Rock: A High-Performance Sparc CMT Processor

Rock, Sun's third-generation chip-multithreading processor, contains 16 high-performance cores, each of which can support two software threads. Rock uses a novel checkpoint-based architecture to support automatic hardware scouting under a load miss, speculative out-of-order retirement of instructions, and aggressive dynamic hardware parallelization of a sequential instruction stream. It is also the first processor to support transactional memory in hardware.

Shailender Chaudhry

Robert Cypher

Magnus Ekman

Martin Karlsson

Anders Landin

Sherman Yip

Håkan Zeffer

Marc Tremblay

Sun Microsystems

•••••• Designing an aggressive chip-multithreaded (CMT) processor[1] involves many tradeoffs. To maximize throughput performance, each processor core must be highly area and power efficient, so that many cores can coexist on a single die. Similarly, if the processor is to perform well on a wide spectrum of applications, per-thread performance must be high so that serial code also executes efficiently, minimizing performance problems related to Amdahl's law.[2,3] Rock, Sun's third-generation chip-multithreading processor, uses a novel resource-sharing hierarchical structure tuned to maximize performance per area and power.[4]

Rock uses an innovative checkpoint-based architecture to implement highly aggressive speculation techniques. Each core has support for two threads, each with one checkpoint. The hardware support for threads can be used two different ways: a single core can run two application threads, giving each thread hardware support for *execute ahead* (EA) under cache misses; or all of a core's resources can adaptively be combined to run one application thread with even more aggressive *simultaneous speculative threading* (SST), which uses two checkpoints. EA is an area-efficient way of creating a large virtual issue window without the large associative structures. SST dynamically extracts parallelism, letting execution proceed in parallel at two different points in the program.

These schemes let Rock maximize throughput when the parallelism is available, and focus the hardware resources to boost per-thread performance when it is at a premium. Because these speculation techniques make Rock less sensitive to cache misses, we have been able to reduce on-chip cache sizes and instead add more cores to further increase performance. The high core-to-cache ratio requires high off-chip bandwidth. Rock-based systems use large caches in the off-chip memory controllers to maintain a relatively low access rate to DRAM; this maximizes performance and optimizes total system power.

To help the programmability of multiprocessor systems with hundreds of threads and to improve synchronization-related performance, Rock supports transactional
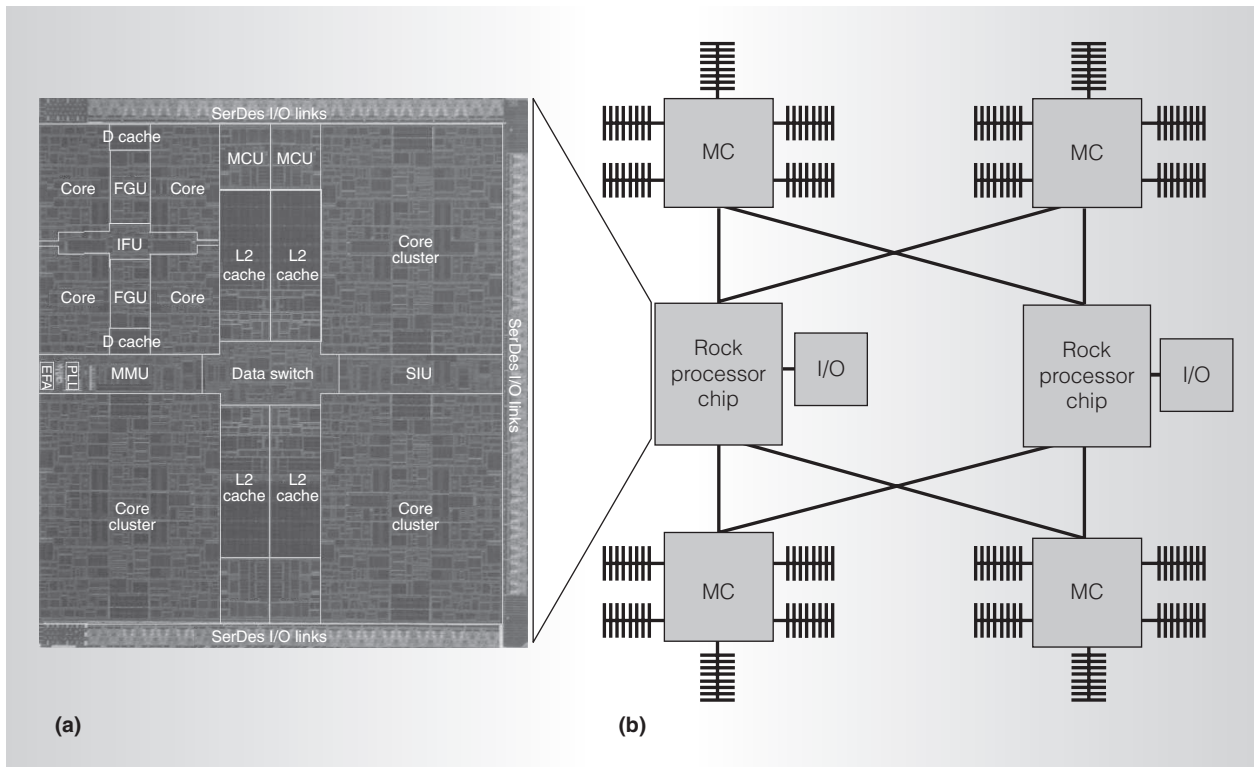
Figure 1. Die photo of the Rock chip (a) and logical overview of a two-processor-chip, 64-thread Rock-based system (b).

memory[5] in hardware; it is the first commercial processor to do so. TM lets a Rock thread efficiently execute all instructions in a transaction as an atomic unit, or not at all. Rock leverages the checkpointing mechanism with which it supports EA to implement the TM semantics with low additional hardware overhead.

