Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor

Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay
Sun Microsystems, Inc.
4180 Network Circle, Mailstop SCA18-211
Santa Clara, CA 95054, USA
{shailender.chaudhry, robert.cypher, magnus.ekman, martin.karlsson, anders.landin, sherman.yip, haakan.zeffer, marc.tremblay}@sun.com

ABSTRACT

This paper presents Simultaneous Speculative Threading (SST), which is a technique for creating high-performance area- and power-efficient cores for chip multiprocessors. SST hardware dynamically extracts two threads of execution from a single sequential program (one consisting of a load miss and its dependents, and the other consisting of the instructions that are independent of the load miss) and executes them in parallel. SST uses an efficient checkpointing mechanism to eliminate the need for complex and power-inefficient structures such as register renaming logic, reorder buffers, memory disambiguation buffers, and large issue windows. Simulations of certain SST implementations show 18% better per-thread performance on commercial benchmarks than larger and higher-powered out-of-order cores. Sun Microsystems' ROCK processor, which is the first processor to use SST cores, has been implemented and is scheduled to be commercially available in 2009.

Categories and Subject Descriptors

C.1.0 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES—*General*; C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS—*Design studies*

General Terms

Design, Performance

Keywords

CMP, chip multiprocessor, processor architecture, instruction-level parallelism, memory-level parallelism, hardware speculation, checkpoint-based architecture, SST

1. INTRODUCTION

Recent years have seen a rapid adoption of Chip Multiprocessors (CMPs) [3, 12, 18, 24], thus enabling dramatic increases in throughput performance. The number of cores per chip, and hence the throughput of the chip, is largely limited by the area and power of each core. As a result, in order to maximize throughput, cores for CMPs should be area- and power-efficient. One approach has been the use of simple, in-order designs. However, this approach achieves throughput gains at the expense of per-thread performance. Another approach consists of using complex out-of-order (OoO) cores to provide good per-thread performance, but the area and power consumed by such cores [19] limits the throughput that can be achieved.

This paper introduces a new core architecture for CMPs, called Simultaneous Speculative Threading (SST), which outperforms an OoO core on commercial workloads while eliminating the need for power- and area-consuming structures such as register renaming logic, reorder buffers, memory disambiguation buffers [8, 11], and large issue windows. Furthermore, due to its greater ability to tolerate cache misses and to extract memory-level parallelism (MLP), an aggressive SST core outperforms a traditional OoO core on integer applications in a CMP with a large core-to-cache ratio.

SST uses a pipeline similar to a traditional multithreaded processor with an additional mechanism to checkpoint the register file. SST implements two hardware threads to execute a single program thread simultaneously at two different points: an ahead thread which speculatively executes under a cache miss and speculatively retires instructions out of order, and a behind thread which executes instructions dependent on the cache miss. In addition to being an efficient mechanism for achieving high per-thread performance, SST's threading logic lends itself to supporting multithreading where application parallelism is abundant, and SST's checkpointing logic lends itself to supporting hardware transactional memory [10].

Sun Microsystems' ROCK processor [5, 25] is a CMP which supports SST, multithreading, and hardware transactional memory. It is scheduled to be commercially available in 2009. ROCK implements 16 cores in a 65nm process, where each core (including its share of the L1 caches and the fetch and floating point units) occupies approximately 14 mm^2 and consumes approximately 10W at 2.3 GHz and 1.2V. By way of comparison, we know of no other commercial general-purpose processor in a 65nm process with more than 8 cores.

Copyright 2009 SUN Microsystems
ISCA'09, June 20–24, 2009, Austin, Texas, USA.
ACM 978-1-60558-526-0/09/06.

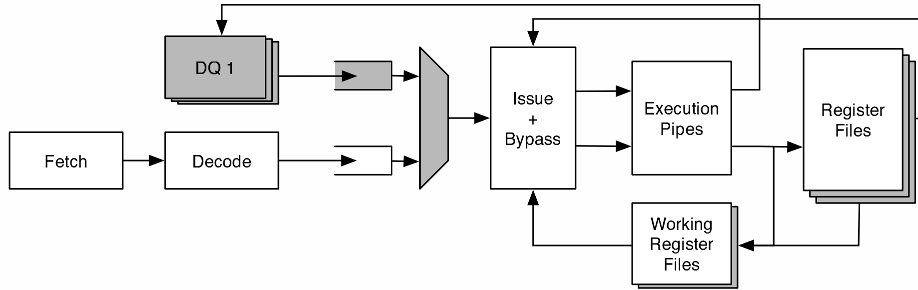


Figure 1: Pipeline Augmented with SST Logic.

2. SIMULTANEOUS SPECULATIVE THREADING

In order to support SST, each core is logically extended with the hardware structures shaded in Figure 1. In an SST implementation with N checkpoints per core (in the ROCK implementation, $N = 2$), there are N deferred queues (DQs) which hold decoded instructions and available operand values for instructions that could not be executed due (directly or indirectly) to a cache miss or other long-latency instruction. In addition to the architectural register file, N speculative register files and a second working register file are provided. Furthermore, each of these registers has an NA bit which is set if the register's value is "Not Available". Every instruction that is executed is classified as being either "deferrable" or "retirable". An instruction is deferrable if and only if it is a long-latency instruction (such as a load which misses in the L1 data cache, or a load or store which misses in the translation lookaside buffer) or if at least one of its operands is NA.

The core starts execution in a nonspeculative phase. In such a phase, all instructions are retired in order and update the architectural register file as well as a working register file. The DQs and the speculative register files are not used. When the first deferrable instruction is encountered, the core takes a checkpoint of the architectural state (called the "committed checkpoint") and starts a speculative phase. The deferrable instruction is placed in the first DQ and its destination register is marked as NA. Subsequent deferrable instructions are placed in a DQ and their destination registers are marked as NA. Subsequent retirable instructions are executed and speculatively retired. The retirable instructions write their results to a working register file and a speculative register file and clear the NA bits for the destination registers.

The core continues to execute instructions in this manner until one of the deferred instructions can be retired (e.g., the data returns for a load miss). At this point, one thread of execution, called the "ahead thread", continues to fetch and execute new instructions while a separate thread of execution, called the "behind thread", starts executing the instructions from the first DQ. Each instruction executed by the behind thread is again classified as being either deferrable or retirable. Deferrable instructions are re-inserted into the same DQ from which they were read and their destination registers are marked as NA. Retirable instructions write their results to a working register file and clear the NA bits for the destination registers. In addition, certain retirable instructions also update a speculative register file and the corresponding NA bits (using rules defined in Section 2.2.2).

