head of the queue is the architectural strand, and is the only strand enabled to execute a join
5 operation, while subsequent strands are speculative strands. In some scenarios, strands contain
6 complex control flow (such as branches, calls, and loops) independent of other strands. In some
7 circumstances, strands execute thousands of instructions between creation (at a fork point) and
8 termination (at a join point). In some situations, relatively large amounts of strand level
9 parallelism are available over the thousands of instructions even with relatively few outstanding
10 strands.

11

12 **[0025]** Some systems use speculative multithreading for a variety of purposes (such as
13 prefetching), while some systems use speculative multithreading only for prefetching. For
14 example, a particular strand encounters a cache miss while executing a load instruction that
15 results in an access to a relatively slow L3 cache or main memory. The system forks a prefetch
16 strand from the load instruction, and stalls the particular strand. The system continues to stall
17 the particular strand while waiting for return data for the (missing) load. Unlike some other
18 types of strands, a missing load does not block a prefetch strand, but rather provides a predicted
19 or a dummy value without waiting for the miss to be satisfied. In various usage scenarios,
20 prefetch strands enable prefetching for loads that have addresses calculated independently of an
21 initial missing load, enable prefetching for loads related to processing a linked list, enable tuning
22 or pre-correcting a branch predictor, or any combination thereof. A prefetch strand forked in
23 response to a missing load is aborted when the missing load is satisfied e.g. since the prefetch
24 strand used predicted or dummy values and is not suitable for joining to another strand.

25

26 **[0026]** In some circumstances, performance improvements obtained via speculative
27 multithreading depend on particular choices of fork and join points. In some embodiments, the
28 system places fork points at control quasi-independent points, e.g. points that all possible
29 execution paths eventually reach. For example, with respect to a current iteration of a loop, the
30 system forks a strand starting at the iteration immediately following the current iteration, thus
31 enabling the two strands to execute wholly or partially in parallel. For another example (e.g.
32 when iterations of the loop are interdependent), the system forks a strand to execute code that
33 follows a loop end, enabling iterations of the loop to execute in one strand while the code after
34 the loop executes in another strand. For another example, the system forks a strand to start
35 executing code that follows a return from a called function (optionally predicting a return value
36 of the called function), enabling the called function and the code following the return to execute

1 wholly or partially in parallel via two strands. In various embodiments, fork points are inserted
2 by one or more of: automatically by a compiler and/or strandware (optionally based at least in
3 part on profiling execution, analyzing dynamic program behavior, or both), automatically by
4 hardware, and manually by a programmer.

5
6 **[0027]** Various embodiments of speculative multithreading are automatic and/or
7 unobservable. Some of the automatic and/or unobservable speculative multithreading
8 embodiments are applicable to all types of target software (e.g. application software, device
9 drivers, operating system routines or kernels, and hypervisors) without any programmer
10 intervention. (Note that the description sometimes refers to target software as target code, and
11 the target code is comprised of target instructions.) Some of the automatic and/or unobservable
12 speculative multithreading embodiments are compatible with industry-standard instruction sets
13 (such as an x86 instruction set), industry-standard programming tools or languages (such as C,
14 C++, and other languages), and industry-standard general-purpose computer systems (such as
15 servers, workstations, desktop computers, and notebook computers).

18 SYSTEM ARCHITECTURE

20 System of Strand-Enabled Computers

21
22 **[0028]** Fig. 1A illustrates a system with strand-enabled computers, each having one or
23 more strand-enabled microprocessors with access to a strandware image, memory, non-volatile
24 storage, input/output devices, and networking. Conceptually the system executes the strandware
25 to observe (via hardware assistance) and analyze dynamic execution of (e.g. x86) instructions of
26 target software (e.g. application, driver, operating system, and hypervisor software). The
27 strandware uses the observations to determine how to partition the x86 instructions into a
28 plurality of strands suitable for parallel execution on VLIW core resources of the strand-enabled
29 microprocessors. The strandware translates the partitioned instructions into operations (e.g.
30 micro-operations or uops), and then arranges the operations into bundles for efficient execution
31 on the VLIW core resources. The strandware stores the bundles in a translation cache for later
32 use (e.g. as one or more strand images). The translation optionally includes augmentation with
33 additional operations having no direct correspondence to the x86 instructions (e.g. to improve
34 performance or to enable parallel execution of the strands). The system subsequently arranges
35 for execution of and executes the stored bundles (e.g. strand images instead of portions of the
36 x86 instructions) to attempt to improve performance. In some embodiments, one or more of the

1 observing, analyzing, partitioning, and the arranging for and execution of are with respect to
2 traces of instructions.

3
4 **[0029]** The figure illustrates Strand-Enabled Computers **2000.1-2000.2**, enabled for
5 communication with each other via couplings **2063**, **2064**, and Network **2009**. Strand-Enabled
6 Computer **2000.1** couples to Storage **2010** via coupling **2050**, Keyboard/Display **2005** via
7 coupling **2055**, and Peripherals **2006** via coupling **2056**.

8
9 **[0030]** The Network is any communication infrastructure that enables communication
10 between the Strand-Enabled Computers, such as any combination of a Local Area Network
11 (LAN), Metro Area Network (MAN), Wide Area Network (WAN), and the Internet. Coupling
12 **2063** is compatible with, for example, Ethernet (such as 10Base-T, 100Base-T, and 1 or 10
13 Gigabit), optical networking (such as Synchronous Optical NETworking or SONET), or a node
14 interconnect mechanism for a cluster (such as Infiniband, MyriNet, QsNET, or a blade server
15 backplane network). The Storage element is any non-volatile mass-storage element, array, or
16 network of same (such as flash, magnetic, or optical disk(s), as well as elements coupled via
17 Network Attached Storage or NAS and/or Storage Array Network or SAN techniques).
18 Coupling **2050** is compatible with, for example, Ethernet or optical networking, Fibre Channel,
19 Advanced Technology Attachment or ATA, Serial ATA or SATA, external SATA or eSATA, as
20 well as Small Computer System Interface or SCSI.

21
22 **[0031]** The Keyboard/Display element is conceptually representative of any type of one
23 or more of alphanumeric, graphical, or other human input/output device(s) (such as a
24 combination of a QWERTY keyboard, an optical mouse, and a flat-panel display). Coupling
25 **2055** is conceptually representative of one or more couplings enabling communication between
26 the Strand-Enabled Computer and the Keyboard/Display. In one example, one element of
27 coupling **2055** is compatible with a Universal Serial Bus (USB) and another element is
28 compatible with a Video Graphics Adapter (VGA) connector. The Peripherals element is
29 conceptually representative of any type of one or more input/output device(s) usable in
30 conjunction with the Strand-Enabled Computer (such as a scanner or a printer). Coupling **2056**
31 is conceptually representative of one or more couplings enabling communication between the
32 Strand-Enabled Computer and the Peripherals.

33
34 **[0032]** In various embodiments (not illustrated), various elements illustrated as external
35 to the Strand-Enabled Computer (such as Storage **2010**, Keyboard/Display **2005**, and Peripherals
36 **2006**), are included in the Strand-Enabled Computer. In some embodiments, one or more of

Strand-Enabled Microprocessors **2001.1-2001.2** include hardware to enable coupling to elements identical or similar in function to any of the elements illustrated as external to the Strand-Enabled Computer. In various embodiments, the included hardware is compatible with one or more particular protocols, such as one or more of a Peripheral Component Interconnect (PCI) bus, a PCI eXtended (PCI-X) bus, a PCI Express (PCI-E) bus, a HyperTransport (HT) bus, and a Quick Path Interconnect (QPI) bus. In various embodiments, the included hardware is compatible with a proprietary protocol used to communicate with an (intermediate) chipset that is enabled to communicate via any one or more of the particular protocols.

[0033] In some embodiments, the Strand-Enabled Computers are identical to each other, and in other embodiments the Strand-Enabled Computers vary according to differences relating to market and/or customer requirements. In some embodiments, the Strand-Enabled Computers operate as server, workstation, desktop, notebook, personal, or portable computers.

[0034] As illustrated, Strand-Enabled Computer **2000.1** includes two Strand-Enabled Microprocessors **2001.1-2001.2** coupled respectively to Dynamic Random Access Memory (DRAM) elements **2002.1-2002.2**. The Strand-Enabled Microprocessors communicate with Flash **2003** respectively via couplings **2051.1-2051.2** and with each other via coupling **2053**. Strand-Enabled Microprocessor **2001.1** includes Profiling Unit **2011.1**, Strand Management unit **2012.1**, VLIW Cores **2013.1**, and Transactional Memory **2014.1**.

[0035] In some embodiments, the Strand-Enabled Microprocessors are identical to each other, and in other embodiments the Strand-Enabled Microprocessors vary according to differences relating to market and/or customer requirements. In various embodiments, a Strand-Enabled Microprocessor is implemented in any of a single integrated circuit die, a plurality of integrated circuit dice, a multi-die module, and a plurality of packaged circuits.

[0036] For brevity, the following description is with respect to one of the illustrated Strand-Enabled Microprocessors. Operation of the other Strandware-Enabled Strand-Enabled Microprocessors is similar. Strandware-Enabled Microprocessor **2001.1** exits a reset state (such as when performing a cold boot) and begins fetching and executing instructions of strandware from a code portion of Strandware Image **2004** contained in Flash **2003**. The execution of the instructions initializes various strandware data structures (e.g. Strandware Data **2002.1A** and Translation Cache **2002.1B**, illustrated as portions of DRAM **2002.1**). The initializing includes copying all or any subsets of the code portion of the Strandware Image to a portion of the

1 Strandware Data, and setting aside regions of the Strandware Data for strandware heap, stack,
2 and private data storage.

3
4 **[0037]** Then the Strand-Enabled Microprocessor begins processing x86 instructions
5 (such as x86 boot firmware contained, in some embodiments, in the Flash), subject to the
6 aforementioned observing (via at least in part Profiling Unit **2011.1**) and analyzing. The
7 processing is further subject to the aforementioned partitioning into strands for parallel
8 execution, translating into operations and arranging into bundles corresponding to various strand
9 images, and storage into translation cache (such as Translation Cache **2002.1B**). The processing
10 is further subject to the aforementioned subsequent arranging for and execution of the stored
11 bundles (via at least in part Strand Management unit **2012.1**, VLIW Cores **2013.1**, and
12 Transactional Memory **2014.1**).

13
14 **[0038]** Partitioning of elements illustrated in the figure is illustrative only, as there are
15 other embodiments with other partitioning. For example, various embodiments include all or
16 any portion of the Flash and/or the DRAM in a Strand-Enabled Microprocessor. For another
17 example, various embodiments include storage for all or any portion of the Strandware Data
18 and/or the Translation Cache in a Strand-Enabled Microprocessor (such as in one or more Static
19 Random Access Memories or SRAMs on an integrated circuit die). For another example, in
20 some embodiments, Strandware Data **2002.1A** and Translation Cache **2002.1B** are contained in
21 different DRAMs (such as one in a first Dual In-line Memory Module or DIMM and another in a
22 second DIMM). For another example, various embodiments store all or any portion of the
23 Strandware Image on Storage **2010**.

24
25 Massively Multithreaded Hardware and Strandware

26
27 **[0039]** Figs. 1B and 1C collectively illustrate conceptual hardware, strandware
28 (software), and target software layers (e.g. subsystems) relating to a strand-enabled
29 microprocessor (such as either of Strand-Enabled Microprocessors **2001.1-2001.2** of Fig. 1A).
30 The figure is conceptual in nature, and for brevity, the figure omits various control and some
31 data couplings.

32
33 **[0040]** Hardware Layer **190** includes one or more independent cores (e.g. instances of
34 VLIW Cores **191.1-191.4**), each core enabled to process in accordance with one or more
35 hardware thread contexts (e.g. stored in instances of Register Files **194A.1-194A.4** and/or Strand
36 Contexts **194B.1-194B.4**), suitable for simultaneous multithreading (SMT) and/or hardware

context switching. The microprocessor is enabled to execute instructions in accordance with an instruction set architecture. The microprocessor includes speculative multithreading extensions and enhancements, such as hardware to enable processing of fork and join instructions and/or operations, inter-thread and inter-core register propagation logic and/or circuitry (Multi-Core Interconnect Network **195**), Transactional Memory **183** enabling memory versioning and conflict detection capabilities, Profiling Hardware **181**, and other hardware elements that enable speculative multithreading processing. In the illustrated embodiment, the microprocessor also includes a multi-level cache hierarchy (e.g. instances of L1 D-Caches **193.1-193.4** and L2/L3 Caches **196**), one or more interfaces to mass memory and/or hardware devices external to the microprocessor (DRAM Controllers and Northbridge **197** coupled to external System/Strandware DRAM **184A**), a socket-to-socket system interconnect (Multi-Socket System Interconnect **198**) useful, e.g. in a computer with a plurality of microprocessors (each microprocessor optionally including a plurality of cores), and interfaces/couplings to external hardware devices (Chipset/PCIe Bus Interface **186** for coupling via external PCI Express, QPI, HyperTransport **199**).