System overview

Figure 1 shows a die photo of the Rock processor and a logical overview of a Rock-based system using two processor chips. Each Rock processor has 16 cores, each configurable to run one or two threads; thus, each chip can run up to 32 threads. The 16 cores are divided into core clusters, with four cores per cluster. This design enables resource sharing among cores within a cluster, thus reducing the area requirements. All cores in a cluster share an instruction fetch unit (IFU) that includes the level-one (L1) instruction cache. We decided that all four cores in a cluster should share the IFU because it is relatively simple to fetch a large number of instructions per cycle. Thus, four cores can share one IFU in a round-robin fashion while maintaining full fetch bandwidth. Furthermore, the shared instruction cache enables constructive sharing of common code, as is encountered in shared libraries and operating system routines.

Each core cluster contains two L1 data caches (D cache) and two floating-point units (FGU), each of which is shared by a pair of cores. As Figure 1a shows, these structures are relatively large, so sharing them provides significant area savings. A crossbar connects the four core clusters with the four level-two banks (L2 cache), the second-level memory-management unit (MMU), and the system interface unit (SIU). Rock's L2 cache is 2 Mbytes, consisting of four 512-Kbyte 8-way set-associative banks. A relatively large fraction of the die area is devoted to core clusters as opposed to L2 cache. We selected this design point after simulating designs with different ratios of core clusters to L2 cache. It provides the highest

throughput of the different options and yields good single-thread performance.

Each Rock chip is directly connected to an I/O bridge ASIC and to multiple memory controller ASICs via high-bandwidth serializer/deserializer (SerDes) links. This system topology provides truly uniform access to all memory locations. The memory controllers implement a directory-based MESI coherence protocol (which uses the states *modified, exclusive, shared,* and *invalid*) supporting multiprocessor configurations. We decided not to support the *owned* cache state (O) because there are no direct links between processors, and so accesses to a cache line in the O state in another processor would require a four-hop transaction, which would increase the latency and the link bandwidth requirements.

To further reduce link bandwidth requirements, Rock systems use hardware to compress packets sent over the links between the processors and the memory controllers. Each memory controller has an 8-Mbyte level-three cache, with all the L3 caches in the system forming a single large, globally shared cache. This L3 cache reduces both the latency of memory requests and the DRAM bandwidth requirements. Each memory controller has 4 + 1 fully buffered dual in-line memory module (FB-DIMM) memory channels, with the fifth of these used for redundancy.

Pipeline overview

A *strand* consists of the hardware resources needed to execute a software thread. Thus, in a Rock system, eight strands share a fetch unit, which uses a round-robin scheme to determine which strand gets access to the instruction cache each cycle. The 32-Kbyte shared instruction cache is four-way set associative, and virtual-to-physical translation is provided by a 64-entry, four-way-associative instruction translation look-aside buffer (ITLB). Because fetch logic in the IFU keeps track of when a fetch request can cross a page boundary, most fetch requests can bypass the ITLB access and thereby save ITLB power and fetch latency. Each strand has a sequential next-cache-line prefetch mechanism that automatically requests the next cache line, thereby exploiting the great spatial locality typical in instruction data. The fetch unit fetches a full cache line of 16 instructions and forwards it to a strand-specific fetch buffer.

Rock's branch predictor is a 60-Kbit (30K two-bit saturating counters) gshare branch predictor, enhanced to take advantage of the software prediction bit provided in Sparc V9 branches. In each cycle, two program-counter-relative (PC-relative) branches are predicted. Indirect branches are predicted by either a 128-entry branch-target buffer or a 128-entry return-address predictor.

As Figure 2 shows, the fetch stages are connected to the decode pipeline through an instruction buffer. The decode unit, which serves one strand per cycle, has a helper ROM that it uses to decompose complex instructions into sequences of simpler instructions. Once decoded, instructions proceed to the appropriate strand-specific instruction queue. A round-robin arbiter decides which strand will have access to the issue pipeline the next cycle. On every cycle, up to four instructions can be steered from the instruction queue to the issue first-in, first-out buffers (FIFOs). There are five issue FIFOs (labeled A0, A1, BR, FP, and MEM in Figure 2); each is three entries deep and directly connected to its respective execution pipe. An issue FIFO will stall an instruction if there is a read-after-write (RAW) hazard. However, such a stall does not necessarily affect instructions in the other issue FIFOs. Independent younger instructions in the other issue FIFOs can proceed to the execution unit out of order. Instructions that can't complete their execution go into the deferred instruction queue (DQ) for later reexecution. The DQ is an SRAM-based structure with only a single read/write port, so it is area and power efficient.