At any given time, the ahead thread writes the results of its retirable instructions to a current speculative register file i and places its deferrable instructions in the corresponding DQ_i. Based on policy decisions, the ahead thread can choose to take a speculative checkpoint (if the hardware resources are available) and start using the next speculative register file and DQ at any time. For example, the ahead thread could detect that DQ_i is nearly full and therefore choose to take a speculative checkpoint i and start using speculative register file $i + 1$ and DQ_{i+1}. In any case, the ahead thread must take a speculative checkpoint i before the behind thread can start executing instructions from DQ_i.

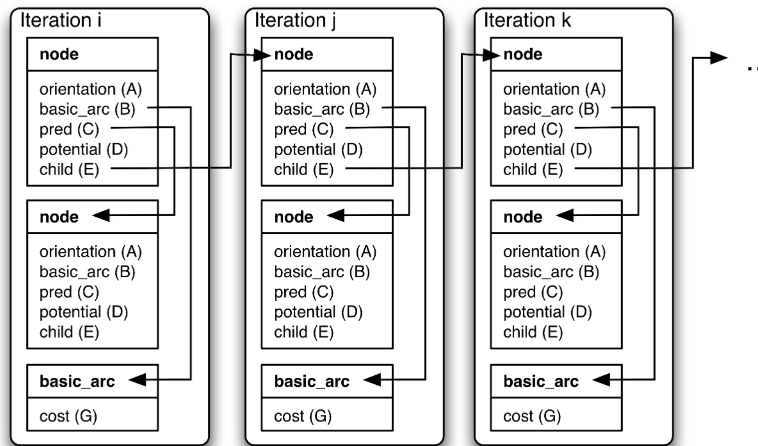
At any given time, the behind thread attempts to execute instructions from the oldest DQ. In particular, assuming that the oldest DQ is DQ_i, the behind thread waits until at least one of the instructions in DQ_i can be retired, at which point the behind thread executes all of the instructions from DQ_i, redeferring them as necessary. Once all of the instructions in DQ_i have been speculatively retired, the committed checkpoint is discarded, speculative register file i becomes the new committed checkpoint, and speculative register file i is freed (and can thus be used by the ahead thread when needed). This operation will be referred to as a "commit". Next, the behind thread attempts to execute instructions from DQ_{i+1}, which is now the oldest DQ. At this point, if the ahead thread is deferring instructions to DQ_{i+1}, it takes a speculative checkpoint and starts using speculative register file $i + 2$ and DQ_{i+2} (this is done in order to bound the number of instructions placed in DQ_{i+1}).

If at any time there are no deferred instructions (that is, all DQ's are empty), a "thread sync" operation is performed. The thread sync operation ends the speculative phase and starts a nonspeculative phase. The nonspeculative phase begins execution from the program counter (PC) of the ahead thread. For each register, if the ahead thread's speculative copy of the register was NA, the behind thread's value for the register (which is guaranteed to be available) is taken as the architectural value. On the other hand, if the ahead thread's speculative copy of the register was available, the ahead thread's value for the register is taken as the architectural value. In either case, the register's NA bit is cleared. Execution in the nonspeculative phase continues until the next deferrable instruction is encountered.

During a speculative phase in which the ahead thread is using speculative register file i and DQ_i, it is possible

```
while( node )
{
    if( node->orientation == UP )
        node->potential = node->basic_arc->cost + node->pred->potential;
    else /* == DOWN */
    {
        node->potential = node->pred->potential - node->basic_arc->cost;
        checksum++;
    }
    tmp = node;
    node = node->child;
}
```

(a) mcf code



(b) mcf data

Figure 2: Example of mcf Code Fragment and Data Structures

that the ahead thread will run out of hardware resources or encounter an instruction that cannot be executed speculatively. For example, it could exceed the capacity of the store queue or DQ or it could encounter a non-cacheable instruction that could have a side-effect. At this point, the ahead thread marks speculative checkpoint i as being not committable, it stops placing instructions in DQ_i, and it continues execution in order to prefetch data and instructions (that is, it operates as a hardware scout thread [4]). When speculative checkpoint i becomes the oldest speculative checkpoint, the speculative phase ends and a nonspeculative phase begins execution from the committed checkpoint.

Similarly, during a speculative phase in which the behind thread is executing instructions from DQ_i, it is possible that the behind thread is able to resolve a branch which was unresolvable by the ahead thread when it was first executed (because the condition code or register on which the branch depends was NA). If the behind thread determines that the branch was mispredicted, and thus that the ahead thread executed on the wrong path, the speculative phase ends and a nonspeculative phase begins execution from the committed checkpoint.

In a speculative phase, the ahead thread always reads and writes one working register file (denoted the "ahead thread working register file") and the behind thread always reads and writes the other working register file (denoted the "behind thread working register file"). In a nonspeculative phase, all register reads and writes access the ahead thread working register file.

2.1 Example of SST Execution

During a speculative phase, extensive instruction-level parallelism (ILP) is achieved by having the ahead thread and the behind thread execute from separate points in the program. In general, the behind thread executes a long-latency instruction (such as a load that misses in the cache) and its dependent instructions. While these instructions are being executed by the behind thread, the ahead thread is able to execute and speculatively retire independent instructions. Furthermore, the ahead thread is able to discover new cache misses, thus providing MLP in addition to ILP.

Figure 2(a) shows a loop from the mcf benchmark in SPEC INT 2006. Figure 2(b) shows the data structures accessed by that code. Consecutive iterations of the loop are denoted i, j, and k. Certain fields are denoted $A - G$ in order to relate them to their corresponding load operations in Figure 3. An example of an SST execution of this code is given in Figure 3, where loop iterations i, j, and k