[0041] Strandware Layers **110A** and **110B** (sometimes referred to collectively as Strandware Layer **110**) and (x86) Target Software Layer **101** are executed at least in part by all or any portion of one or more cores included in and/or coupled to the microprocessor (such as any of the instances of VLIW Cores **191.1-191.4** of Fig. 1C). The strandware layer is conceptually invisible to elements of the target software layer, conceptually operating transparently “underneath” and/or “at the same level” as the target software layer. The target software layer includes Operating System Kernel **102** and programs (illustrated as instances of Application Programs **103.1-103.4**), illustrated as being executed “above” the operating system kernel. In some embodiments and/or usage scenarios, the target software layer includes a hypervisor program (e.g. similar to VMware or Xen) that manages a plurality of operating system instances.

[0042] In various embodiments, the strandware layer enables one or more of the following capabilities:

- Virtualization of the microprocessor hardware to present one or more virtual CPUs (e.g. instances of VCPUs **104.1-104.6**) and associated Virtual Devices **174** to the target software. The VCPUs appear to execute a target instruction set the Target Software Layer **101** is coded in. The VCPUs are dynamically mapped onto native cores (e.g. instances of VLIW Cores **191.1-191.4** that are

enabled to execute a native instruction set) and strand contexts (retained, e.g. in one or more instances of Register Files **194A.1-194A.4** and/or Strand Contexts **194B.1-194B.4**) of the microprocessor.

- Instrumentation, profiling, and analysis of the target software while the target software is executed, at least in part to identify opportunities for splitting (sequential) streams of instructions into speculatively multithreaded strands. For example, the system partitions respective sequential streams of instructions executed by one or more of the VCPUs into multiple speculatively multithreaded strands.
- Insertion of instructions and/or code sequences into the target software, based on the analysis, to invoke various speculative multithreading hardware units of the microprocessor to fork and join strands, to predict and/or propagate live-in values to strands, to manage memory versioning and conflicts between strands, and to fork prefetch strands.
- Optimization of target software to accelerate speculative multithreading performance, such as rescheduling instructions to generate critical strand live-ins values earlier in time, deferring and/or reordering operations that inhibit parallelism to break or eliminate cross-strand dependencies and remove memory aliasing, and removing redundant operations within prefetch strands.
- Maintenance of a repository of modified, instrumented, and/or optimized code (e.g. via Translation Cache Management **111**) so that the code in the repository is invisible to target code and is available to be invoked by the strandware in place of original target code (e.g. a portion of the target code before being modified, instrumented, or optimized).
- To process any internal exceptions or errors that are a result of any of the modifications, instrumentations, and optimizations (such as speculative multithreading) that would otherwise not have occurred when executing the target software. In some circumstances, the processing of the internal exceptions or errors includes re-optimizing and/or disabling optimizations that decrease performance.
- Providing an optional mechanism to target code for providing the strandware with hints, such as potentially profitable fork points, synchronization points, likely cross-strand aliasing points, and other optimization information.

1 Binary Translation and Dynamic Optimization

2
3 **[0043]** In some embodiments, the microprocessor hardware is enabled to execute an
4 internal instruction set that is different than the instruction set of the target software. The
5 strandware, in various embodiments, optionally in concert with any combination of one or more
6 hardware acceleration mechanisms, performs dynamic binary translation (such as via x86 Binary
7 Translation **115**) to translate target software of one or more target instruction sets (such as an
8 x86-compatible instruction set, e.g., the x86-64 instruction set) into native micro-operations
9 (uops). The hardware acceleration mechanisms include all or any portion of one or more of
10 Profiling Hardware **181**, Hardware Acceleration unit **182**, Transactional Memory **183**, and
11 Hardware x86 Decoder **187**. The microprocessor hardware (such as instances of VLIW Cores
12 **191.1-191.4**) is enabled to directly execute the uops (and in various embodiments, the
13 microprocessor hardware is not enabled to directly execute instructions of one or more of the
14 target instruction sets). The translations are then stored in a repository (e.g. via Translation
15 Cache Management **111**) for rapid recall and reuse (e.g. as strand images), thus eliminating
16 translating again, at least under some circumstances.

17
18 **[0044]** In various embodiments, the microprocessor is enabled to access (such as by
19 being coupled or attached to) a relatively large memory area. The system implements the
20 memory area via a dedicated DRAM module (included in or external to the microprocessor, in
21 various embodiments) or alternatively as part of a reserved area in external System/Strandware
22 DRAM **184A** that is invisible to target code. The memory area provides storage for various
23 elements of the strandware (such as one or more of code, stack, heap, and data) and, in some
24 embodiments, all or any portion of a translation cache (e.g. as managed by Translation Cache
25 Management **111**), as well as optionally one or more buffers (such as speculative multithreading
26 temporary state buffers). When the microprocessor first boots (such as by performing a cold
27 boot), the strandware code is copied from a flash ROM into the memory area (such as into the
28 dedicated DRAM module or a reserved portion of external System/Strandware DRAM **184A**),
29 that the microprocessor then fetches native uops from. After the strandware initializes the
30 microprocessor (such as via Hardware Control **172**) and internal data structures of the
31 strandware, the strandware begins execution of boot firmware and/or operating system kernel
32 boot code (coded in one or more of the target instruction sets) using binary translation (such as
33 via x86 Binary Translation **115**), similar to a conventional hardware based microprocessor
34 without a binary translation layer.

1 **[0045]** In some usage scenarios, using the strandware to perform binary translation
2 and/or dynamic optimization offers advantages compared to adding speculative multithreading
3 instructions to the target instruction set. In some circumstances, the binary translation and/or
4 dynamic optimization enable simplifying hardware of each core, for example by removing
5 and/or reducing hardware for decoding the target instruction sets (such as Hardware x86
6 Decoder **187**) and hardware for out-of-order execution. In some embodiments, the removed
7 and/or reduced hardware is conceptually replaced with one or more VLIW (Very Long
8 Instruction Word) microprocessor cores (such as instances of instances of VLIW Cores **191.1-**
9 **191.4**). The VLIW cores, for example, execute pre-scheduled bundles of uops, where all of the
10 uops of a bundle execute (or begin execution) in parallel (e.g. on a plurality of functional units
11 such as instances of ALUs **192A.1-192A.4** and FPUs **192B.1-192B.4**). In various embodiments,
12 the VLIW cores lack one or more of relatively complicated decoding, hardware-based
13 dependency analysis, and dynamic out of order scheduling. The VLIW cores optionally include
14 local storage (such as instances of L1 D- Caches **193.1-193.4** and Register Files **194A.1-194A.4**)
15 and other per-core hardware structures for efficient processing of instructions.

16
17 **[0046]** In some usage scenarios and/or embodiments, the VLIW cores are small enough
18 to enable one or more of packing more cores into a given die area, powering more cores within a
19 given power budget, and clocking cores at a higher frequency than would otherwise be possible
20 with complex out-of-order cores. In some usage scenarios and/or embodiments, semantically
21 isolating the VLIW cores from the target instruction sets via binary translation enables efficient
22 encoding of uop formats, registers, and various details of the VLIW core relevant to efficient
23 speculative multithreading, without modifying the target instruction sets.

24 25 Role of Strandware Dynamic Optimization Software

26
27 **[0047]** A trace construction subsystem of the strandware layer (such as Trace Profiling
28 and Capture **120**), when executed by the microprocessor, collects and/or organizes translated
29 uops into traces (e.g. from uops of a sequence of translated basic blocks having common control
30 flow paths through the target code). The strandware performs relatively extensive optimizations
31 (such as via Optimize **163**), using a variety of techniques. Some of the techniques are similar in
32 scope to what an optimizing compiler having access to source code performs, but the strandware
33 uses dynamically measured program behavior collected during profiling (such as via one or
34 more of Physical Page Profiling **121**, Branch Profiling **124**, Predictive Optimization **125**, and
35 Memory Profiling **127**) to guide at least some optimizations. For instance, loads and stores to
36 memory are selectively reordered (such as a function of information obtained via Memory

1 Aliasing Analysis 162) to initiate cache misses as early as possible. In some embodiments, the
2 selective reordering is based at least in part on measurements (such as made via Memory
3 Profiling 127) of loads and stores that reference a same address. In some usage scenarios and/or
4 embodiments, the selective reordering enables relatively aggressive optimizations over a scope
5 of hundreds of instructions. Each uop is then scheduled (such as by insertion into a schedule by
6 Schedule each uop 165) according to when input operands are to be available and when various
7 hardware resources (such as functional units) are to be free. In some embodiments (such as
8 some embodiments having functionality as illustrated by Encode VLIW-like bundles 167), the
9 scheduling attempts to pack up to four uops into each bundle. Having a plurality of uops in a
10 bundle enables a particular VLIW core (such as any of VLIW Cores 191.1-191.4) to execute the
11 uops in parallel when the scheduled trace is later executed. Finally, the optimized trace (having
12 VLIW bundles each having one or more uops) is inserted into a repository (such as via
13 Translation Cache Management 111) as all or part of a strand image. In some embodiments, the
14 hardware only executes native uops from traces stored in the translation cache, thus enabling
15 continuous reuse of optimization work performed by the strandware. In some usage scenarios
16 and/or embodiments, traces are successively re-optimized through a series of increasingly higher
17 performance optimization levels, each level being relatively more expensive to perform (such as
18 via Promote 130), depending, for example, on how frequently a trace is executed.

19
20 [0048] In some embodiments, the dynamic optimization software enables some
21 relatively aggressive optimizations via use of atomic execution. In some circumstances,
22 instances of the relatively aggressive optimizations would be “unsafe” without atomic execution,
23 e.g. incorrect modifications to architectural state would result. An example of atomic execution
24 is treating a group of uops (termed a commit group) as an indivisible unit with respect to
25 modifications to architectural state. A trace optionally comprises one or more commit groups.
26 If all of the uops of a commit group complete correctly (such as without any exceptions or
27 errors), then changes are made to the architectural state in accordance with results of all of the
28 uops of the commit group. Under other circumstances, the results of all of the uops of the
29 commit group are discarded, and there are no changes made to the architectural state with
30 respect to the uops of the commit group. For example, in the event of an exception detected
31 with respect to a uop of a commit group (such as a page fault or a branch that follows a different
32 path than the path that the trace was originally generated along), a rollback occurs, and all results
33 generated by all of the uops of the commit group are discarded. After a rollback, in some
34 embodiments and/or usage scenarios, the microprocessor and/or the strandware re-executes
35 instructions corresponding to the uops of the commit group in original program order (and
36 optionally without one or more optimizations) to pinpoint a source of the exception. Co-pending

1 U.S. patent application 10/994,774 entitled "Method and Apparatus for Incremental
2 Commitment to Architectural State" discloses other information regarding dynamic optimization
3 and commit groups.

4
5 **[0049]** The hardware and the software operating in combination enable, in some
6 embodiments and/or usage scenarios, benefits similar to an out-of-order dynamically scheduled
7 microprocessor, such as by extracting fine-grained parallelism within a single strand via
8 relatively aggressive VLIW trace scheduling and optimization. The hardware and the software
9 perform the fine-grained parallelism extracting, in various embodiments, while relatively
10 efficiently reordering and interleaving independent strands to cover memory latency stalls,
11 similar to an out-of-order microprocessor. In some circumstances, the hardware and the
12 software enable relatively efficient scaling across many cores and/or threads, enabling an
13 effective issue width of potentially hundreds of uops per clock.