A two-level register file provides register operands. The first-level register file is a highly ported structure to support the source operand requirements of the maximum issue width.[6] It contains one register window, whereas the second-level register file contains all of the eight register windows that Rock supports. The second-level register file has a single read/write port and is made out of SRAM. For each architectural register in
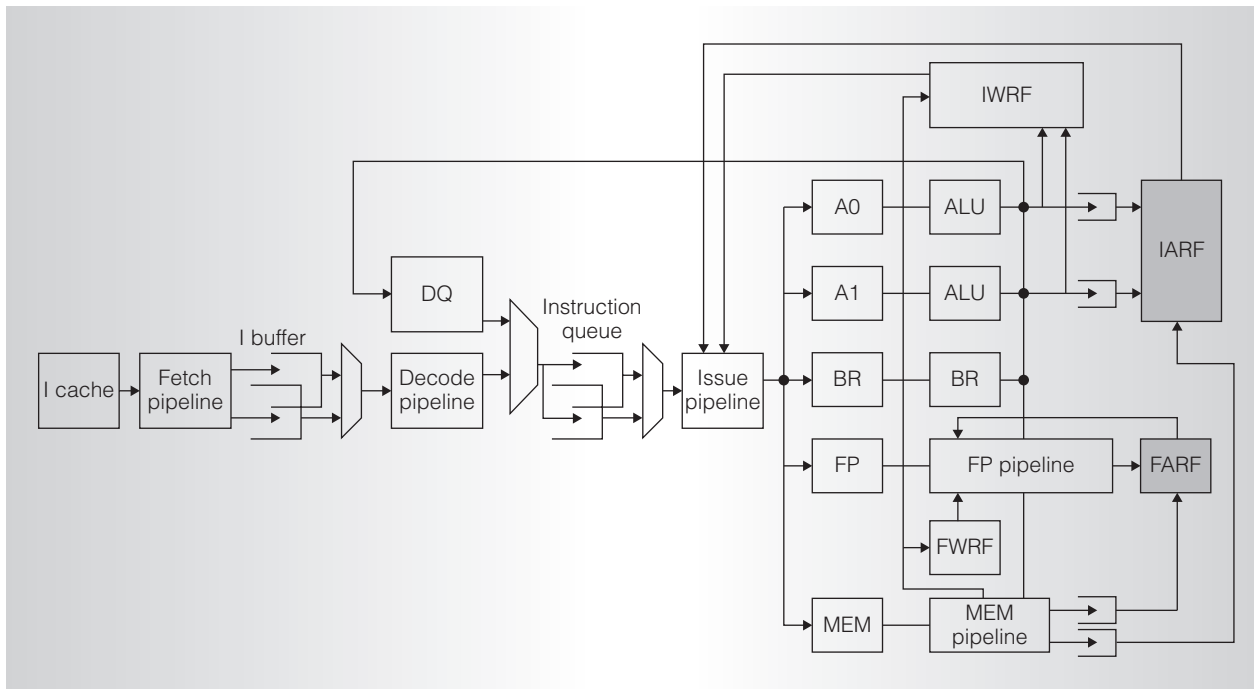
Figure 2. Pipeline overview showing five execution units (ALU, ALU, BR, FP pipeline, MEM pipeline), the integer architectural register file (IARF), the integer working register file (IWRF), the floating-point architectural register file (FARF), and the floating-point working register file (FWRF).

the second-level register file, there is both a main and a speculative copy to support checkpointing and speculation. Although the second-level register file holds about 16 times as much data as the first-level register file, because it is constructed of SRAM, the two structures are almost the same size. To keep track of which register copy to write to—main or speculative—each architectural register has a one-bit pointer associated with it.

The execution unit, which can execute up to five instructions per cycle, consists of five pipelines: simple ALU (A0), complex ALU (A1), branch (BR), floating-point (FP), and memory (MEM). The complex ALU provides all the functionality of the simple ALU and can also handle multicycle instructions such as leading-zero detect. The memory pipeline accepts one load or store instruction per cycle.

Two cores share each 32-Kbyte four-way set-associative data cache. The data cache is dual ported, so both cores sharing it can access it in each cycle. The two cores accessing the same data cache each have a private tag array, which enables a three-cycle load-use latency. The data cache is accompanied by a 32-entry, two-way skewed-associative way-predicted micro data translation look-aside buffer (micro-DTLB) with full least recently used (LRU) replacement. The way-predictor has a conflict avoidance mechanism that removes conflicts by duplicating entries. Stores are handled by a 32-entry store buffer in each core. The store buffer, which is two-way banked for timing reasons, supports store merging. Stores that hit in the data cache update both the data cache and the L2 cache. Stores that miss in the data cache update the L2 cache and do not allocate in the data cache. To increase memory-level parallelism, stores generate prefetch requests to the L2 and can update the L2 out of program order while maintaining the total store order (TSO)[7] memory model.

To conserve area, two cores share each floating-point unit in Rock. A floating-point unit contains a floating-point multiply-accumulate unit and a graphics and logical instructions unit that executes the Sparc VIS floating-point instructions. The unit

computes divides and square roots through the iterative Newton-Raphson method. Rock implements new instructions to allow register values to be moved between integer and floating-point registers. The floating-point unit further implements a new Sparc V9 instruction that produces random values with high statistical entropy.

In the pipeline's back end, the commit unit maintains architectural state and controls some of the checkpointed state required for speculation. The commit unit can retire up to four instructions per cycle while the pipeline is not speculating. An unlimited number of instructions can be committed upon successful completion of speculation.