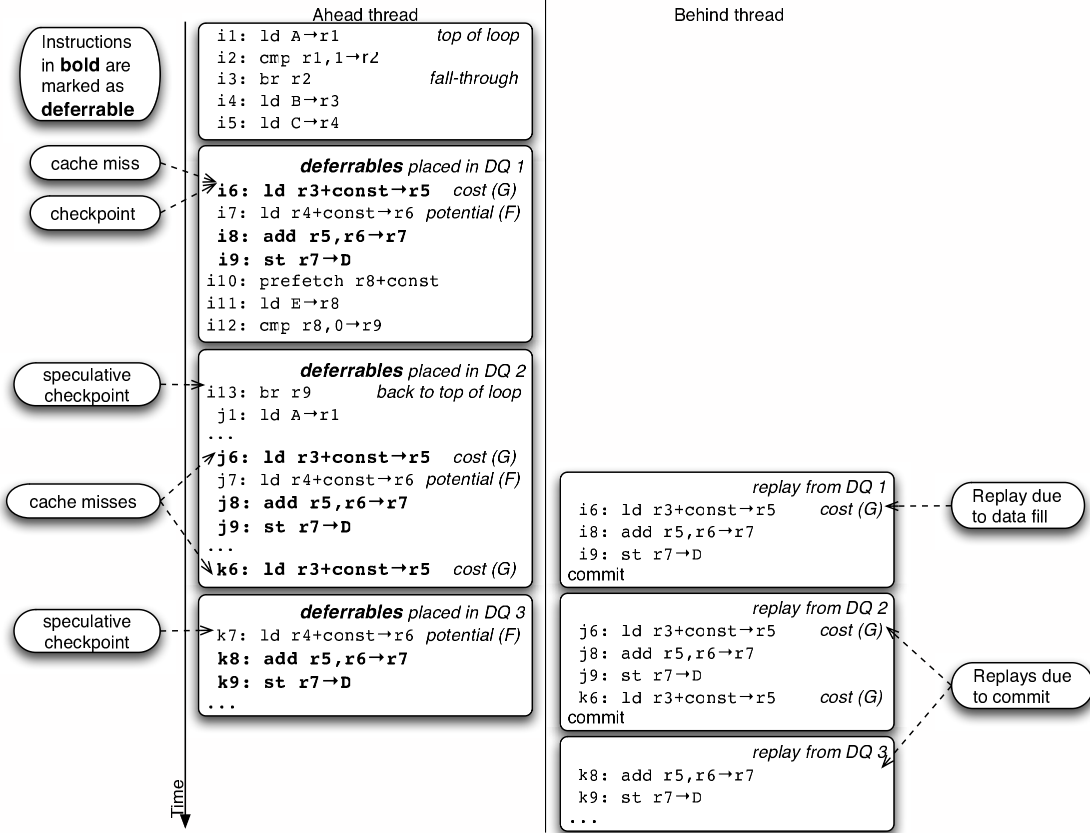


Figure 3: Example mcf SST Execution.

of instructions $1 - 13$ of the assembly code are shown. For example, $i10$ denotes the execution of instruction 10 in iteration i. In the common case, the load of the child (E) by instruction 11, and thus the successive loads of the orientation (A) by instruction 1, basic_arc (B) by instruction 4, and pred (C) by instruction 5, all hit in the cache due to a software prefetch by instruction 10 executed in an earlier iteration. In addition, the load of potential (F) from a predecessor node by instruction 7 usually hits in the cache. In contrast, the load of cost (G) by instruction 6 usually misses in the cache.

In this example instruction $i6$ is a cache miss. Therefore, a checkpoint is taken and instruction $i6$ and its dependents, $i8$ and $i9$, are placed in DQ_1. Instructions $i7$, $i10$, $i11$, and $i12$ are independent of the load miss and thus are speculatively retired to speculative register file 1 by the ahead thread. In this example, the ahead thread chose to take a speculative checkpoint on instruction $i13$. Next, the ahead thread speculatively retires instruction $i13$ and instructions 1 through 5 of iteration j by updating speculative register file 2. Instruction $j6$ is a cache miss, so $j6$ and its dependents are deferred to DQ_2.

Next, data returns for instruction $i6$ which is then executed from DQ_1 by the behind thread. The behind thread also executes deferred instructions $i8$ and $i9$, at which point DQ_1 becomes empty. As a result, speculative checkpoint 1 becomes the committed checkpoint. At this point, the ahead thread takes a new speculative checkpoint and the behind thread starts executing deferred instructions from DQ_2. Instructions $j6$ and $k6$ are cache hits when executed by the behind thread, so all instructions from DQ_2 are speculatively retired and speculative checkpoint 2 is committed. The behind thread continues by executing instructions from DQ_3 and the ahead thread continues to execute new instructions.

Note that the ahead thread and the behind thread are able to execute in parallel, thus extracting ILP. Furthermore, the performance of the ahead thread is unaffected by cache misses. As long as the behind thread is able to successfully retire the deferred instructions, the overall performance approaches that of a core with an infinite data cache.

2.2 Implementation

2.2.1 Registers

The above description of SST uses an architectural register file, N speculative register files and two working register files. In addition, in the description above, the operation of taking a speculative checkpoint requires copying the ahead thread's working register file to a new speculative register file and the operation of starting a nonspeculative phase requires copying the committed checkpoint to the architectural register file. In reality, it is possible to remove the separate architectural register file, to eliminate the above register file copy operations, and to implement most of the register files in SRAM. In order to achieve these improve-

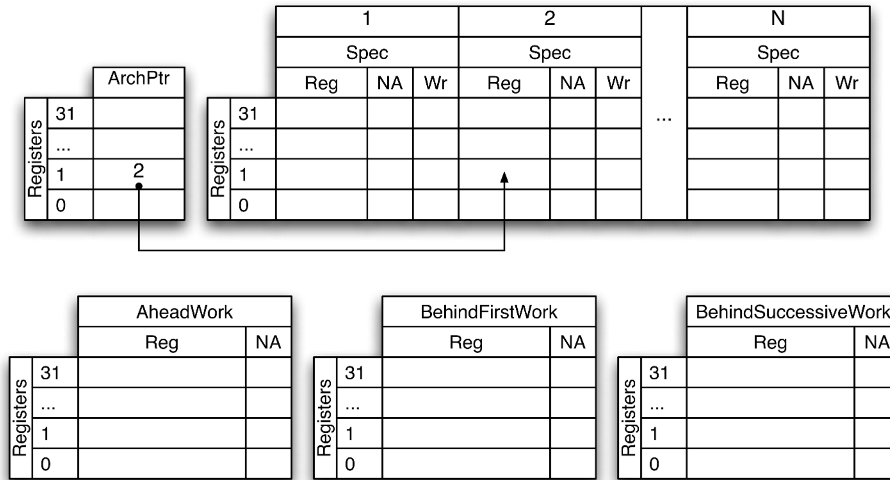


Figure 4: SST Register Implementation.

ments, the behind thread's working register file is split into two.

Figure 4 shows a more efficient implementation of the register files. Note that the separate architectural register file has been eliminated. Instead, for each architectural register j, ArchPtr[j] points to the SpecReg copy which holds the architectural value for register j. Also, corresponding to each set of speculative registers, SpecReg_i ($1 \leq i \leq N$), there is a set of speculatively written bits, SpecWr_i, and a set of NA bits, SpecNA_i. Whenever register j is written in a nonspeculative phase, $\text{SpecReg}_k[j]$, where $k = \text{ArchPtr}[j]$, is written. Whenever $\text{SpecReg}_k[j]$ is written in a speculative phase, $\text{SpecWr}_k[j]$ is set. Whenever a speculative phase which uses speculative register file k commits, for each register j, $\text{ArchPtr}[j] := k$ if $\text{SpecWr}_k[j]$ is set and ArchPtr[j] is unchanged otherwise. Then, $\text{SpecWr}_k[j]$ is cleared. As a result, there is no need to copy the ahead thread's working register file to a new speculative register file when taking a speculative checkpoint. Similarly, when starting a nonspeculative phase, there is no need to copy the committed checkpoint to the architectural register file. Therefore, all time-sensitive register copy operations have been eliminated (it is still necessary to copy the committed checkpoint registers to a working register file when starting a nonspeculative phase due to failed speculation; however, in this case the pipeline has to be flushed and restarted from the committed checkpoint PC, so there are many idle cycles in which this can be done). Furthermore, note that all register reads are from a working register file. Therefore, the speculative register files can be single-ported and thus can be implemented in an area-efficient manner using SRAM.