14 15 Multithreaded Dynamic Optimization

16
17 **[0050]** In some embodiments having a massively multi-core and/or multithreaded
18 microprocessor, the dynamic optimization software is implemented to relatively efficiently use
19 resources of the plurality of cores and/or threads. For example, one or more of Trace Profiling
20 and Capture **120**, Strand Construction **140**, Scheduling and Optimization **160**, and x86 Binary
21 Translation **115** are pervasively multithreaded at one or more levels, enabling a reduction,
22 elimination, or effective hiding of some or all overhead associated with binary translation and/or
23 dynamic optimization. The microprocessor executes the dynamic optimization software in a
24 background manner so that forward progress in executing target code (e.g. through optimized
25 code from a translation cache) is not impeded. Various embodiments implement one or more
26 mechanisms to enable the background manner of executing the dynamic optimization software.
27 For example, the microprocessor and/or the strandware dedicate portions of resources (such as
28 one or more cores in a multi-core microprocessor embodiment) specifically to executing the
29 dynamic optimization software. The dedication is either permanent, or alternatively transient
30 and/or dynamic, e.g. when the portions of resources are available (such as when target code
31 explicitly places unused VCPUs into an idle state). For another example, priority control
32 mechanisms of one or more cores enable strandware threads (mapped, e.g. to target-visible
33 VCPUs) to share the cores and associated cache(s) with little or no observable performance
34 degradation (for instance, by using slack cycles created by stalled target threads executing in
35 accordance with a target Instruction Set Architecture or ISA).

Hardware and Strandware Implementation

[0051] In various embodiments, elements illustrated in Fig. 1A correspond to all or portions of functionality illustrated in Figs. 1B and 1C. For example, in some embodiments, DRAM **2002.1** of Fig. 1A corresponds to external System/Strandware DRAM **184A** of Fig. 1C, and Translation Cache Management **111** manages Translation Cache **2002.1B**. For another example, in some embodiments, VLIW Cores **2013.1** of Fig. 1A correspond to one or more of VLIW Cores **191.1-191.4** of Fig. 1C, Transactional Memory **2014.1** of Fig. 1A corresponds to Transactional Memory **183** of Fig. 1C, and Profiling Unit **2011.1** of Fig. 1A corresponds to Profiling Hardware **181** of Fig. 1C. For another example, in some embodiments Strand Management unit **2012.1** of Fig. 1A corresponds to control logic coupled to one or more of Register Files **194A.1-194A.4** and/or Strand Contexts **194B.1-194B.4** of Fig. 1C.

[0052] For another example of the correspondence between elements of Figs. 1A, 1B, and 1C, in some embodiments, Strandware Image **2004** of Fig. 1A has an initial image of all or any portion of Strandware Layers **110A** and **110B** of Figs. 1B and 1C. For another example, in some embodiments, Strand-Enabled Microprocessor **2001.1** of Fig. 1A implements functions as exemplified by Hardware Layer **190** of Fig. 1C.

[0053] In various embodiments, all or any portion of Chipset/PCIe Bus Interface **186**, Multi-Socket System Interconnect **198**, and/or PCI Express, QPI, HyperTransport **199** of Fig. 1C, implement all or any portion of interfaces associated with couplings **2050**, **2055**, **2056**, **2063**, **2051.1**, and **2053** of Fig. 1A. In various embodiments, all or any portion of Chipset/PCIe Bus Interface **186** and/or PCI Express, QPI, HyperTransport **199**, operating in conjunction with Interrupts, SMP, and Timers **175** of Fig. 1C, implement all or any portion of all or any portion of Keyboard/Display **2005** and/or Peripherals **2006** of Fig. 1A. In various embodiments, all or any portion of DRAM Controllers and Northbridge **197** of Fig. 1C, implement all or any portion of interfaces associated with coupling **2052.1** of Fig. 1A.

Speculative Multithreading Model

[0054] The speculative multithreading of various embodiments is for use on unmodified target code where an appearance of fully deterministic program ordered execution is always maintained. In some embodiments, the speculative multithreading provides a strictly program ordered non-nested speculative multithreading model where each parent strand has at most one successor strand at any given time. If a parent strand P forks a first child strand S1 and

1 then attempts to fork a second child strand S2 before joining with S1 and/or before S1
 2 terminates, then the fork of S2 is ineffective (e.g. the fork of S2 is suppressed such as by treating
 3 the fork of S2 as a no-operation or as a NOP). If a parent strand attempts a fork and there are not
 4 enough resources (e.g. there are no free thread contexts) to complete the fork, then the fork is
 5 suppressed or alternatively the forked thread is blocked until resources become available,
 6 optionally depending on what type of fork the fork is.

7
 8 **[0055]** In some embodiments, the microprocessor is enabled to execute in accordance
 9 with a native uop instruction set that includes a variety of uops, features, and internal registers
 10 usable to fork strands, control interactions between strands, join strands, and abort (e.g. kill)
 11 strands. In some embodiments, the variety of uops includes:

- 12 • *fork.type target, inherit* directs the microprocessor to create a new
 13 successor strand S of parent strand P. The microprocessor (via any combination
 14 of hardware and software elements) maps the successor strand to a specific core
 15 and thread of the microprocessor in accordance with one or more strandware
 16 and/or hardware defined policies. A particular VCPU executing a *fork* uop of
 17 a parent strand owns the successor strand (along with the parent strand).
 18 Execution of the successor strand begins at a target address specified by the
 19 *target* parameter (either in terms of a native uop address within a strandware
 20 address space or as a target code RIP). The *inherit* parameter is used as an
 21 indication of which registers will be modified by the parent strand after
 22 executing the fork operation, and which registers should be copied (inherited) to
 23 the successor strand (see the section “Skipahead Strands” located elsewhere
 24 herein). The *type* parameter specifies one of several different strand types for
 25 the successor strand (such as a fine-grained skipahead strand, a fully speculative
 26 multithreaded strand, a prefetch strand, or strands having other semantics or
 27 purposes). The *fork* uop provides an output value that is a strand ID. The
 28 strand ID is an identifier (that is globally unique at least within a same VCPU)
 29 associated with the successor strand that specifies the program order of the
 30 successor strand relative to all other strands that are associated with the
 31 particular VCPU owning both the parent and the successor strands.
- 32 • *kill.cmp type.cc ra, rb, T* directs the microprocessor to eliminate one
 33 or more strands. More specifically, when executed within parent strand P,
 34 *kill* recursively aborts successor strand S (if any) of P and all successor
 35 strands of S (if any). Execution of the *kill* uop compares register operands

1 *ra* and *rb* via specified ALU operation *cmptype* (e.g. *kill.sub* or
2 *kill.and*) thus generating a result, and then checks specified condition code
3 *cc* (e.g. less-than-or-equal) of the result. If the specified condition is true, the
4 strand scope identifier *T* matches the strand scope identifier of the associated
5 fork uop, and the nested fork depth is zero, then successor strands of parent
6 strand *P* are killed. See the sections “Strand Scope Identification” and “Nested
7 Strands” located elsewhere herein for further disclosure.

- 8 • *wait.type [object]* directs the microprocessor to stall execution pending
9 a specified condition. More specifically, when executed within strand *S*, *wait*
10 causes execution of strand *S* to wait on a specified condition (and optionally on
11 a specified object such as a memory address) before proceeding. For example,
12 in some embodiments, the microprocessor is enabled to wait until a strand is
13 architectural (e.g. non-speculative), to wait for a specific memory location to be
14 written, to wait until a successor strand completes, and to wait until a parent
15 strand reaches some state.
- 16 • *join* directs the microprocessor to block execution of a speculative successor
17 strand associated with a parent strand, until the parent strand joins with the
18 successor strand. The *join* uop is executed by the strandware when a
19 particular strand is unable to make forward progress while speculative.
- 20 • Uops optionally include a propagate bit that instructs the hardware to transmit
21 results of the uop (in a parent strand) to a successor strand of the parent strand.
22 See the section “Skipahead Strands” located elsewhere herein for further
23 disclosure relating to the propagate bit.

24
25 [0056] In some embodiments, some or all of the functionality of the aforementioned
26 uops is implemented by executing a plurality of other uops, performing writes to internal
27 machine state registers, invoking a separate non-uop-based hardware mechanism in an
28 optionally automatic manner, or any combination thereof.

29
30 [0057] In various usage scenarios where a parent strand forks a speculative successor
31 strand, there are several reasons for the successor strand to wait or stop execution (e.g. halt or
32 suspend) and wait for the parent strand to join the successor strand. For example, if an exception
33 occurs in a speculative strand, in some cases the exception indicates a mis-speculation or a
34 situation where it is not productive for the parent strand to have forked the successor strand. For
35 another example, a speculative strand attempts a particular operation that results in an exception

1 since the particular operation is restricted for use only in a (non-speculative) architectural strand.
2 Instances of the restricted operations optionally include accessing an I/O device (such as via PCI
3 Express, QPI, HyperTransport 199), reading or writing particular memory regions (such as
4 uncacheable memory), entering a portion of strandware that is limited to executing non-
5 speculatively, or attempting to use a deferred operation result.

6
7 **[0058]** When a parent strand intersects with a waiting successor strand and if the parent
8 verifies that all live-outs of the parent match the live-ins of the successor, then an exception of
9 the successor strand is “genuine”. The exception is genuine in the sense that the exception is not
10 a side effect of incorrect speculation and thus the microprocessor treats the exception in an
11 architecturally visible manner. In various cases, when execution of the successor strand
12 resumes, the successor strand immediately vectors based on the exception (such as into the
13 operating system kernel) to process the exception (e.g. a page fault). In some cases, when
14 execution of the successor strand resumes, execution continues without errors, since the
15 successor strand is now architectural (non-speculative).

16
17 **[0059]** The microprocessor joins strands in program order, and each VCPU owns one
18 or more of the strands. The most up-to-date architectural strand represents architectural state of
19 the VCPU owning the strand. The microprocessor makes the architectural state available for
20 observation outside of the owning VCPU (e.g. via a committed store to memory). The
21 microprocessor is enabled to freely move the most up-to-date architectural strand between cores
22 within the microprocessor, and meanwhile the owning VCPU appears to execute continuously
23 (observed, for example, by an operating system kernel executed with respect to the owning
24 VCPU).

25 26 Speculative Multithreading Strategies

27
28 **[0060]** The microprocessor hardware and the microprocessor strandware (software)
29 enable speculative multithreading on several levels with progressively wider scopes:

- 30 • A prefetch strand (see the section “Prefetch Strands” located elsewhere herein)
31 is optionally automatically forked when a strand stalls on a relatively long
32 latency cache operation (e.g. a cache miss that is satisfied from main memory).
33 A prefetch strand attempts to fetch data that is expected to be used into one or
34 more caches and/or attempts to prime one or more branch predictors with
35 appropriate data, before the data is used, for example before the data is accessed
36 by the (parent) strand the prefetch strand was forked from. In some

1 circumstances, a prefetch strand is active for several hundred cycles. In some
2 embodiments, the system provides for any type of strand to fork a prefetch
3 strand, as long as the forking strand has not forked another strand (thus
4 preventing scenarios where a particular strand has more than one successor
5 strand). In some embodiments, the system provides for any type of strand to
6 fork a prefetch strand, even when the forking strand has forked another strand
7 (leading to scenarios where a particular strand has more than one successor
8 strand). In some embodiments, the hardware has logic to selectively activate or
9 suppress creation of prefetch strands in accordance with one or more software
10 and/or strandware controllable prefetching policies.

- 11 • A skipahead strand (see the section “Skipahead Strands” located elsewhere
12 herein) is forked by a parent strand when strandware determines the parent
13 strand is relatively likely to stall on a particular instruction (e.g. a load that
14 relatively frequently encounters a cache miss). Alternatively, a skipahead strand
15 is forked so the skipahead strand begins executing after a relatively highly
16 predictable final branch (e.g. a branch that has a correct prediction rate greater
17 than a predetermined and/or programmable threshold). A skipahead strand
18 blocks until the parent strand provides live-ins the skipahead strand depends on,
19 for example, values for live-outs are transmitted to the skipahead strand (where
20 a subset of the live-outs of the parent strand are live-ins of the skipahead strand)
21 as the parent strand generates the live outs. The transmitted live-outs optionally
22 include registers and/or memory locations.

- 23 • A Speculative Strand Threading (SST) strand (see the section “Speculative
24 Strand Threading (SST)” located elsewhere herein) is forked based on
25 strandware dynamically (and optionally statically) inferring control flow
26 structures and idioms. The structures and idioms include iteration constructs
27 (e.g. loops), calls and returns (e.g. of subroutines, functions, procedures, and
28 libraries), and control flow joins (e.g. in a conditional block a common join
29 point reached by both “if” and “else” paths). An SST strand contains one or
30 more instruction sequences (e.g. basic blocks, traces, commit groups, or other
31 quanta of instructions). Dynamic control flow changes occur in some scenarios
32 at the end of each instruction sequence to determine the next instruction
33 sequence for the strand to execute. Control flow changes within an SST strand
34 (unlike some other strand types) occur independently of control flow within the
35 successor strands of the SST strand. The control flow changes within the SST
36 strand relatively infrequently invalidate the successor strands. In some

1 situations, the system selectively changes an SST strand to a prefetch strand. In
2 some circumstances, an SST strand is active for tens or hundreds of thousands
3 of cycles.