Checkpoint-based architecture

Rock implements a checkpoint-based architecture,[8-10] in which a checkpoint of the architectural state can be taken on an arbitrary instruction. Once a strand takes a checkpoint, it operates in a speculative mode, in which each write to a register updates a "shadow" speculative copy of the register. If the speculation succeeds, the strand discards the checkpoint, and the speculative copies become the new architectural copies; we call this operation a *join*. If the speculation fails, the strand discards speculative copies of registers and resumes execution from the checkpoint. In addition to enabling EA, SST, and TM, the checkpoint mechanism simplifies trap processing and increases the issue rate by relaxing the issue rules.

Execute ahead

When a Rock system strand encounters a long-latency instruction, it takes a checkpoint and initiates an EA phase. A data cache miss, a micro-DTLB miss, and a divide are examples of long-latency events triggering an EA phase. The destination register of the long-latency instruction is marked as *not available* (NA) in a dedicated NA scoreboard, and the long-latency instruction goes into the DQ for later reexecution. For every subsequent instruction that has at least one source operand that is NA, its destination register is marked as NA and the instruction goes into the DQ. Instructions for which none of the operands is NA are executed, their results are written to

the speculative copy of the destination register, and the NA bit of the destination register is cleared. When the long-latency operation completes, the strand transitions from the EA phase to a replay phase, in which the deferred instructions are read out of the DQ and reexecuted.

In Figure 3a, the strand executes in a non-speculative phase until instruction 1 experiences a data cache miss, at which point the strand takes a checkpoint, defers instruction 1 to the DQ, and marks destination register r8 as NA. When a subsequent instruction that uses r8 as a source register (instruction 2) enters the pipeline, it is also deferred because of the NA bit for r8. As a result, its destination register, r10, is also marked as NA. An independent instruction (3) executes and writes its result into the speculative copy of the register. Because source register r10 for instruction 4 has been marked NA, that instruction also goes to the DQ.

When instruction 1's data is returned, the strand transitions from an EA to a replay phase. The instruction queue stops accepting instructions from the instruction buffer and starts accepting instructions from the DQ. Because instruction 1 is now a cache hit, r8 is no longer marked NA. The chain of dependent instructions (2 and 4) can also execute and write the results to the speculative copies of the registers. With all deferred instructions successfully reexecuted, the strand initiates a join operation to promote the speculative values as the new architectural state. The join operation merely involves manipulating the one-bit pointers that indicate which of the two register copies holds the architectural state values. It is thus a fast and power-efficient operation. The store buffer is notified that the EA phase has joined, and the buffered stores (if any) are allowed to start draining to the L2 cache. The join also notifies the instruction queue to resume accepting instructions from the instruction buffer, and the strand begins a nonspeculative phase.

In the EA process, only the long-latency instruction and its dependents are stored in the DQ; independent instructions update the speculative copies of their destination registers, and these instructions are speculatively retired. As a result, independent
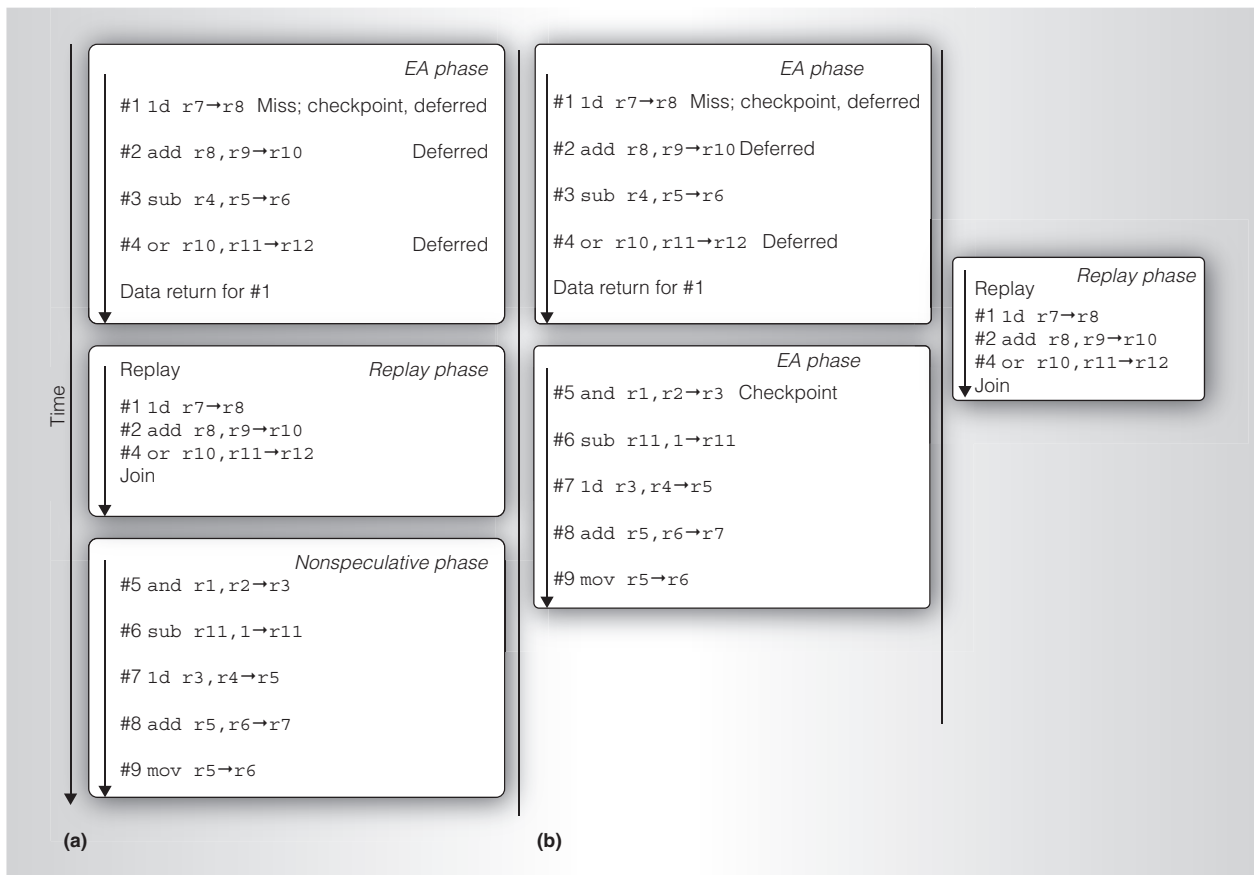
Figure 3. Examples of execute ahead (a) and simultaneous speculative threading (b). The nonspeculative phase in the EA example (a) is replaced in the SST example (b) by an EA phase that executes in parallel with the replay phase.