2.2.2 *Removing Register Hazards*

During a speculative phase, the ahead thread and behind thread operate largely independently. In particular, no results are passed from the behind thread to the ahead thread. The DQs are used to pass available operands of deferrable instructions from the ahead thread to the behind thread. No other values are ever passed from the ahead thread to the behind thread. As a result, each thread has its own scoreboard and no register bypasses are provided between the two threads.

In order to remove register Read-After-Write (RAW) hazards, two mechanisms are used. First, the per-thread scoreboards are used to block instruction issue until all operands are guaranteed to be ready or marked as NA. Second, whenever an instruction has an NA operand, the instruction is classified as being deferrable and will not become retirable until all its operands are available.

Write-After-Read (WAR) hazards are eliminated similarly. The per-thread scoreboard guarantees that between the issue of the producer and consumer of a register, no later instruction (in program order) which writes to the same register is allowed to issue. Furthermore, when the producer of the register first makes it available (as opposed to NA), the consumer is either retirable (in which case it consumes the provided value) or it is deferrable (in which case it stores the provided value with the instruction in the DQ). As a result, in either case the register is free to be overwritten.

The NA bits are used to remove Write-After-Write (WAW) hazards as follows. The ahead thread always writes results to both its working register file and the corresponding speculative register file. The first time that the behind thread executes an instruction from a given DQ_i, it reads operands that were not stored in DQ_i from the BehindFirstWork Register file (BFWR) and writes its result to the BFWR and updates the corresponding BehindFirstWork NA (BFWNA) bit. In addition, it checks the SpecNA_i bit for the destination register. If that bit is set, the destination register's value in speculative checkpoint i has not yet been produced, so the instruction also writes its result to the destination register in SpecReg_i (but does not update the corresponding SpecNA_i). Once the behind thread has executed all instructions from DQ_i, it "merges" the SpecNA_i and BFWNA bits by setting each SpecNA_i bit to the logical-AND of itself and the corresponding BFWNA bit.

When and if the behind thread executes a given instruction for a successive (the second or any other subsequent) time from a given DQ_i, it reads operands that were not stored in DQ_i from the BehindSuccessiveWork Register file

(BSWR) and writes its result to the BSWR and updates the corresponding BehindSuccessiveWork NA (BSWNA) bit. In addition, it checks the SpecNA_i bit for the destination register. If that bit is set, the destination register's value in speculative checkpoint i has not yet been produced, so the instruction also writes its result to the destination register in SpecReg_i and in BFWR (for use by the behind thread when it first executes instructions from DQ_{i+1}), updates the corresponding BFWNA bit, but does not update the corresponding SpecNA_i bit. Each time the behind thread completes executing all instructions from DQ_i, it "merges" the SpecNA_i and BFWNA bits as described above.

Intuitively, the above rules guarantee that once the youngest writer within a speculative checkpoint i has written a given register, that value cannot be overwritten. As a result, when speculative checkpoint i commits, SpecReg_i contains the value produced by the youngest writer within checkpoint i. Similarly, when speculative checkpoint i commits, the BFWR contains all values that may be needed by the behind thread when executing from DQ_{i+1}. Finally, the BSWR contains all values that are passed from one instruction to another in the second or later execution of instructions from a DQ.

Note that a single working register file would not be sufficient for the behind strand. For example, consider the execution shown in Figure 3 and now assume that during the first replay of instructions from DQ_2 that instruction $j6$ is a cache miss while instruction $k6$ is a cache hit. In this case, instructions $j6$, $j8$, and $j9$ are all redeferred to DQ_2. Instruction $k6$ writes BFWR[5] and detects that $\text{SpecNA}_2[5]$ is set, and therefore it writes to $\text{SpecReg}_2[5]$ and clears $\text{SpecNA}_2[5]$. Next, when instruction $j6$ is replayed from DQ_2 the second time, assume that it is a cache hit. In this case, instruction $j6$ now writes BSWR[5] and detects that $\text{SpecNA}_2[5]$ is not set, and therefore it does not write to BFWR[5]. Next, instruction $j8$ obtains the value of its source register $r5$ from instruction $j6$ via BSWR[5]. Then, instruction $j9$ executes and is speculatively retired and speculative checkpoint 2 is committed. The behind thread now executes instruction $k8$ from DQ_3, which obtains the value of its source register $r5$ from instruction $k6$ via BFWR[5]. In this example, when instruction $j6$ executes from DQ_2 the second time, two versions of register $r5$ must be stored, one of which will be used by instruction $j8$ and one of which will be used by instruction $k8$.

Given the above use of the NA bits, and the per-thread scoreboard, the WAW hazards are prevented. In particular, consider any register r that is written in SpecReg_i and let L denote the last instruction in program order which writes this register. If L was retirable when it was first executed, r is written by L and its corresponding SpecNA_i bit is cleared, so no other instruction can write to r in SpecReg_i. On the other hand, if L was deferrable when it was first executed, the SpecNA_i bit corresponding to r will remain set until L becomes retirable, at which point L will update register r in SpecReg_i and the following "merge" operation will clear the corresponding SpecNA_i bit. As a result, all other instructions will be prevented from writing to register r in SpecReg_i.

Similarly, consider any register r that is written in BFWR and let L denote the last instruction in program order which writes this register. Note that L must have been deferrable when it was first executed. As a result, the SpecNA_i bit corresponding to r will remain set until L becomes retirable, at which point L will update register r in BFWR and the following "merge" operation will clear the corresponding SpecNA_i bit. As a result, all other instructions will be prevented from writing to register r in BFWR.

2.2.3 *Store Queue*

All stores are placed in the store queue in program order by the nonspeculative or ahead thread (even if the value and/or address of the store is NA). Each store in the store queue has an NA bit for its address, an NA bit for its data, a speculative bit (which is set if the store was executed in a speculative phase and is cleared when the speculative phase is committed), and the corresponding checkpoint number. Stores for which the speculative bit is set in the store queue are not allowed to update memory. If a store is executed in a speculative checkpoint which never commits (due to a speculation's failure), the store is removed from the store queue.

Because all stores are placed in the store queue in program order, loads that are performed by the nonspeculative or ahead thread will see the value of the most recent store preceding the load. Loads performed by the behind thread can be executed when younger (in program order) stores have already been placed in the store queue. As a result, each load and store is assigned an "age" value in program order. In particular, the age of a store gives the location of the store in the store queue and the age of a load gives the location in the store queue of the most recent store preceding the load in program order. When the behind thread executes a load, the load's age is used to limit the RAW-bypasses provided by the store queue to be from older stores only.