- 4 • A profiling strand (see the section “Instrumentation for Profiling” located
5 elsewhere herein) is used, in some embodiments, during the construction of SST
6 strands to gather cross-strand forwarding data. With respect to other strands, a
7 profiling strand is executed serially (e.g. in program order) rather than in
8 parallel with the parent strand of the profiling strand.

11 PREFETCH STRANDS

12
13 **[0061]** In some circumstances of strand execution, the execution encounters stalling
14 events (e.g., a cache miss to main memory) that would otherwise block progress. In response,
15 the microprocessor optionally forks a prefetch strand while stalling the strand encountering the
16 stalling event. The microprocessor allocates the (new) prefetch strand (in some embodiments,
17 on the same core as the parent strand, but in a different strand context), such that the prefetch
18 strand starts with the architectural state (register and memory) of the parent strand. The prefetch
19 strand continues executing until delivery of information to the stalled (parent) strand enables the
20 stalled strand to resume processing (e.g., data for the cache miss is delivered to the stalled
21 strand). Then the microprocessor (e.g. elements of Hardware Layer 190) automatically destroys
22 the prefetch strand and unblocks the stalled strand. In some embodiments, the microprocessor
23 has logic to selectively activate or suppress creation of prefetch strands in accordance with one
24 or more software and/or strandware controllable prefetching policies. For example, strandware
25 configures the microprocessor to fork a prefetch strand when an L1 miss encountered by a strand
26 results in a main memory access, and to stall the strand when an L1miss results in an L2 or L3
27 hit.

28
29 **[0062]** In some circumstances of executing a load, the prefetch strand encounters a
30 relatively long latency cache miss (such as a miss that led to the forking of the prefetch strand).
31 If so, then instead of blocking, the load delivers (in the context of the prefetch strand) an
32 ‘ambiguous’ placeholder value distinguished (e.g. by an ‘ambiguous bit’) from all other data
33 values delivered by loads (such as all data values that are obtainable via a cache hit). The
34 prefetch strand continues executing, using the ambiguous value for a result of the load. When a
35 uop has at least one input operand of the ambiguous value (sometimes referred to as the “uop
36 having an ambiguous input”), the uop propagates the ambiguous indication as a result for the

1 uop (sometimes referred to as “uop outputs an ambiguous value”). The microprocessor executes
2 a branch having an ambiguous input as if a predicted destination of the branch matches the
3 actual destination of the branch. When a prefetch strand executes a store, the prefetch strand
4 allocates a new cache line or temporary memory buffer element visible (e.g. observable and
5 controllable) only by the prefetch strand, to prevent the parent strand from observing the store.
6 In some embodiments, if a store writes an ambiguous value (e.g. into a cache), then the
7 destination of the store receives the ambiguous value (e.g. affected bytes in one or more cache
8 lines of the cache are marked as ambiguous). Subsequent loads of the destination receive the
9 ambiguous value, thus propagating the ambiguous value. In various usage scenarios, the
10 propagating of the ambiguous value enables avoiding prefetching unneeded data (e.g. when
11 loading a pointer) and/or avoiding what would otherwise be incorrectly or inefficiently updating
12 a branch predictor (e.g. when loading a branch condition).

13

14 **[0063]** In some embodiments, the microprocessor has logic to configure conditions and
15 thresholds for loads encountering cache misses to return an ambiguous result in lieu of stalling a
16 prefetch strand. For example, strandware configures the microprocessor to produce ambiguous
17 values only for cache misses resulting in a main memory access, and to stall for other cache
18 misses.

19

20 **[0064]** Prefetch strands, in various usage scenarios (such as integer and/or floating-
21 point code), make data available before use by a parent strand (reducing or eliminating cache
22 misses) and/or prime a branch predictor (reducing or eliminating mispredictions). Various
23 embodiments use prefetch strands instead of (or in addition to) hardware prefetching.

24

25 **[0065]** In some circumstances where a prefetch strand is forked from a parent strand,
26 the prefetch strand executes for several hundred cycles while the parent strand is waiting for a
27 cache miss (such as when this miss is satisfied from main memory that is implemented, e.g., as
28 DRAM). In some usage scenarios and/or embodiments, a system enables a prefetch strand to
29 make forward progress for a relatively significant portion of the time a parent strand is waiting.
30 For example, strandware constructs one or more traces for use in prefetch strands, and the traces
31 optionally exclude uops with certain properties. E.g., the strandware optionally excludes uops
32 that have no contribution to memory address generation. E.g., the strandware optionally
33 excludes uops only used to verify relatively easily predicted branches. E.g., with respect to a
34 trace within a particular prefetch strand, the strandware optionally excludes uops that store to
35 memory a value that is not read (or is relatively unlikely to be read) within the prefetch strand.
36 For yet another example, the strandware optionally excludes uops that load data that is already

1 present (or relatively likely to be present) in a cache before execution of the uop. E.g., the
2 strandware optionally excludes uops having properties that render the uops irrelevant to
3 prefetching.

4
5 **[0066]** In some embodiments and/or usage scenarios, the microprocessor attempts to
6 execute a prefetch strand relatively far ahead of a (waiting) parent strand, given available time.
7 For example, the strandware attempts to minimize (by eliminating or reducing) uops in a
8 prefetch trace, leaving only uops that are on one or more critical paths to execution of particular
9 loads. The particular loads are, e.g., loads that relatively frequently result in a cache miss, loads
10 that result in a cache miss with a relatively long latency to fill, or any combination thereof. In
11 some embodiments, the strandware, in conjunction with the hardware (such as cache miss
12 performance counters), collects and maintains profiling data structures used to determine the
13 particular loads, such as by collecting information about delinquent loads. When optimizing a
14 prefetch trace, the strandware optionally operates to reduce dataflow graphs that produce target
15 addresses of the particular loads.

18 SKIPAHEAD STRANDS

20 Skipahead Multithreading Model

21
22 **[0067]** A profiling subsystem of the strandware layer (such as Trace Profiling and Capture 120
23 of Fig. 1B), when executed by the microprocessor, identifies selected traces as candidates for
24 skipahead speculative multithreading. In some embodiments and/or usage scenarios, the system
25 uses skipahead strands for traces that have a relatively highly predictable terminal branch (such
26 as an unconditional branch, a loop instruction branch, or a branch that the system has predicted
27 relatively successfully). The system optionally selects candidates based on one or more
28 characteristics. An example characteristic is relatively low static Instruction Level Parallelism
29 (ILP), such as due to relatively many NOPs. Another example characteristic is a relatively low
30 dynamic ILP (such as having loads that relatively frequently stall, resulting in dynamic schedule
31 gaps that are relatively difficult to observe statically). Another example characteristic is a
32 potential for parallel issue that is greater than what a single core is capable of providing.

33
34 **[0068]** Skipahead speculative multithreading is effective in some usage scenarios having traces
35 that contain entire loop iterations and/or where there are relatively few dependencies between
36 loop iterations. Skipahead speculative multithreading is effective in some usage scenarios

1 having calls and returns that are not candidates for inline expansion into a single trace.
2 Skipahead speculative multithreading, in some usage scenarios and/or embodiments, yields
3 performance levels similar to an ROB-based out-of-order core (but with relatively less hardware
4 complexity). In some skipahead speculative multithreading circumstances, a successor strand
5 skips several hundred instructions ahead of a start of a trace. Performance improvements
6 effected by skipahead speculative multithreading (such as achieved by relatively high or
7 maximum overlap) depend, in some situations, on relatively accurate prediction of a start
8 address of a successor and data independence.

9
10 **[0069]** Fig. 2 illustrates an example of hardware executing a skipahead strand (such as
11 synthesized by strandware), plotted against time in cycles versus core or interconnect. In the
12 description, the term “skipahead strand” refers to execution (as a strand) of the target code (or a
13 binary translated version thereof), where the skipahead strand begins execution at the next
14 instruction (or binary translated equivalent) executed (in some circumstances) after the end of
15 the terminal trace of a parent strand. For each skipahead strand, a code generator of the
16 strandware layer (such as one or more elements of Scheduling and Optimization **160** of Fig. 1B
17 and/or Strand Construction **140** of Fig. 1C) inserts a fork.skip uop into the terminal trace of the
18 parent strand. The “terminal trace” of a strand refers to the final trace executed by the strand
19 before the strand reaches its join point. When the system executes the fork.skip uop, the system
20 forks a new (e.g. successor or child) strand as the skipahead strand. The skipahead strand begins
21 execution at the next instruction (or binary translated version thereof) executed in program order
22 after reaching the end of the trace containing the fork.skip uop. For terminal traces ending with
23 a conditional or indirect branch, in some embodiments, the skipahead strand starts at a
24 dynamically determined target of the branch. In some usage scenarios and/or embodiments, the
25 system selects the fork target dynamically via a trace predictor and/or branch predictor. In
26 scenarios where the terminal trace ends with an unconditional branch and/or the strandware ends
27 the trace in the middle of a basic block, the starting point of the skipahead strand is determined
28 when the terminal trace is generated.

29
30 **[0070]** In Fig. 2, fork.skip uop **211** in parent strand **200** has created a successor strand **201**,
31 illustrated in the right column executing as strand ID 22 on core 2. The successor strand starts
32 after some delay due to inter-core communication latency (illustrated as three cycles). The
33 successor strand then begins executing the trace corresponding to the fork target address.

34
35 **[0071]** The fork.skip uop encodes a propagate set (illustrated as propagated_archreg_set field
36 dashed-box element **212**) that specifies a bitmap of architectural registers to be written by the

terminal trace of the parent (other architectural registers are not modified by the trace). Execution of the successor strand stalls on the first read of an architectural register that is a member of the propagate set, unless the successor strand has previously written the register, so the successor will subsequently read its own private version of the register in lieu of the not yet propagated version of the parent.

[0072] With respect to the terminal trace of the parent strand, the uop format includes a mechanism to indicate that results of the uop are to propagate to the successor strand. In some embodiments, a VLIW bundle includes one or more “propagate” bits, each associated with one or more uops of the bundle. When the strandware schedules and optimizes a terminal trace for skipahead, the strandware sets the propagate bit of each uop if and only if the uop is the final uop (relative to the original program order of the uops of the trace) to write to a particular architectural register A, thus producing a live-out value. In some embodiments, the original program order is different from the execution order of a scheduled VLIW trace, and in other embodiments, the orders are identical.

[0073] When a uop targeting architectural register A executes and the propagate bit of the uop is set, the uop output value V is transmitted to the successor strand S (of the current strand). Conceptually, the value V is then written into the register file of strand S so that future attempts in S to read architectural register A receive the value V until a uop in strand S overwrites architectural register A with a new (locally produced) value. If successor strand S had been stalled while attempting to read live-in architectural register A, strand S is then unblocked to continue executing now that the value V has arrived. The parent strand, in some circumstances, propagates particular live-out architectural registers before the successor strand reads the registers. The particular registers are written into the register file of the successor strand (e.g. any of Register Files **194A.1-194A.4**) in the background and are not a source of stalls. The architectural registers that are not members of the propagate set are not be written by the terminal trace, and the successor strand thus inherits the values of the registers at the start of the terminal trace. The values are propagated in the background into the register file associated with the successor strand. The successor strand stalls if an inherited architectural register is not propagated before the successor strand accesses the register.

[0074] Fig. 2 illustrates an example of the propagation. After fork uop **211** creates successor strand **201**, the first three bundles **280**, **281**, and **282** of the first trace of the successor strand execute (respectively in cycles 3, 4, and 5), since the bundles are not dependent on any live-in registers (e.g. live-out registers from the parent strand terminal trace). However, when bundle

1 **283** attempts to execute during cycle 6, the bundle stalls, since the bundle is dependent on live-in
2 architectural registers `%rbx` and `%rbp` that the terminal trace of the parent strand has not yet
3 generated. In cycle 9, bundle **269** of the parent strand terminal trace computes the live-out
4 values of `%rbx` and `%rbp` via uops **215** and **216**, respectively, and propagates the values to the
5 successor strand. The values arrive at the core executing successor strand **201** several cycles
6 later (e.g. corresponding to inter-core communication latency), and in cycle 12, (successor
7 strand) trace **201** wakes up and executes bundles **284** and **285**. When the next bundle of the
8 successor strand attempts to read `%rdi`, a value is unavailable. The parent strand generates the
9 live-out value of `%rdi` in cycle 13 via uop **217** and propagates the value to successor strand **201**
10 for arrival in cycle 16. Then bundle **286** wakes up and executes in cycle 16. The figure
11 illustrates background propagation of some live-out architectural registers (such as `%rsp` and
12 `%xmmh0`, propagated by uops **213** and **214** respectively) before the registers are read by the
13 successor strand.