instructions require no additional architectural resources, so there is no strict architectural limit on the number of independent instructions executed. Therefore, with a relatively small DQ, an EA phase can have more than a thousand instructions in flight.[11]

In the example in Figure 3a, all instructions that went into the DQ can execute successfully in a single replay phase. However, if additional cache misses or other long-latency instructions arise in the EA phase, some instructions might have one or more NA source registers when they reexecute in the replay phase. In this case, these instructions will return to the DQ, and additional replay phases will run.

If an overflow of the store buffer or DQ occurs during an EA phase, the EA phase continues execution until the long-latency instruction completes. At this point, the EA phase fails and execution resumes from the checkpoint. In this case, the EA phase execution acts as a hardware scout thread that can encounter additional cache misses, thus providing memory-level parallelism.[12,13] In addition, the branch predictors are updated so as to reduce branch mispredictions when execution returns to the checkpoint. (Chaudhry et al. have described the performance gains from the hardware scout feature.[14])

Because the EA and replay phases execute instructions out of program order, they must address several register hazards. In particular, write-after-read hazards could occur in which an older instruction reexecuting in a replay phase reads a register that was written by a younger instruction in the preceding EA phase. To prevent such hazards (and similar hazards between successive replay phases), whenever an instruction goes into the DQ, all of the instruction's operands that are

available (not NA) go into the DQ with the instruction.

In addition, write-after-write register hazards could exist in which an older instruction that is reexecuted in a replay phase overwrites a register written by a younger instruction in the preceding EA phase. To prevent this type of hazard (and similar hazards between successive replay phases), older instructions are prevented from writing the speculative copy of a register that has already been written by a younger instruction (although all instructions are allowed to write the copy of their destination register in the first-level register file to provide data forwarding).

The EA and replay phases can speculatively execute loads out of program order. However, to support the TSO memory model, it is essential that no other thread can detect that the loads executed out of order. Thus, a bit per cache line per strand, called an *s-bit,* is added to the data cache. Every load in an EA or replay phase sets the appropriate s-bit. If the cache line containing a set s-bit is invalidated or evicted, the speculation fails and execution resumes from the checkpoint. Once an EA phase joins or fails, the strand's s-bits are cleared.

Simultaneous speculative threading

Rock's ability to dedicate the resources of two strands to a single software thread enables SST, a more aggressive form of speculation. In Figure 3a, when the long-latency instruction (1) completes, the strand stops fetching new instructions and instead replays instructions from the DQ. In contrast, with SST, two instruction queues are dedicated to the single software thread, so one instruction queue continues to receive new instructions from the fetch unit while the other replays instructions from the DQ. As a result, with SST it is possible to start a new EA phase in parallel with execution of the replay phase.

Figure 3b shows an example of SST: When the data returns for instruction 1, one strand begins a replay phase to complete the deferred instructions 1, 2, and 4. In parallel, the other strand takes an additional checkpoint and begins a new EA phase in which it executes new instructions 5 through 9.

In Figure 3b, the second EA phase completed successfully without deferring any of the instructions (5 through 9) to the DQ. However, if one or more of these instructions were deferred (for example, because of a cache miss within this EA phase or a dependence on a cache miss from the earlier EA phase), the strand that performed the first replay phase would then start a new replay phase to complete these deferred instructions. In this manner, it is possible to repeatedly initiate new EA phases with one strand, while performing replay phases for deferred instructions with the other strand.

In addition to enabling the parallel execution of an EA phase and a replay phase, SST reduces the EA phase failures due to resource limitations. Specifically, if the store queue or DQ is nearly full during a first EA phase, the core takes a second checkpoint and a new EA phase begins. In this manner, the first EA phase can complete successfully even if the store queue or DQ overflows during the second EA phase.