When a store has an NA bit set for its address or data, the operands for the store were not available and thus the store was classified as being deferrable. When a deferrable store is executed from a DQ its age is used to update its store queue entry by updating its address and data fields and the corresponding NA bits. If a load receives a RAW-bypass value from a store in the store queue for which the NA bit is set for the store's data, the load is classified as being deferrable and its destination register's NA bit is set. However, when a load is executed after a store with an NA address has been placed in the store queue, it is impossible to determine whether or not that load should receive a RAW-bypass from that store. As a result, the load is executed assuming that no such RAW-bypass is required. If the load is speculatively retired, a Memory Disambiguation Buffer (MDB) can be used to track the address of the load and to fail speculation if it is later determined that a RAW-bypass was missed. Alternatively, if no MDB is used, the speculative checkpoint containing the load must be marked as being not committable and the ahead thread continues execution as a hardware scout thread.

2.2.4 *Load Ordering*

During a speculative phase, loads can be speculatively retired out of program order with respect to other loads. In order to support a memory model such as Total Store Order (TSO), it is necessary to make it appear as though the loads were retired in program order. This is accomplished by maintaining an array of N speculatively-loaded bits (called "s-bits") per each data cache entry. Whenever a load which is part of speculative checkpoint i is speculatively retired,

s-bit[i] of the cache line entry read is set. Upon invalidation or victimization of a cache line from the data cache, if s-bit[i] of the cache line is set, speculative checkpoint i is made to fail. When a speculative checkpoint i either commits or fails, s-bit[i] of every line in the data cache is cleared. Note that by grouping speculative instructions into N distinct speculative checkpoints, it is possible to use this simple s-bit mechanism to track load reordering rather than using a more traditional Content-Addressable Memory (CAM) structure.

2.2.5 *Issue Logic*

Because loads that miss in the cache can be executed (by classifying them as deferrable) with the same latency as loads that hit in the cache, it is possible to have a fixed latency for every instruction. This greatly simplifies the issue logic, as it enables each decoded instruction to have its time of execution scheduled without waiting to determine the latency of its producers. In particular, it is possible to obtain a minimal load-use latency without having the issue logic speculate on load hits and misses.

3. SIMULATION METHODOLOGY

3.1 Simulator Infrastructure

Results were produced by an in-house cycle-accurate simulator capable of simulating SST and the other architectures studied in this paper. The simulator, when configured with ROCK's parameters, has been validated against ROCK's hardware implementation. A sampling approach similar to the one described by Wenisch et al. [26] was used.

3.2 Simulated CMP

The simulations use several core-clusters (containing L1-instruction and L1-data caches), a shared L2 cache, and four on-chip memory controllers. The core-clusters and the L2 cache are connected with crossbar networks. A core-cluster consists of four cores, each of which has its own L1-data cache. Cores are grouped pairwise, where each pair shares an instruction fetch unit (IFU) and an L1-instruction cache. An IFU can fetch up to 16 instructions per cycle. The core consists of five execution units (2 integer, 1 branch, 1 memory, and 1 floating point) and has a sustainable issue rate of four instructions per cycle. The parameters used for the cache hierarchy and branch predictors are shown in Table 1.

ROCK's implementation of SST is similar to the default configuration described above but does differ in a few key areas. For example, ROCK has 16 SST threads (or, alternatively, 32 non-SST threads), a 2 MB second-level cache, and two checkpoints per core. In addition to the default parameters specified above, we also simulated configurations with 8 cores and with 2 to 16 checkpoints per core.

3.3 Benchmarks

Simulation results are given for three commercial server-side benchmarks and the SPEC CPU2006 suite, all of which were compiled with the SunStudio compiler. The commercial benchmarks consist of OLTP, JBB, and SAP. OLTP is an on-line transaction processing server-side database workload. The transactions include entering and delivering orders, recording of payments, checking the status of orders, and monitoring the level of stock at warehouses. JBB is the SPECjbb2005 server-side Java benchmark that models a 3-tier system, focusing on middle-ware server business logic and object manipulation. SAP is a server-side "Sales and Distribution" benchmark that covers a sell-from-stock scenario.

Feature	SST	OoO
Number of cores	32 (8 core clusters)	
L1 i-cache	32KB/IFU, 4-way, 2 cycle, next-line prefetch, 64B line	
L1 d-cache	32KB/core, 4-way, 3 cycle, 64B line	
L2 cache	4MB/chip, 8-way, 8 banks, 25 cycle, 64B line	
Store queue	64 entries/core, 2 banks	
Branch predictor	32K entries/IFU, 13 bits history, 13 cycles mispred	
BTB	128 entries/IFU	
RAS	8 entries/core	
Memory latency	300 cycles	
Trap / fail latency	24 cycles	
Checkpoints	8 per core	NA
DQ size	32 entries each	NA
Checkpoint policy	On every branch or after 30 instructions	NA
Issue window	2 entries/pipe	32 entries
Reorder buffer	NA	128 entries

Table 1: Default Simulated CMP Parameters.

4. PERFORMANCE RESULTS

4.1 SST Execution Analysis

As mentioned in Section 2, the core can be in a non-speculative phase (NonSpeculative), a speculative phase in which the ahead thread has not detected a failure condition (Speculative), or a speculative phase in which the ahead thread acts as a hardware scout (HWS) thread. Figure 5 shows the number of cycles spent in each of these phases for the various benchmarks.

Note that high miss-rate applications (such as the commercial benchmarks and mcf) spend almost all of their time in a speculative phase, while lower miss-rate applications (such as perlbench and sjeng) spend the majority of the time in a non-speculative phase. A few of the benchmarks (such as lbm and JBB) spend significant time in an HWS phase.

Figure 6 shows the frequency of the three most common types of speculation failure, namely DQ overflow, store queue (SQ) overflow, and deferred mispredicted branches (Branch). The frequency of other reasons for failure, such as non-cacheable memory accesses, is negligible.

The figure shows that the largest source of failures for OLTP is deferred mispredicted branches. This can be explained by its irregular branch pattern in combination with a high cache miss rate. On the other hand, most of the failures for gcc are due to store queue overflows which are caused by numerous memory copy operations. Finally, a significant source of failures for mcf is DQ overflow, which is indicative of its high L2 cache miss rate.