14
15 **[0075]** In some circumstances, the parent strand attempts to overwrite an architectural register
16 the successor strand is to inherit before a value for the register has been transmitted to the
17 successor strand. In some embodiments, interlock hardware prevents the parent from
18 overwriting an old value of a register until the old value is en route to the successor. In some
19 circumstances, the successor overwrites a live-in architectural register without reading the
20 register before the parent has propagated a corresponding live-out value to the successor. In
21 some embodiments, the successor notifies the parent that the successor is no longer waiting for
22 the propagated register value, since the successor has a more up-to-date (locally generated)
23 value.

24
25 **[0076]** Various mechanisms are used in various embodiments to propagate register values from
26 the parent strand to the successor strand. Some embodiments use different propagation
27 mechanisms and/or priorities for live-out propagated registers versus inherited registers. In
28 some embodiments, the register values are not copied. Instead, the successor strand uses a copy-
29 on-write register caching mechanism to retrieve inherited and live-out values from the parent
30 strand on-demand. The mechanism uses a copy-on-write function to prevent inherited values
31 from overwriting by the parent before communication to the successor, and to suppress
32 propagation when the successor no longer depends on a value. In some embodiments, a register
33 renaming mechanism is used to avoid copying actual values. The fork operation copies a
34 rename table of the parent strand to the successor strand (instead of copying values), and both
35 strands share one or more physical registers until one strand overwrites one or more of the
36 physical registers.

SPECULATIVE STRAND THREADING (SST)

SST Overview

[0077] The strandware partitions target software into a plurality of independently executable strands, to enable increased parallelism, performance, or both. The strandware and hardware operate collectively to dynamically profile target software to detect relatively large regions of control and data flow of the target software that have relatively few or no inter-dependencies between the regions. The strandware transforms each region into a strand by inserting a fork point at the start, and a join point / fork target at the end. Strands are program ordered with respect to each other, and execute independently.

[0078] In various embodiments, the hardware and strandware continue to monitor and refine the selection of fork and join points based on real-time feedback from observing and profiling dynamic control flow and data dependencies, enabling, in some usage scenarios, one or more of improved performance, improved adaptability, and improved /robustness.

Strand Scope Identification

[0079] In some speculative multithreading embodiments, a fork point produces two parallel strands: a new successor strand that starts executing at the fork target address in the target software and the existing parent strand that continues executing (in the target software) after the fork point. A trace predictor and/or branch predictor select the fork target dynamically.

[0080] After a fork, the scope (e.g. lifetime) of the parent strand includes all code executed after the fork operation until the execution path of the parent strand reaches the initial start address of the successor strand, or some other limits are reached. A strandware strand profiling subsystem derives the scope of each strand.

[0081] If the strandware identifies a loop for parallelization, both the fork point (where a fork operation is executed) and fork target (where the successor strand begins execution) refer to the top of the loop and branches that terminate the loop limit the scope of the parent. In a scenario of a conditional branch at the end of a loop (that jumps to the top of the loop for the next iteration), the terminating direction of the branch is not taken.

1

2 **[0082]** The strandware uses heuristics to identify terminating branches and directions
3 based on output of various compilers (such as GCC, ICC, Microsoft Visual Studio, Sun Studio,
4 PathScale Compiler Suite, and PGI). The compilers generate roughly equivalent control flow
5 idioms for a given instruction set (e.g. x86). For example, bounds of a loop are identified by
6 finding any taken branch that skips to the basic block immediately after the basic block(s) that
7 jump back to the top of the loop for the next iteration. Other terminating branches include return
8 instructions and unconditional branches to addresses after the last basic block in the loop body.

9

10 **[0083]** Consider call-return forks where the fork origin point is immediately before a
11 function call (e.g. prior to an x86 CALL instruction) and the target address is immediately after
12 the call instruction (i.e. at the return address). The scope of the parent strand is determined only
13 by the body of the function call, and is terminated by the intersection of the parent strand with
14 the return address. Dynamically, function calls relatively frequently return to the call site unless
15 the program executes erroneous code or an exception handler.

16

17 **[0084]** There are other relatively more generalized types of forks, such as when the fork
18 is performed before beginning a relatively large block of code and the fork target is after the end
19 of the block. Internal branches within the block (e.g. the scope of the parent strand) optionally
20 exit the block and branch into the successor scope. The strandware identifies and instruments
21 the internal branches as terminating branches. In various embodiments, various structured
22 programming cases (e.g. for loops, calls, and returns) are processed as part of a more generalized
23 control flow analysis technique.

24

25 **[0085]** In some embodiments, terminating branches are be found by executing a depth
26 first traversal through the basic blocks on the control flow graph, starting at the basic block
27 containing the fork origin and recursively following both taken and not-taken exits to every
28 branch. In usage scenarios, locating the terminating branches is complicated by a variety of
29 situations (e.g. branches not mapped into the address space, invalid or indeterminate branch
30 targets, and other situations giving rise to difficult to determine control flow changes).
31 However, the strandware preserves correctness of target software, even if the strandware does
32 not detect all terminating branches. Accommodating undetected terminal branches enables
33 strandware operation even when the strandware lacks any knowledge of high-level program
34 structure information (e.g. source code).

35

1 **[0086]** The strandware identifies and instruments traces containing each terminating
2 branch by injecting a conditional kill uop into the traces. Execution of the conditional kill uop
3 aborts all successor strands of the strand executing the kill uop if a condition specified by the kill
4 uop evaluates to true. Execution of an alternative type of conditional kill uop aborts the strand
5 executing the kill uop and all successor strands of same if the strand executing the kill uop is
6 speculative (see the section “Bridge Traces and Live-In Register Prediction” located elsewhere
7 herein).

8
9 **[0087]** If a terminating basic block ends with a branch uop, such as “br.cc R,R2”,
10 (where registers R1 and R2 are compared and the branch is taken only if comparison condition
11 cc is true), then the strandware injects a matching kill uop, such as “kill.cc R1,R2,T”. The kill
12 uop specifies cc, R1, and R2 that match the branch.

13
14 Nested Strands

15
16 **[0088]** To maintain fully deterministic execution of target software, in some
17 embodiments the strandware uses a strictly program ordered non-nested speculative
18 multithreading model, where a parent strand P has at most one successor strand S1 (with
19 optional recursion of S1 to a successor S2, and so forth). Some embodiments enable a strand to
20 have a plurality of successor prefetch strands (optionally in addition to a single non-prefetch
21 successor strand), since the prefetch strands make no modifications to architectural state.

22
23 **[0089]** In some programs, P encounters another fork point before joining S1. To
24 preserve deterministic behavior, the hardware suppresses any fork points in a parent strand when
25 a successor exists. To ensure that P does eventually join S1, the strandware uses heuristics and
26 hardware-implemented functions (e.g. timeouts) to detect and abort runaway strands, and then
27 re-analyze the target software for terminal branches to reduce or prevent future occurrences.

28
29 **[0090]** Each kill uop is marked with a strand scope identifier, so if a fork point for a
30 strand is suppressed, then any kill uops for the strand scope are also suppressed.

31
32 **[0091]** To perform recursive functions, each strand maintains a private fork nesting
33 counter (initialized to zero when the strand is created) that is incremented when a fork is
34 suppressed. When the hardware processes a kill uop, the kill uop only aborts a strand if the
35 nesting counter of the strand is zero, otherwise the nesting counter is decremented and the strand
36 is not aborted.

Candidate Strand Selection

[0092] In some usage scenarios, some loops are good candidates for speculative multithreading (with one or a plurality of iterations per strand). In some embodiments, the hardware includes profiling logic units and the strandware synthesizes instrumentation code (that interacts with the profiling logic units) for determining which loops are appropriate for breaking into parallel strands.

[0093] Each backward (looping) branch in target software has a unique target physical address P that the strandware uses for identification and profiling. The hardware filters out loops that are determined to be too small to optimize productively, by tracking total cycles and iterations and using strandware tunable thresholds for total cycles and iterations (e.g. the hardware filters out loops with less than 256 cycles per iteration). The hardware allocates a Loop Profile Counter (LPC), indexed by P, to relatively larger loops. The LPC holds total cycles, iterations, confidence estimators, and other information relevant to determining if the loop is a good candidate for optimization. The strandware periodically inspects the LPCs to identify strand candidates. The strandware manages LPCs. In various embodiments, one or more of the LPCs are cached in hardware and/or stored in memory.

[0094] Similar techniques are used in some embodiments for other types of candidate strands, such as called functions. For calls, a set of call profiling counters (CPCs) are optionally used to record various statistics, e.g. the number of cycles spent in the called function, which registers were modified, the most likely return values, and other information potentially useful in determining if the strand is a good candidate for optimization.

Strand Nesting Graph Construction

[0095] In some embodiments, the strandware dynamically constructs one or a more data structures representing relationships between regions of the target code as a strands or candidate strands known to the strandware. The strandware uses the structures to track nesting of strands inside each other. For example, for a plurality of nested loops (e.g. inner loops and outer loops), a strand having a function body optionally contains a nested function call (the function call containing a strand) or one or more loops. In some embodiments, the strandware represents nesting relationships as a tree or graph data structure.

1 **[0096]** In some embodiments, the strandware adds instrumentation code to translated
2 uops (such as maintained in a translation cache), to update the strand nesting data structures at
3 runtime as the translated uops are executed. In some embodiments, the hardware includes logic
4 to assist strandware with dynamic discovery of strand nesting relationships.

5
6 **[0097]** Based on strand nesting hierarchy as represented in the strand nesting data
7 structures, the strandware uses heuristics to select relatively more effective regions of code to
8 transform into strands, and the strandware instruments each selected strand for further profiling
9 as described below. In some embodiments, the heuristics include one or more techniques to
10 select an appropriate strand from nested inner and outer loops.

11 12 Instrumentation for Profiling

13
14 **[0098]** Based on the fork origin, the fork target, and the set of terminating branches and
15 respective directions, the strandware injects instrumentation into the uop-based translation of the
16 target software (e.g. as stored in a translation cache) to form a complete and properly scoped
17 strand. In some embodiments, the strandware injects a profiling fork into the trace or trace(s)
18 containing the basic block at the fork origin point. The profiling fork instructs the hardware to
19 create a profiling strand, such as described in the sections “Parent Strand Profiling” and
20 “Successor Strand Profiling” located elsewhere herein. The strandware identifies and
21 instruments the trace or trace(s) containing each terminating branch, such as described in section
22 “Strand Scope Identification” located elsewhere herein.

23 24 Parent Strand Profiling

25
26 **[0099]** After instrumentation for profiling, the next time the trace containing the fork
27 point is executed, the hardware creates a profiling strand as a successor strand of a parent strand.
28 The profiling strand blocks until the parent strand intersects with the starting address of the
29 profiling strand. Then the profiling strand begins executing, while the parent strand blocks.
30 When the profiling strand completes (e.g. via an intersection, a terminating branch, or another
31 fork), the parent unblocks and joins the profiling strand. The hardware invokes the strandware
32 to complete strand construction as described following.

33
34 **[0100]** After performing a profiling fork, the hardware enters a special profiling mode
35 to execute the remainder of the parent strand.

1 **[0101]** For each occurrence of certain events in the parent strand, the strandware
2 arranges for a Strand Execution Profiling Record (SEPR) to be written into a memory buffer
3 allocated by Strandware to hold SEPRs generated by the parent strand. In some preferred
4 embodiments, an SEPR is written whenever certain types of memory accesses (loads or stores)
5 are performed. In some embodiments, additional SEPRs are written to enable the strandware to
6 later reconstruct the exact code sequence executed by the strand, for instance by recording the
7 execution of basic blocks, traces, control flow changes, or similar data.

8
9 Successor Strand Profiling
10

11 **[0102]** A parent strand blocks when completed, while the successor (profiling) strand
12 executes and register and memory dependencies are identified. With respect to register
13 dependencies, as the successor strand executes, the hardware updates a per-strand bitmask when
14 the hardware first reads an architectural register, prior to the hardware writing over the register
15 in the successor strand. The bitmask represents the live-outs from the parent strand that are used
16 as live-ins for the successor strand.