In terms of on-chip real estate, no extra state is added to support SST. The bookkeeping structures needed for two-thread EA are simply reused differently to allow for SST. Hence, SST requires virtually no extra area.

Transactional memory

To support TM, Rock uses two new instructions that have been added to the Sparc instruction set—*checkpoint* and *commit.* The checkpoint instruction denotes the beginning of a transaction, whereas commit denotes the end of the transaction. The checkpoint instruction has one parameter, called the *fail-pc.* If the transaction fails for any reason, the instructions within the transaction have no effect on the architectural state (with the exception of nontransactional stores, described later), and the program continues execution from the fail-pc. If the transaction succeeds, the architectural state includes the effects of all instructions within the transaction, the loads and stores within the transaction are atomic within the logical memory order, and execution continues after the commit instruction.

Rock's implementation of TM relies heavily on the same mechanisms that enable

EA operation. In particular, a strand views the checkpoint instruction as a long-latency instruction (analogous to a load miss) that checkpoints the register state and the fail-pc. If the transaction fails, the strand discards the speculative register updates performed within the transaction, restores the checkpointed state, and continues execution from the fail-pc. If the transaction succeeds, the speculative register updates are committed to architectural state, and the checkpoint is discarded.

Loads within the transaction are executed speculatively and thus set s-bits on the cache lines that are read. The invalidation or replacement of a line with the thread's s-bit set prior to committing the transaction will fail the transaction. As a result, the same mechanism that allows reordering of speculative loads during EA lets the strand make transactional loads appear atomic. Stores within the transaction are placed in the store queue in program order. The addresses of stores are sent to the L2 cache, which then tracks conflicts with loads or stores from other threads. If the L2 cache detects such a conflict, it reports the conflict to the core, which then fails the transaction. When the commit instruction executes, the L2 locks all lines being written by the transaction. Locked lines cannot be read or written by any other threads. This is the point at which other threads view the transaction's loads and stores as being performed, thus guaranteeing atomicity. The stores then drain from the store queue and update the lines, with the last store to each line releasing the lock on that line. The support for locking lines stored to by a committed transaction is the primary hardware mechanism that we added to Rock to implement TM.

Rock's TM support primarily targets efficient execution of moderately sized transactions that fit within the hardware resources. As an example, all stores in a transaction are buffered in the store queue, so the store queue size places a limit on the size of a successful transaction.

Along with the checkpoint and commit instructions, Rock provides additional registers and instructions to aid in debugging and to provide greater control over a transaction's execution. Specifically, Rock provides
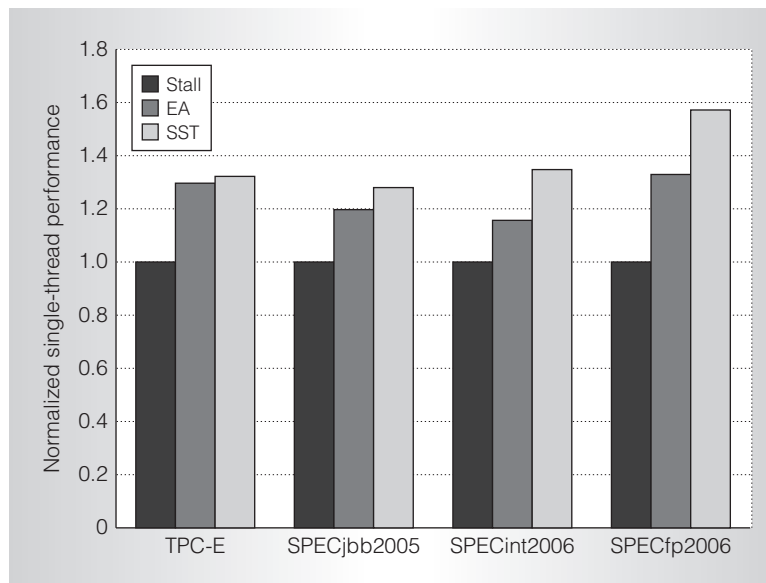

Figure 4. Single-thread performance improvements over stalling in-order processor achieved by speculative techniques execute ahead (EA) and simultaneous speculative threading (SST).

a checkpoint status register that can be read at the fail-pc to determine the cause of a failed transaction. It also provides nontransactional loads and stores.

Nontransactional loads are logically not part of the transaction; they do not cause failure when another thread stores to the location that was read nontransactionally. As a result, nontransactional loads can be used to synchronize the committing of a transaction. For example, it is possible to use nontransactional loads to repeatedly read a *flag* variable until it has been set by another thread.

Nontransactional stores, similarly, are logically not part of the transaction; they update memory even if the transaction fails. Therefore, it is possible to use nontransactional stores to determine what fraction of a transaction executed prior to a failure, thus aiding debugging.