Note that when the ahead thread causes a DQ or store queue overflow, it continues execution in an HWS phase. In contrast, when a deferred branch is determined to have

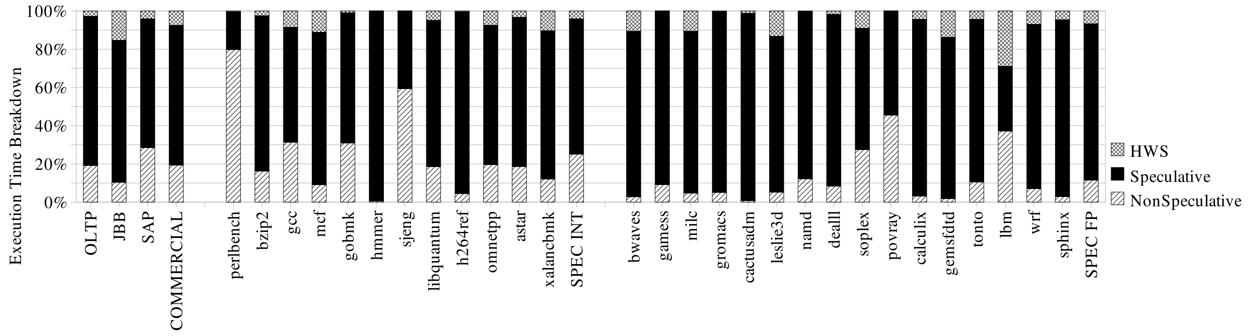


Figure 5: SST Execution Mode Breakdown.

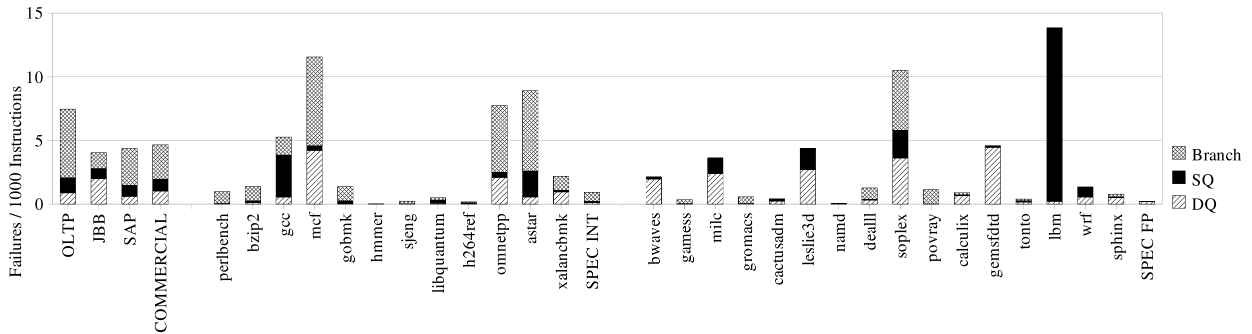


Figure 6: Speculation Failures per 1000 Instructions.

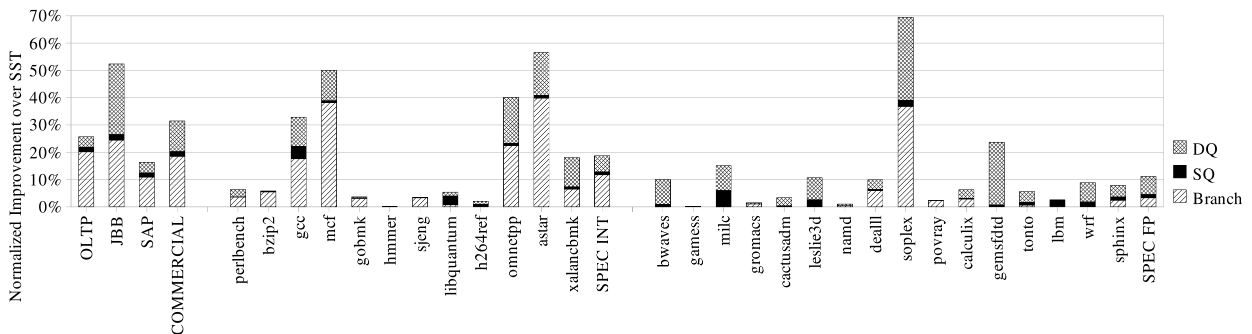


Figure 7: SST Performance Improvement from Eliminating Failures.

been mispredicted, speculation fails immediately and the core starts a nonspeculative phase from the committed checkpoint. As a result, benchmarks that have many DQ and/or store queue overflows in Figure 6 tend to spend more cycles in HWS phases as shown in Figure 5. In contrast, benchmarks that have many deferred mispredicted branches do NOT tend to spend more cycles in HWS phases.

In order to quantify the performance impact of each type of speculation failure (and to bound any potential improvement from corresponding architectural improvements), a limit study shown in Figure 7 was performed.

In this study, the effect of deferred mispredicted branches was quantified by modeling the performance gain provided by an "oracle" that detects mispredictions of deferrable branches when they are first executed. As can be seen in Figure 7, such an oracle would improve performance by nearly 20% on commercial benchmarks, so it is clear that a significant number of speculative cycles are lost due to deferred mispredicted branches. While it is obviously impossible to achieve the idealized results which depend on an oracle, the results do highlight the value of any reduction in the mispredict rate for deferred branches. Note that it would be possible to create a second branch predictor which only predicts the outcome of deferrable branches (thus greatly limiting the number of branches for which it makes predictions) and which only needs to make its prediction at execution time rather than instruction fetch time (thus greatly increasing the time which can be spent in this predictor). As a result, creating such specialized branch predictors is a promising topic for future research. Another approach to reducing de-

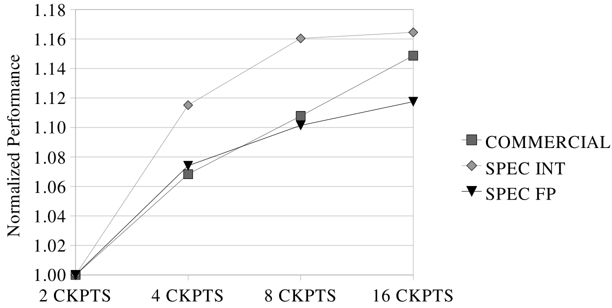


Figure 8: SST Performance vs. Number of Checkpoints.

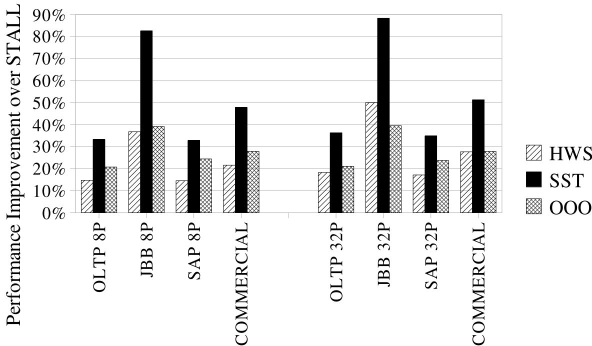


Figure 9: Commercial Performance.

ferred mispredicted branches would be to use predicated instructions to replace hard-to-predict branches that are often deferred.