17
18 **[0103]** With respect to memory dependencies, in some embodiments transactional
19 memory versioning systems enable speculation within the data cache. When a strand loads data,
20 the hardware makes a reservation on the memory location at cache line (or byte level)
21 granularity. The hardware tracks the reservations by updating a bitmap of which bytes (or
22 chunks of multiple bytes) speculative strands have loaded. The hardware optionally tracks
23 metadata, e.g. a list of which specific future strands have loaded a memory location. The
24 hardware stores the bitmap with the cache line and/or in a separate structure.

25
26 **[0104]** The data for the load comes from the latest of all strands that have written that
27 address earlier than the loading strand (in program order). In some circumstances, the earliest
28 strand is the architectural strand (e.g., when the line is clean). In some circumstances, the
29 earliest strand is a speculative strand (e.g. when the line is dirty) that is earlier than the loading
30 strand.

31
32 **[0105]** When a strand writes to a cache line, the hardware checks if any future strands
33 have reservations on the cache line. If so, then the hardware has detected a cross-strand alias,
34 and the hardware aborts the future strand and any later strands. Alternatively, the hardware
35 notifies the strandware of the cross-strand alias, to enable the strandware to implement a flexible
36 software defined policy for aborting strands.

1

2 [0106] Since the hardware serializes a profiling strand to begin execution after the
3 parent strand has completed, cross-strand aliasing does not occur; the hardware executes all
4 loads and stores in program order (with respect to the strand order, not necessarily the order of
5 uops within a strand), and therefore the reservation hardware is free for other purposes. While in
6 profiling mode, in some embodiments the system (e.g. any combination of the hardware and
7 strandware) uses the memory reservation hardware to analyze cross-strand memory forwarding.

8

9 [0107] The scope of a profiling strand is finite for a loop: the profiling ends when
10 execution reaches the top of the loop. Other types of forks, such as a call/return fork or a
11 generalized fork, have potentially unlimited scope, and hence the system uses heuristics to limit
12 the scope of the profiling strand. When the hardware detects that the profiling strand has
13 completed execution, the parent strand is unblocked and the strandware begins to execute a join
14 handler that constructs instrumentation needed for a fully speculative strand.

15

16 Dataflow Graph Construction via SEPR Processing

17

18 [0108] Using the program ordered SEPR data that the system previously collected, the
19 strandware builds up a data flow graph (DFG), starting with the live-outs of the parent as root
20 nodes.

21

22 [0109] As described elsewhere herein, while executing the parent strand, the hardware
23 maintains a list of program ordered SEPRs as a record of which traces and/or basic blocks the
24 hardware executed, as well as the cache tags and index metadata of relevant loads and stores.
25 Using the record, the strandware decodes each basic block in each executed trace into a stream
26 of program ordered uops. To construct the DFG, uop operands are converted into pointers to
27 earlier uops in program order, using a register renaming table.

28

29 [0110] To track memory dependencies, the strandware maintains a memory renaming
30 table that maps cache locations to the latest store operation to write to an address. Thus, loads
31 and stores selectively specify a previous store as a source operand. The strandware uses the
32 cache locations recorded in the SEPRs, with the memory renaming table, to include memory
33 dependencies in the DFG.

34

1 [0111] At the conclusion of the process, all uops executed in the parent strand have
2 been incorporated into a dataflow graph, with the root nodes (live outs) of the graph pointed to
3 by the current register renaming table and the memory renaming table.

4 5 Bridge Traces and Live-In Register Prediction 6

7 [0112] The live-in set of a speculative successor strand (e.g. final live-outs of the
8 parent) are predicted from the architectural register values that existed when the parent strand
9 forked. The strandware searches the dynamic DFG, depth first, from each live-out (both
10 registers and memory) to produce a subset of generating uops. The union of all the subsets, in
11 program order, is the live-out generating set.

12
13 [0113] The strandware creates a bridge trace that starts with the architectural register
14 and memory values at the fork point in the parent strand, and only includes the live-out
15 generating set used to predict final live-outs (as indicated by the live-in bitmask of the successor
16 speculative strand). The bridge trace also copies any live-out register predictions to a memory
17 buffer. Later the system uses the copies to detect mispredictions.

18
19 [0114] When a trace forks to a speculative strand, the strandware sets up the new strand
20 to begin execution at the bridge trace, rather than the first uop of the speculative strand. In
21 addition to handling register dependencies, the bridge trace converts any terminating branches
22 (and related uops that calculate the branch condition) into uops that abort the speculative strand.
23 Last, the bridge trace sets up various internal registers for the strand, such as pointers to the
24 predicted memory value list, deferral list, and an unconditional branch, to the start of the
25 speculative strand.

26 27 Bridge Trace Optimizations 28

29 [0115] Once the strandware has constructed the bridge trace, the strandware attempts to
30 reduce or minimize the length using various dynamic optimization techniques. Some idioms
31 such as spilling and filling registers or using many calls and returns in a strand sometimes result
32 in a register being repeatedly loaded and stored from the stack, without being changed.
33 Similarly, a stack pointer or other register is sometimes repeatedly incremented or decremented,
34 while in aggregate, the dependency chain is equivalent to the addition of a constant.

1 **[0116]** The strandware recognizes at least some of the idioms and patterns and
2 optimizes away the dependency chains into relatively few or fewer operations. For instance, the
3 strandware uses def-store-load-use short-circuiting, where a load reading data from a previous
4 store is speculatively replaced by the value of the store (the speculation is verified at the join
5 point along with the register and memory predictions).

6
7 **[0117]** If the strandware is unable to reduce the bridge trace to a predetermined or
8 programmable length, the strandware abandons the optimizing of the strand. The abandoning
9 occurs in various circumstances, such as when there are true cross-strand register dependencies,
10 or when a live-out is computed relatively late in the parent strand and consumed relatively early
11 in the successor strand (thus resulting in a relatively long dependency chain).

12
13 Memory Value Prediction
14

15 **[0118]** For some strands, the bridge trace predicts memory values. The strandware uses
16 the load reservation data collected during execution of the successor profiling strand (such as
17 described in section “Successor Strand Profiling” located elsewhere herein) to determine which
18 memory locations were written by the parent strand and subsequently read by the successor
19 profiling strand (sometimes referred to as cross-strand forwarding). In some embodiments, the
20 strandware directly accesses the hardware data cache tags and metadata to build a list of cache
21 locations that were forwarded across strands.

22
23 **[0119]** The strandware looks up each cache location affected by cross-strand
24 forwarding in the memory renaming table for the DFG. The table points to the most recent store
25 uop (in program order) to write to the location. Then the strandware builds the sub-graph of
26 uops necessary to generate the value of the store uop (e.g. using a depth first search). The
27 strandware includes uops into the bridge trace along with any other uops used to generate
28 register value predictions.

29
30 **[0120]** The store uop in a bridge trace decouples the store in the parent strand from
31 subsequent successor strands (the successor strands instead load the prediction from the bridge
32 trace). Last, the strandware copies information about each predicted store into a per-strand store
33 prediction validation list that is later compared with the actual store values to validate the
34 speculation. In various embodiments, the information includes one or more of the physical
35 address of the store, the value stored, and the mask of bytes written by the store (or alternatively,
36 the size in bytes and offset of the store).

1

2 Join Handler Trace

3

4 **[0121]** Each speculative strand constructed by the strandware has a matching bridge
5 trace and join handler trace. The join handler trace validates all register or memory value
6 predictions made by the bridge trace that were actually used (e.g., unused predictions are
7 ignored). Whenever a parent strand ends (such as via an intersection with the successor, a
8 terminating branch, or other event), the hardware redirects the successor strand to begin
9 executing the join handler defined for the parent strand.

10

11 **[0122]** For each register value prediction used, the join handler reads the predicted
12 value from the memory buffer (such as described in section “Bridge Traces and Live-In Register
13 Prediction” located elsewhere herein), and compares the predicted value with the live-out value
14 from the parent strand. The hardware includes “see through” register read and memory load
15 functions that enable a join trace to read state (e.g. registers and memory) of the join trace and
16 corresponding state of the parent strand for comparison. Some embodiments only compare
17 registers read by the successor strand.

18

19 **[0123]** Similarly, to validate memory value predictions, the join trace iterates through
20 the list of predicted stores that were used (in various embodiments, including one or more of a
21 physical address, value, and bytemask for each entry), and compares each predicted store value
22 with the locally produced live-out value of the parent strand at the same physical address. If the
23 system detects any mismatches, then the system aborts the successor strand and the parent strand
24 continues past the join point as if the system had not forked the successor.

25

26 **[0124]** If the join is successful, the system discards the parent strand and the successor
27 strand becomes the new architectural strand for the corresponding VCPU.

CONCLUSION

[0125] Certain choices have been made in the description merely for convenience in preparing the text and drawings and unless there is an indication to the contrary the choices should not be construed per se as conveying additional information regarding structure or operation of the embodiments described. Examples of the choices include: the particular organization or assignment of the designations used for the figure numbering and the particular organization or assignment of the element identifiers (i.e., the callouts or numerical designators) used to identify and reference the features and elements of the embodiments.

10

[0126] The words “includes” or “including” are specifically intended to be construed as abstractions describing logical sets of open-ended scope and are not meant to convey physical containment unless explicitly followed by the word “within.”

15 [0127] Although the foregoing embodiments have been described in some detail for purposes of clarity of description and understanding, the invention is not limited to the details provided. There are many embodiments of the invention. The disclosed embodiments are exemplary and not restrictive.

20 [0128] It will be understood that many variations in construction, arrangement, and use are possible consistent with the description, and are within the scope of the claims of the issued patent. For example, interconnect and function-unit bit-widths, clock speeds, and the type of technology used are variable according to various embodiments in each component block. The names given to interconnect and logic are merely exemplary, and should not be construed as
25 limiting the concepts described. The order and arrangement of flowchart and flow diagram process, action, and function elements are variable according to various embodiments. Also, unless specifically stated to the contrary, value ranges specified, maximum and minimum values used, or other particular specifications (such as ISA, number of cycles, and the number of entries or stages in registers and buffers), are merely those of the described embodiments, are expected
30 to track improvements and changes in implementation technology, and should not be construed as limitations.

[0129] Functionally equivalent techniques known in the art are employable instead of those described to implement various components, subsystems, functions, operations, routines, sub-routines, in-line routines, procedures, macros, or portions thereof. It is also understood that
35 many functional aspects of embodiments are realizable selectively in either hardware (i.e.,

generally dedicated circuitry) or software (i.e., via some manner of programmed controller or processor), as a function of embodiment dependent design constraints and technology trends of faster processing (facilitating migration of functions previously in hardware into software) and higher integration density (facilitating migration of functions previously in software into hardware). Specific variations in various embodiments include, but are not limited to: differences in partitioning; different form factors and configurations; use of different operating systems and other system software; use of different interface standards, network protocols, or communication links; and other variations to be expected when implementing the concepts described herein in accordance with the unique engineering and business constraints of a particular application.

[0130] The embodiments have been described with detail and environmental context well beyond that required for a minimal implementation of many aspects of the embodiments described. Those of ordinary skill in the art will recognize that some embodiments omit disclosed components or features without altering the basic cooperation among the remaining elements. It is thus understood that much of the details disclosed are not required to implement various aspects of the embodiments described. To the extent that the remaining elements are distinguishable from the prior art, components and features that are omitted are not limiting on the concepts described herein.

[0131] All such variations in design are insubstantial changes over the teachings conveyed by the described embodiments. It is also understood that the embodiments described herein have broad applicability to other applications, and are not limited to the particular application or industry of the described embodiments. The invention is thus to be construed as including all possible modifications and variations encompassed within the scope of the claims of the issued patent.

WHAT IS CLAIMED IS:

- 1 1. A system comprising:
2 strand construction means for dynamically constructing a dynamic-profiling-
3 directed strand-partitioned-thread-portion of at least one of one or more threads, wherein
4 the strand construction means is implemented at least in part via at least one of the one
5 or more threads, and wherein the strand-partitioned-thread-portion of each strand-
6 partitioned thread comprises a respective plurality of strand images;
7 execution means for strand-based processing of the one or more threads; and
8 wherein for each strand-partitioned thread processed, the execution means
9 enables simultaneous execution of an intra-thread plurality of strands corresponding to
10 two or more of the respective plurality of strand images of the strand-partitioned thread.
- 1 2. The system of claim 1, wherein for each strand-partitioned thread processed, the intra-thread
2 plurality of strands comprises an architectural strand of the strand-partitioned thread and
3 at least one or more successor strands of the strand-partitioned thread, the architectural
4 strand updates an architectural strand context comprising architectural state of the
5 strand-partitioned thread, each successor strand is younger than the architectural strand
6 of the strand-partitioned thread, and each successor strand updates a respective
7 successor strand context comprising a speculative version of the architectural state of the
8 strand-partitioned thread.
- 1 3. The system of claim 1, wherein the execution means further enables simultaneous execution
2 of an inter-thread plurality of strands comprising an architectural strand associated with
3 each strand-partitioned thread processed, each architectural strand updating respective
4 architectural strand context comprising respective architectural state.