Performance results

Figure 4 shows simulation results for how the speculation techniques we've described improve single-thread performance. We validated the simulator against hardware. The results—for the benchmarks TPC-E, SPECjbb, SPECint2006, and SPECfp2006—are normalized to a stalling
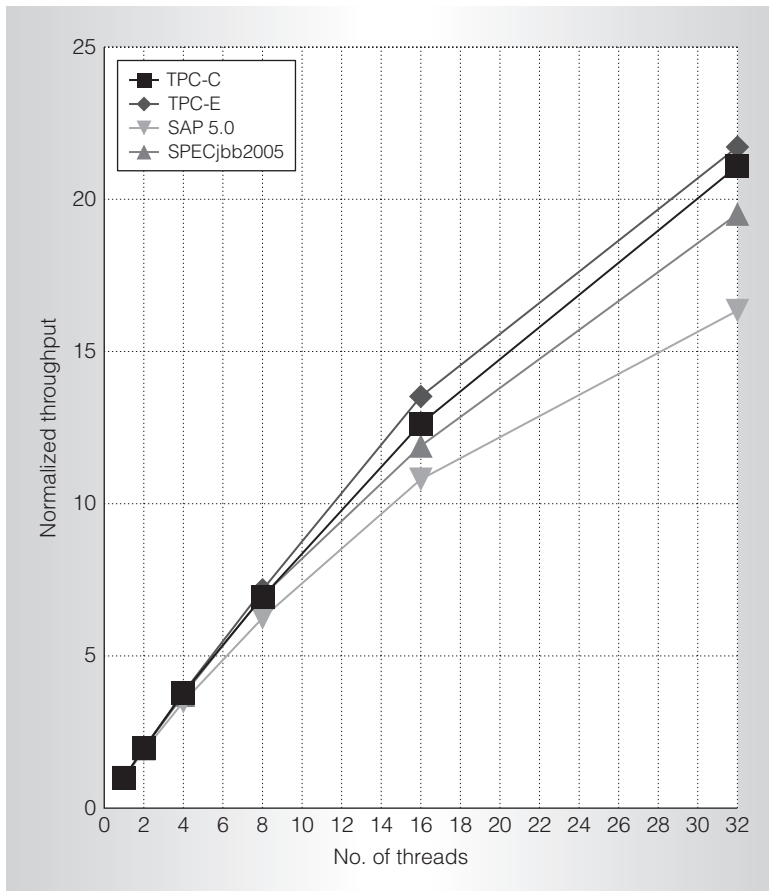
Figure 5. Rock chip throughput for various thread counts, performance normalized to that of a single EA thread.

In Figure 4, the rightmost bar for each benchmark indicates the performance gain from SST. The main advantage of this technique is that a new EA phase can execute in parallel with the replay phase. As with EA, the gains from SST are most pronounced in the low-miss-rate benchmarks.

Per-thread performance also depends on how heavily the chip resources are shared with other threads. The miss rate increases as more threads share caches, and as a result Rock's speculation techniques become more important. To obtain the results shown in Figure 5, we placed the threads on the chip so as to minimize interference between threads. Four or fewer active threads run in different core clusters, and the only shared resource is the L2 cache. When eight threads are active, threads also share an instruction cache and an instruction fetch unit. When 16 threads are active, all cores are used, and hence threads also share a data cache. When we enabled all 32 threads, two threads share each core.

Figure 5 shows the throughput at various thread counts, with performance normalized to that of a single EA thread (with no other threads running on the chip). Despite the sharing of multiple resources, the chip throughput increases up to 22× as the thread count increases from 1 to 32. Thus, the per-thread performance remains high.

R ock is the first commercial processor to use a checkpoint-based architecture to enable speculative, out-of-order retirement of instructions, and the first commercial processor to implement hardware TM. These novel features raised many unique challenges during the processor's design and verification.

To accommodate Rock's checkpointing and speculation, we made several changes to the Sparc verification infrastructure. We verified the processor's operation using a Sparc functional simulator as a reference model. Unlike other processors, Rock speculatively retires instructions out of order, and then either performs a join or fails speculation. As a result, the intermediate results produced by the speculatively retired instructions could not be directly verified by single-stepping the reference model. Instead, we

in-order processor. All of the benchmarks gain significant performance from EA, ranging from 18 percent for SPECint to 35 percent for SPECfp. The two commercial workloads fall in between, with improvements of 20 and 30 percent.

The performance gains due to EA come from the combination of two effects. One is the increased memory-level parallelism that comes from uncovering more cache misses under the original miss. This effect is most pronounced in the two commercial benchmarks (TPC-E and SPECjbb) because of their large working sets and resulting high miss rate in the on-chip caches. The other effect is the overlapping of cache misses with the execution of independent instructions, which is most important in SPECint and SPECfp because of their relatively low miss rate in the shared L2 cache.

had to step the reference model tens or hundreds of times to capture the state after a join, and only at this point could we compare the architectural state to the reference model.

The aggressive speculative design complicated not only correctness debugging, but also performance debugging, because slight changes to the implementation could cause speculation to succeed or fail, thus significantly impacting performance. As a result, accurately modeling the speculation failure conditions in the performance model was essential. Furthermore, failed speculation also complicated correctness debugging by hiding the effects of rare, corner-case bugs that appeared in EA phases, which often failed.

Finally, to verify the TM implementation, we implemented a logical reference memory model that tracked the values of loads and stores from multiple threads. We used this logical reference model to verify the atomicity of the loads and stores within a transaction by applying all of them to the reference model when the transaction logically commits. The greatest challenge was identifying precisely when the transaction logically commits with respect to loads and stores from other threads.