Figure 7 also shows the effect of increasing the size of the store queue to 256 entries per core and the DQs to 512 entries per checkpoint. These sizes were chosen to virtually eliminate failures due to overflows. Increasing the store queue size gives very limited performance gains overall but for individual applications it can be significant (e.g. 5% for gcc). Finally, increasing the DQ size gives significant improvements for many applications (e.g. 26% for JBB). However, additional simulations have shown that increasing the DQ size is not effective unless failures from deferred mispredicted branches are also reduced.

An important design decision to make for an SST architecture is the number of checkpoints to support. Figure 8 shows how performance varies with the number of checkpoints. For all applications, there is a significant improvement when increasing from two to eight checkpoints. For the commercial workloads, performance continues to improve when doubling the number to 16, while SPEC INT and SPEC FP see modest improvements in that range. All simulation results except for this sensitivity analysis were obtained with eight checkpoints.

4.2 Comparison with other Architectures

A comparison of SST and other architectures is shown in Figures 9, 10, and 11. The figures show the performance for three different architectures in 8-core (left) and 32-core (right) CMPs. All simulation results are normalized to in-order stalling cores (STALL). The figures show the performance of the stalling cores extended with Hardware Scout (HWS) [4], cores that support SST, and traditional OoO cores.

Starting with the commercial benchmarks in the 8-core configuration, HWS improves performance by 22% over STALL. This is primarily due to increased MLP since the HWS processor does not overlap any computations while waiting for a cache miss but only warms the caches and the branch predictor. Since the commercial applications have a high L2 miss-rate, this prefetching effect is significant. OoO improves performance 28% over STALL (and 5% over HWS). The gain for OoO compared to STALL consists of both MLP and ILP. Note, however, that OoO extracts less MLP than HWS does due to its limited issue window size, while HWS does not require an issue window and thus does not encounter such a limitation. Finally, like OoO, SST also achieves both MLP (by executing speculatively under a cache miss) and ILP (by executing the ahead and behind threads in parallel). In fact, SST extracts even more MLP than HWS due to the ability of the ahead thread to continue execution while the behind thread consumes data that was prefetched by the ahead thread. Consequently, SST is 21% faster than HWS and 16% faster than OoO. Looking next to the 32-core configuration, the benefit of SST over OoO is even larger (18%) due to the greater L2 cache miss rate and, hence, increased importance of MLP.

The SPEC INT benchmarks behave differently from the commercial benchmarks due to their lower L2 cache miss rates. As can be seen from Figure 10, the MLP extracted by HWS provides very limited speedup over STALL. In fact, in some cases HWS performs worse than STALL due to its restart latency when data for the cache miss returns. In contrast, SST and OoO both improve performance significantly, due to their ability to extract ILP. In an 8-core CMP, OoO is slightly better than SST (by 3%). This is because OoO uses a fine-grained, CAM-based scheduler which is able to select exactly those instructions which have all of their operands available. In contrast, SST eliminates the need for a CAM-based scheduler at the cost of executing some instructions prior to their operands being available. In a 32-core CMP, the results are reversed with SST slightly outperforming OoO (by 2%) as a result of the increased L2 cache miss rate (and thus significance of MLP).

For the SPEC FP benchmarks, OoO shows a 3% improvement over SST with both 8 and 32 cores. This is due to extensive use of software prefetching (thus reducing the impact of hardware-extracted MLP) and the importance of a large issue window for extracting ILP. Because the SST configuration uses a very small (2-instruction) issue window for the floating-point pipeline while the OoO design uses a 32-instruction issue window, the OoO design is able to reorder instructions more efficiently.

4.3 Reliance on CAM-Based Structures

In the results above, it was assumed that both the SST and OoO cores had a CAM-based MDB [8, 11] as well as an unlimited number of physical registers used for renaming (which also utilize CAMs in many implementations). Such CAM-based structures are particularly undesirable in a CMP as they require significant area and power. Figure 12 quantifies the extent to which the performance gains of SST and OoO rely on these structures.

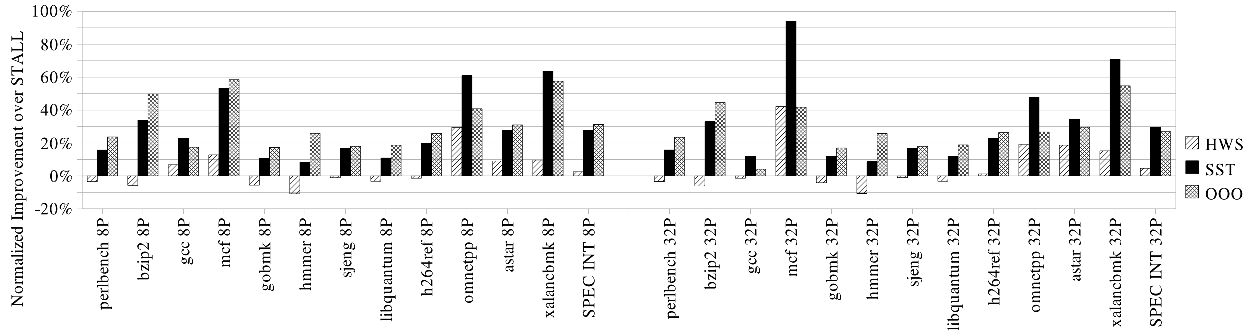


Figure 10: Integer Performance.

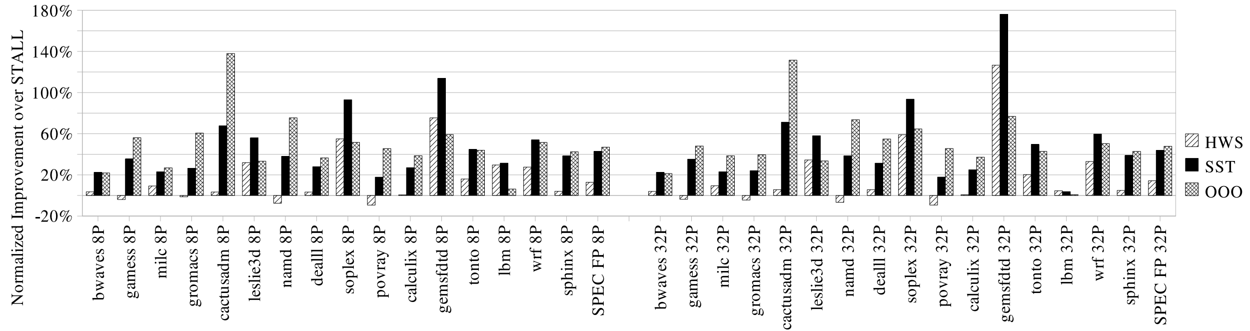


Figure 11: Floating-Point Performance.