- 1 4. The system of claim 3,
2 wherein for each strand-partitioned thread processed, the intra-thread plurality of strands
3 comprises the architectural strand of the strand-partitioned thread and at least
4 one or more successor strands of the strand-partitioned thread, each successor
5 strand is younger than the architectural strand of the strand-partitioned thread,
6 and each successor strand updates a respective successor strand context
7 comprising a speculative version of the architectural state of the strand-
8 partitioned thread; and
9 wherein the strand contexts are held in one or more dedicated context stores within a
10 microprocessor.
- 1 5. The system of claim 1, wherein each of the simultaneously executing intra-thread plurality of
2 strands is executed on respective cores of a microprocessor.
- 1 6. The system of claim 1, wherein each of the simultaneously executing intra-thread plurality of
2 strands is executed on respective function units of a microprocessor.
- 1 7. The system of claim 1, wherein at least parts of the strand construction means are
2 implemented using one or more of executable code and microcode of a microprocessor.
- 1 8. The system of claim 7, wherein at least portions of the one or more of executable code and
2 microcode are maintained in one or more non-volatile-storage devices.
- 1 9. The system of claim 1, further comprising:
2 memory comprising one or more DRAM devices; and
3 wherein the strand construction means is allocated portions of the memory in
4 support of the constructing of each strand-partitioned-thread-portion.
- 1 10. The system of claim 1, further comprising:
2 uop decoder logic enabled to decode at least one type of strand creation uop and
3 at least one type of strand destruction uop.

1 11. The system of claim 2, further comprising:
2 context storage dedicated to storing the strand contexts; and
3 strand join logic coupled to the context storage and enabled to perform, for at
4 least one processed strand-partitioned thread of the one or more threads, hardware-
5 assisted merging of a plurality of the strand contexts.

1 12. The system of claim 2, further comprising:
2 context storage dedicated to storing the strand contexts; and
3 strand fork logic coupled to the context storage and enabled to perform, for at
4 least one processed strand-partitioned thread of the one or more threads, hardware-
5 assisted copying of at least portions of the context of the architectural strand into
6 corresponding portions of the context of at least one of the successor strands.

1 13. The system of claim 2, further comprising:
2 a transactional memory comprising dedicated transactional memory storage and
3 dedicated transactional memory control logic, the transactional memory being enabled
4 to perform, for at least one processed strand-partitioned thread of the one or more
5 threads, hardware-assisted versioning of memory data, wherein multiple data versions
6 are maintained corresponding to each of a plurality of memory locations, wherein for
7 each of the plurality of memory locations a first version of the versions corresponds to
8 the architectural strand and at least at second version of the versions respectively
9 corresponds to at least one of the successor strands.

1 14. The system of claim 1, further comprising:
2 analysis means for identifying one or more latent dependencies corresponding to
3 respective cross strand operations occurring between the plurality of simultaneously
4 executing intra-thread strands and aliasing to one or more respective memory locations;
5 deferral means for removing the one or more latent dependencies via replacing
6 the respective cross strand operations with one or more respective deferred operations;
7 resolution means for evaluating each of the deferred operations performed by
8 the plurality of simultaneously executing intra-thread strands;
9 wherein the identifying and the replacing are enabled to operate dynamically
10 during the processing of each strand-partitioned thread processed; and
11 wherein with respect to execution of each strand-partitioned thread processed,
12 results realized from the processing via the plurality of simultaneously executing intra-
13 thread strands are identical to architecture-specified results for strictly sequential
14 processing.

1 15. A method comprising:
2 dynamically constructing a dynamic-profiling-directed strand-partitioned-
3 thread-portion of at least one of one or more threads, wherein the dynamically
4 constructing is implemented at least in part via at least one of the one or more threads,
5 wherein the strand-partitioned-thread-portion of each strand-partitioned thread
6 comprises a respective plurality of strand images;
7 strand-based processing the one or more threads; and
8 wherein for each strand-partitioned thread processed, simultaneously executing
9 an intra-thread plurality of strands corresponding to two or more of the respective
10 plurality of strand images of the strand-partitioned thread.

- 1 16. The method of claim 15, further comprising:
2 wherein for each strand-partitioned thread processed, the intra-thread plurality
3 of strands comprises an architectural strand of the strand-partitioned thread and at least
4 one or more successor strands of the strand-partitioned thread, each successor strand
5 being younger than the architectural strand of the strand-partitioned thread;
6 wherein for each strand-partitioned thread processed, the architectural strand
7 updates an architectural strand context comprising architectural state of the strand-
8 partitioned thread; and
9 wherein for each strand-partitioned thread processed, each successor strand
10 updates a respective successor strand context comprising a speculative version of the
11 architectural state of the strand-partitioned thread.
- 1 17. The method of claim 15, further comprising simultaneously executing an inter-thread
2 plurality of strands comprising an architectural strand associated with each strand-
3 partitioned thread processed, each architectural strand updating respective architectural
4 strand context comprising respective architectural state.
- 1 18. The method of claim 15, wherein the one or more threads comprise all or any portion of
2 applications executing in user mode and to an operating system kernel executing in
3 privileged mode.
- 1 19. The method of claim 15, wherein the one or more threads comprise all or any portion of a
2 virtual machine monitor and to one or more operating system kernels managed by the
3 virtual machine monitor.
- 1 20. The method of claim 15, wherein the one or more threads are in accordance with at least a
2 first instruction set architecture and the respective pluralities of strand images are in
3 accordance with a second instruction set architecture.
- 1 21. The method of claim 15, wherein the dynamically constructing is automatic and
2 unobservable to the one or more threads.
- 1 22. The method of claim 15, further comprising executing each of the simultaneously executing
2 intra-thread plurality of strands on respective cores of a microprocessor.

- 1 23. The method of claim 15, further comprising executing each of the simultaneously executing
2 intra-thread plurality of strands on respective functional units of a microprocessor.