With its novel organization of on-chip resources and key speculation mechanisms, Rock delivers leading-edge throughput performance in combination with high per-thread performance, with a good performance/power ratio. Its hardware implementation of TM will help programmability of multiprocessor systems and improve synchronization-related performance. MICRO

Acknowledgments

It is an honor to present this work on behalf of the talented Rock team. Rock would not have been possible without their devotion and hard work.

....
References
 1. P. Kongetira, K. Aingaran, and K. Olukotun, ''Niagara: A 32-Way Multithreaded Sparc Processor,'' *IEEE Micro,* vol. 25, no. 2, 2005, pp. 21-29.
 2. G.M. Amdahl, ''Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,'' *Proc. AFIPS Conf.,* vol. 30, AFIPS Press, 1967, pp. 483-485.
 3. M.D. Hill and M.R. Marty, ''Amdahl's Law in the Multicore Era,'' *Computer,* vol. 41, no. 7, 2008, pp. 33-38.
 4. M. Tremblay and S. Chaudhry, ''A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread SPARC Processor,'' *Proc. Int'l Solid-State Circuits Conf. Digest of Technical Papers* (ISSCC 08), IEEE Press, 2008, pp. 82-83.
 5. M. Herlihy and J.E.B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. IEEE/ACM Int'l Symp. Computer Architecture* (ISCA 93), ACM Press, 1993, pp. 289-300.
 6. M. Tremblay, B. Joy, and K. Shin, ''A Three Dimensional Register File for Superscalar Processors,'' *Proc. Hawaii Int'l Conf. System Sciences* (HICSS 95), IEEE CS Press, 1995, pp. 191-201.
 7. SPARC Int'l, *The SPARC Architecture Manual (version 9),* Prentice-Hall, 1994.
 8. H. Akkary, R. Rajwar, and S.T. Srinivasan, ''Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,'' *Proc. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 03), IEEE CS Press, 2003, pp. 423-434.
 9. W.W. Hwu and Y.N. Patt, ''Checkpoint Repair for Out-of-Order Execution Machines,'' *Proc. IEEE/ACM Int'l Symp. Computer Architecture* (ISCA 87), ACM Press, 1987, pp. 18-26.
10. S.T. Srinivasan et al., ''Continual Flow Pipelines: Achieving Resource-Efficient Latency Tolerance,'' *IEEE Micro,* vol. 24, no. 6, 2004, pp. 62-73.
11. A. Cristal et al., ''Out-of-Order Commit Processors,'' *Proc. IEEE Int'l Symp. High-Performance Computer Architectures* (HPCA 04), IEEE CS Press, 2004, pp. 48-59.
12. J. Dundas and T. Mudge, ''Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss,'' *Proc. Int'l Conf. Supercomputing* (ICS 97), ACM Press, 1997, pp. 68-75.
13. O. Mutlu et al., ''Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors,'' *Proc. IEEE Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
14. S. Chaudhry et al., ''High-Performance Throughput Computing,'' *IEEE Micro,* vol. 25, no. 3, 2005, pp. 32-45.

Shailender Chaudhry is a distinguished engineer at Sun Microsystems, where he is chief architect of the Rock processor. His research interests include processor architecture, transactional memory, algorithms, and protocols. Chaudhry has an MS in computer engineering from Syracuse University.

Robert Cypher is a distinguished engineer and a chief system architect at Sun Microsystems. His research interests include parallel computer architecture, processor architecture, transactional memory, and parallel programming. He has a PhD in computer science from the University of Washington.

Magnus Ekman is a staff engineer at Sun Microsystems, where he is a performance architect. His research interests include processor architecture and memory system design. Ekman has a PhD in computer engineering from Chalmers University of Technology. He also has an MS in financial economics from Gothenburg University.

Martin Karlsson is a staff engineer at Sun Microsystems, where he is a core architect working closely with the design team on Rock's microarchitecture. His research interests include transactional memory, instruction scheduling, and checkpoint-based

architectures. He has a PhD in computer science from Uppsala University.

Anders Landin is a distinguished engineer and a chief system architect at Sun Microsystems. His research interests include multiprocessor system architecture, processor architecture, and application and system performance modeling. He has an MS in computer science and engineering from Lund University, Sweden.

Sherman Yip is a staff engineer at Sun Microsystems, where he is a core architect working closely with the design team on Rock's microarchitecture. His research interests include evaluating new processor features and efficient implementation of architectures that use deep speculation. Yip has a BS in computer science from the University of California at Davis.

Håkan Zeffer is a staff engineer at Sun Microsystems, where he is a system and performance architect. His research interests include parallel computer architecture, coherence, memory consistency, and system software. He has a PhD in computer science from Uppsala University.

Marc Tremblay is a Sun Fellow, senior vice president, and chief technology officer for Sun's Microelectronics Group. In this role, Tremblay sets future directions for Sun's processor and system roadmap. His mission has been to move the entire Sparc processor product line to the throughput computing paradigm, incorporating techniques he has helped develop over the years—including multicores, multithreading, speculative multithreading, scout threading, and transactional memory.

Direct questions and comments about this article to Martin Karlsson, Sun Microsystems, 4180 Network Circle, Mailstop SCA18-211, Santa Clara, CA 95054; martin.karlsson@sun.com.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.

IEEE Annals
of the History of Computing

IEEE Annals of the History of Computing **is an active center for the collection and dissemination of information on historical projects and organizations, oral history activities, and international conferences.**

www.computer.org/annals