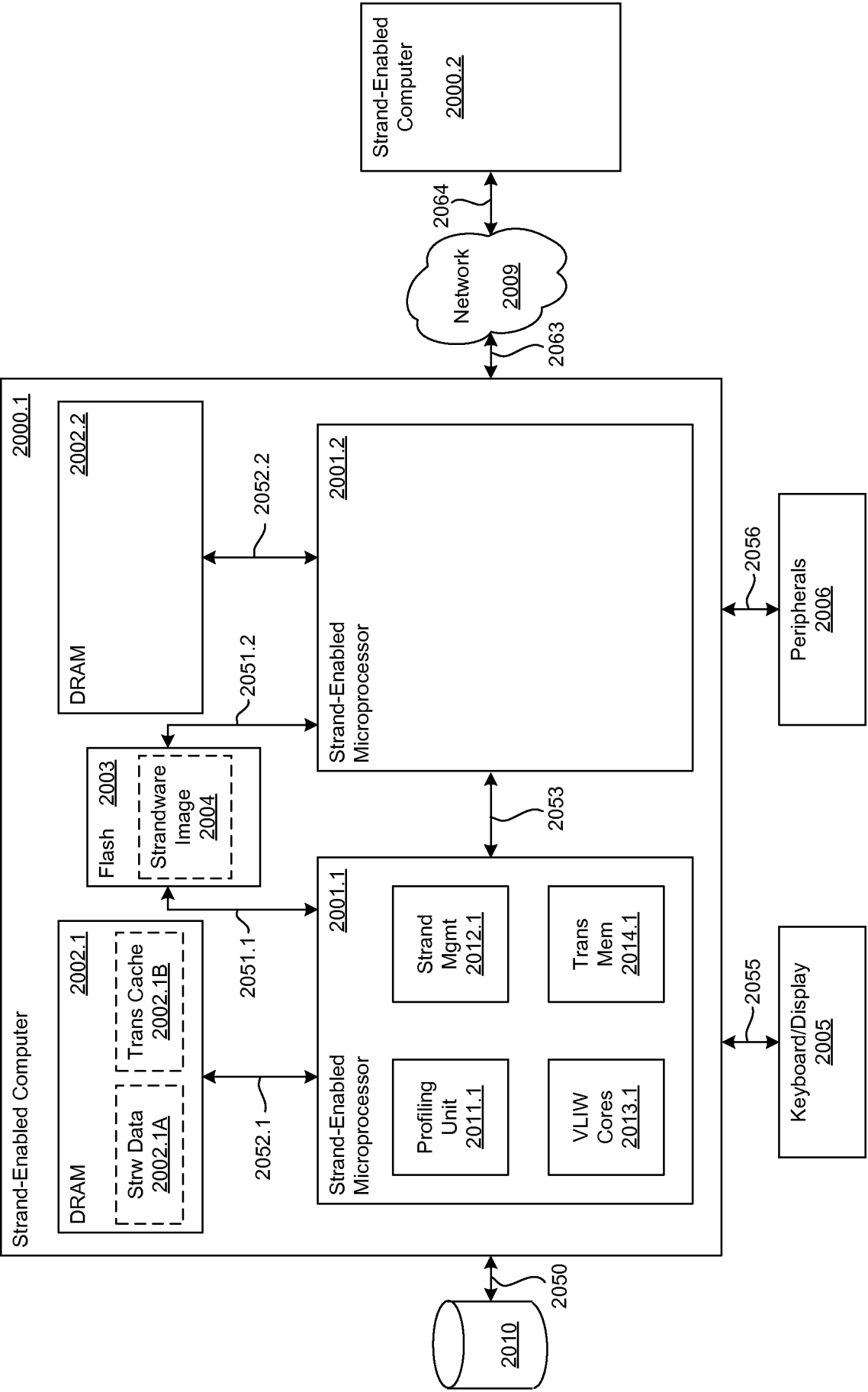


Fig. 1A

2/6

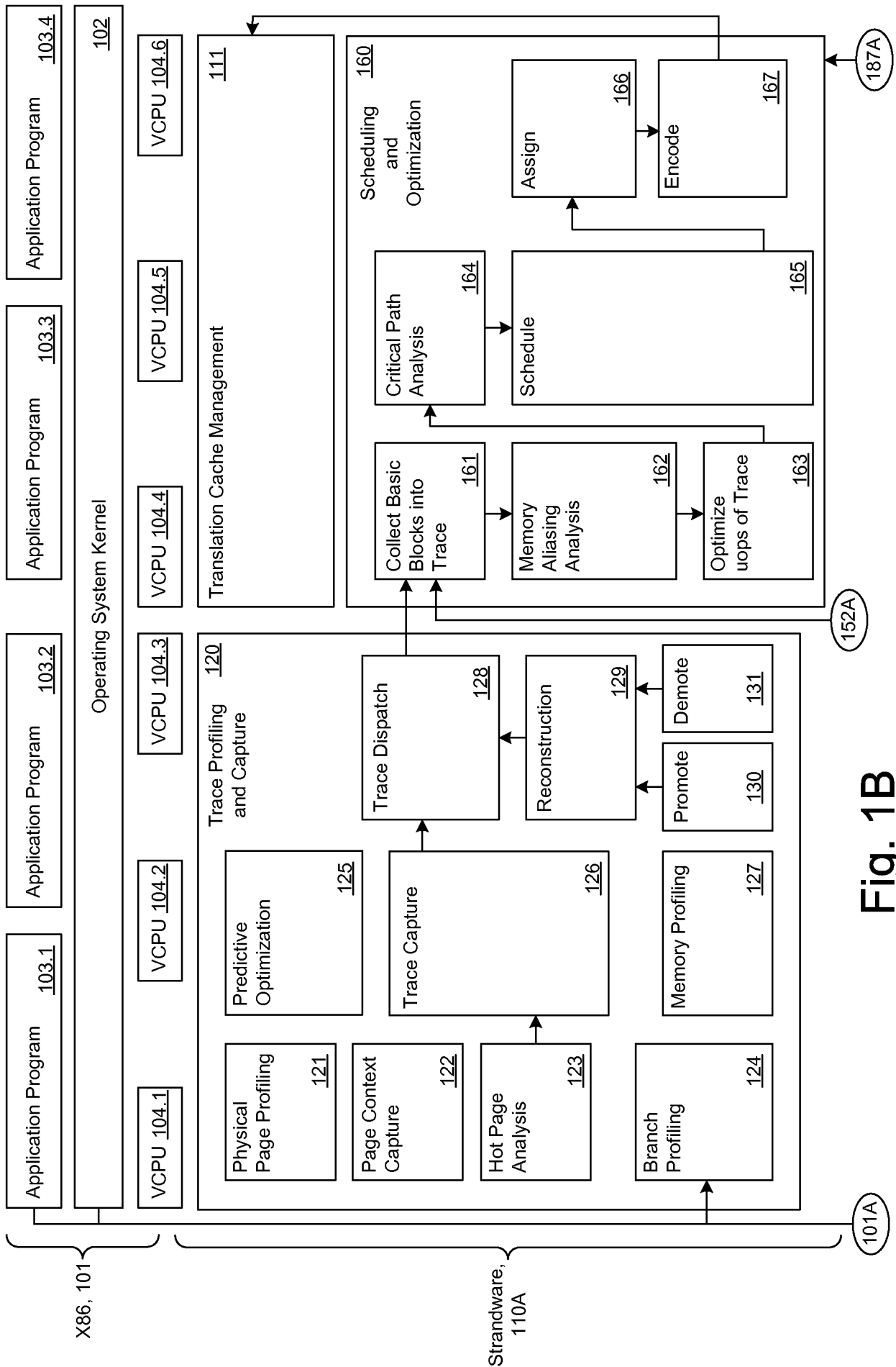


Fig. 1B

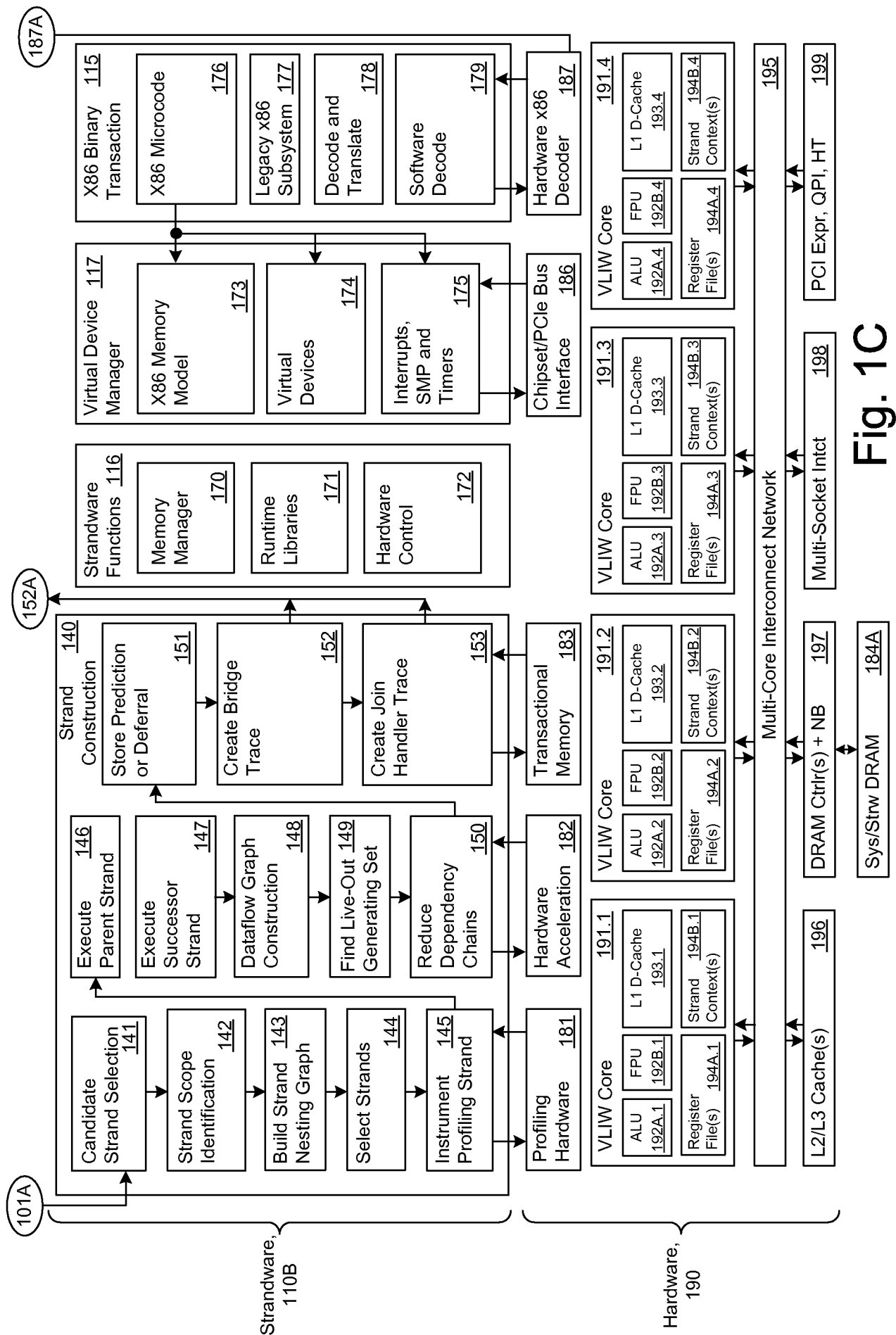


Fig. 1C

4/6

%reg [^] Propagate live out in parent strand's trace		%reg Receive live-in in successor strand's forked trace	
Cyc	Strand 21 (on core 1 thread 0)	Inter-core Transit	Strand 22 (on core 2 thread 0)
0	Strand 21 (on core 1 thread 0) 200 Id.p nop fork.skip 211 <imm32> bundle 0 bundle 1 bundle 2 bundle 3 bundle 4 bundle 5 bundle 6	202 rax rcx rdx rbp rsi rdi r8 r12 xmmi0 xmmh0i bundle 0 bundle 1 bundle 2 bundle 3 bundle 4 bundle 5 bundle 6	201 Stalled Stalled Stalled Start trace at fork target, 203 Ldd <imm32> Ldmfn.p <imm32> bundle 0 bundle 1 bundle 2 bundle 3
1	Ldd Ldd sub 213 mov 214 r5,m12 = [%rsp,120] r4,m5 = [%rsp,184] r1,%rsp [^] = [%rsp,88] r3,%xmmh0 [^] = %zero,%zero		
2	Id Id nop nop r2,m10 = [%rsp,104] r6,m3 = [%rsp,16]		
3	Id Id mulld nop r7,m1 = [%rsp,12] r8,%rdx [^] ,m13 = [%rsp,184] r9 = r4,r5	start strand background write into register file	r1,m10 = [%trace,0x45c460] 0x45c460 r2 = %zero, 89274+43 0x2b015cba bundle 0
4	Id Id nop nop r10,m8 = [%rsp,208] r11,m14 = [%rsp,192]	background write into register file	Id.p nop add movd bundle 1 bundle 2
5	st nop Ldmfn <imm32> Id adda brn.tk <brimm> m6 = [%rsp,12],r6 mfn0 = [90908,-112] 0x9001631c r12,m11 = [r10] r5 = r11,%zero,r8*8 r13,%rip [^] = mfn1+0x25a tid 581263 mfn1+0x25a tid 581263		Id nop br.subd.e <brimm> bundle 1 bundle 2 bundle 3
6		background write into register file	Missing %rbx Missing %rbp bundle 3

Fig. 2A

5/6

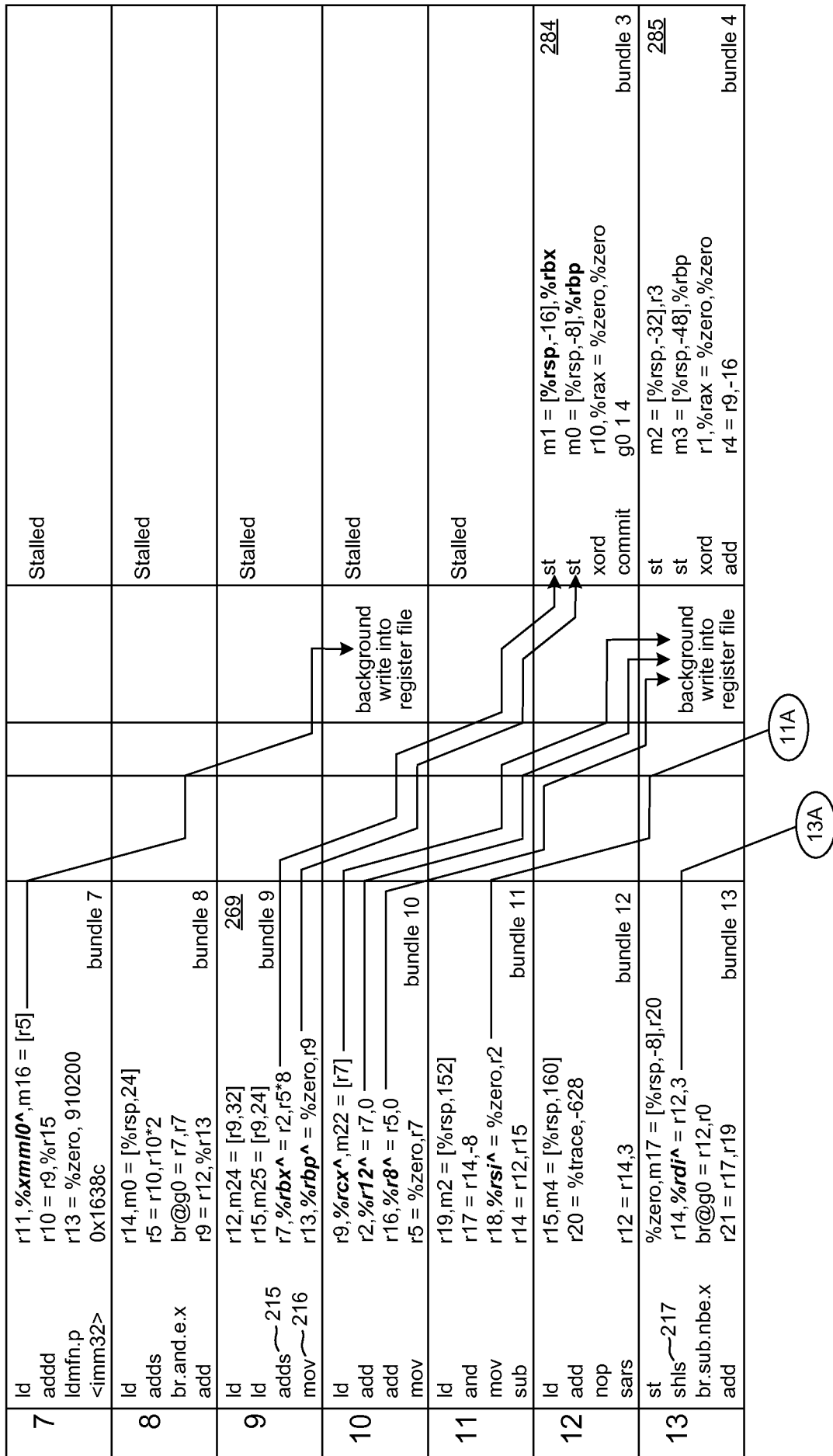


Fig. 2B

6/6

14	st st adds sub	m7 = [%r12,8],r21 m18 = [%rsp,-16],%r15 r17 = r6,r15*8 r19 = r12,r0	bundle 14	background write into register file	Stalled	Missing %rdi	bundle 5
15	ld st collec nop	r6,m23 = [r7,8] m9 = [%r12,16],r17 %zero,cc^ = r14,r14,r19	bundle 15		Stalled		
16	st st nop nop	m19 = [%rsp,-24],r2 m20 = [%rsp,-32],r10	bundle 16		ld sel.cmp.e sub movd	r3,%rbx,m9 = [r4] r8 = %rdi,r5,(%rdi) r0,%rsp = %rsp,56 r6,%rdx = %zero,1	286 bundle 5
17	st st nop sub	m21 = [%rsp,-40],%rbx m28 = [%rsp,-56],r11 r7 = r6,r9	bundle 17		ld <imm32> mov mov	r5,m6 = [%trace,0x4439e8] 0x4439e8 r5,%rbp = %zero,r8 r1,%rdi = %zero,r8	287 bundle 6
18	ld.p sars add <imm32>	r10,m15 = [%ctx,800] r6 = r7,3 r15 = %trace,0xff8fa8c 0xff8fa8c	bundle 18		st st br.and.e.x br.and.e.x	m5 = [%rsp,-40],%r12 m4 = [%rsp,-56],r8 br@g0 = r5,r5 br@g0 = r3,r3	288 bundle 7

End of trace, 204

Fig. 2C