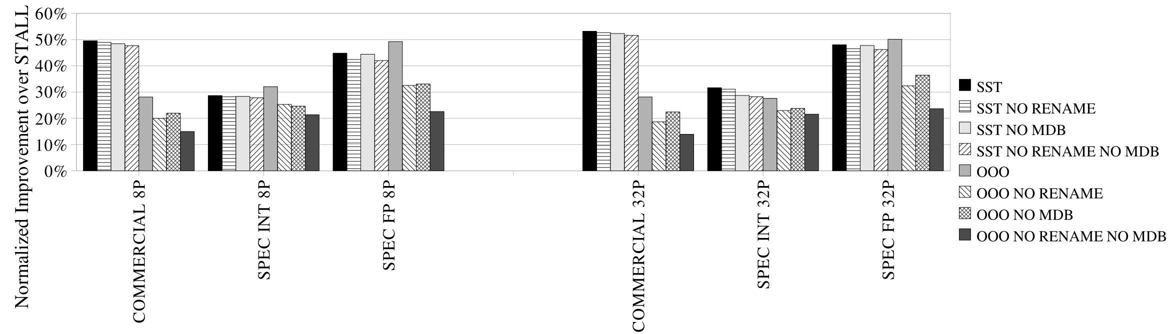


Figure 12: Performance Gains from MDB and Register Renaming for SST and OoO.

The results in Figure 12 are normalized to STALL. It is clear from these results that OoO is far more dependent on these expensive structures than is SST. In particular, when the memory disambiguation buffer is not present, SST outperforms OoO on commercial, integer, and floating point benchmarks at both 8 and 32 cores per chip. Similarly, when register renaming is not present, SST also outperforms OoO on commercial, integer, and floating point benchmarks at both 8 and 32 cores per chip. In fact, in a 32 core CMP, SST without memory disambiguation and without register renaming is 18% faster on commercial workloads and equal on integer workloads compared to OoO with memory disambiguation and with register renaming. Comparing configurations without either of these expensive structures, SST outperforms OoO by 33% on commercial, 5% on integer, and 18% on floating point benchmarks. Furthermore, the SST configurations have a much smaller issue window and no ROB.

SST's lack of reliance on register renaming can be explained by considering the case of a load miss. In OoO without register renaming, instructions that are dependent (directly or indirectly) on a load miss, or which have a WAR or WAW dependence (directly or indirectly) on the load, cannot issue until the load completes. In SST without register renaming, these instructions can issue once the load has been determined to be a cache miss. Therefore, the instructions with WAR or WAW dependence on the load can issue (and be made retirable) much earlier with SST. As can be seen from Figure 12, SST virtually eliminates the need for register renaming.

Similarly, SST's lack of reliance on memory disambiguation can also be explained by considering the case of a load miss. In OoO, a store that is dependent on a load miss is blocked until the load completes. As a result, in OoO without memory disambiguation, younger loads (and their dependents) will be blocked until the original load miss completes. In contrast, in SST without memory disambiguation, the store issues without waiting for the earlier load miss to complete, thereby allowing younger loads to issue. In the case of a store with NA data, SST continues execution by deferring younger loads to the same address. In the case of a store with an NA address, SST continues execution as a hardware scout thread, thus extracting MLP. As can be seen from the results in Figure 12, this eliminates the need for an MDB with SST.

5. RELATED WORK

Lebeck et al. [14] use a waiting instruction buffer to reduce the size of the issue window in an OoO processor. The waiting instruction buffer plays a role that is similar to the DQs in SST. However, SST differs by eliminating the need to retire instructions in order and thus is able to reclaim those instructions' resources earlier.

Checkpointing has been presented as a technique to address scalability issues with structures that support OoO execution [1, 6, 15]. KIP [6] uses checkpoints and a register renaming mechanism to reduce the size of the issue window and reorder buffer (ROB) in an OoO core. CPR [1] uses checkpoints and a register renaming mechanism to reduce the number of physical registers and to eliminate the ROB. SST places known operands for deferred instructions in an SRAM structure (DQ), while KIP and CPR keep such operands (and registers that are part of a checkpoint) in a multiported physical register file. In addition, SST differs from CPR in that SST does not rely on a large issue window. Cherry [15] uses a single checkpoint in order to reduce the number of physical registers and MDB entries in an OoO core. In contrast, SST supports multiple checkpoints (thus eliminating the need to periodically return to nonspeculative execution) and SST speculatively retires instructions out-of-order. Finally, SST differs from all of the above techniques in that SST obtains ILP and MLP via multiple threads of execution and thus does not require an OoO core with register renaming, a large instruction issue window, or an MDB. Furthermore, SST differs from all of those techniques in that it has an ability to operate as a scout thread in the event that resources overflow by using the NA bits in registers to avoid generating memory addresses that are likely to be incorrect.

Hardware scout execution has been proposed and evaluated (under a variety of names) in both in-order and out-of-order cores by several researchers [4, 7, 16, 17]. The key difference between HWS execution and SST is that once a speculative phase completes in an HWS architecture, all speculatively produced results are discarded and have to be re-executed, while an SST architecture does not need to re-execute speculatively retired instructions.

The two architectures in the research literature that are most similar to SST are the flea-flicker two-pass pipeline proposed by Barnes et al. [2] and continual flow pipelines by Srinivasan et al. [23]. The flea-flicker scheme has an ahead and a behind thread where long-latency instructions (and their dependents) are re-executed by the behind thread. The flea-flicker scheme differs from SST in that it uses two separate pipelines rather than executing both threads in the same pipeline. Secondly, in flea-flicker the second pass pipeline stalls execution if a deferred instruction has a cache miss, while SST redefers the instruction and thereby clears the issue window. Continual flow pipelines were also introduced in the context of a checkpoint-based architecture. The key difference between continual flow pipelines and SST is that in continual flow pipelines re-execution of deferred instructions is not performed in parallel with execution of younger independent instructions. Finally, SST differs from both techniques in its ability to exploit multiple checkpoints.

While other researchers have proposed combined hardware/software approaches to extracting independent threads of execution from a single program thread [9, 13, 20, 21, 22], SST differs from these approaches by performing its parallelization automatically in hardware.

6. DISCUSSION

As can be seen from the above performance results, SST provides an efficient core architecture for CMPs. As the first hardware implementation of SST, the ROCK processor contains 16 cores with 2 checkpoints each. This design is suitable for the 65nm process that was used and provides a good balance between throughput and per-thread performance. As can be seen from Figure 8, designs with 4 or more checkpoints per core are also interesting to consider for future implementations. The ROCK processor implements SST without an MDB, as SST performance has very little reliance on this structure (see Figure 12).

In ROCK, the checkpointing mechanism used in SST was leveraged to improve the efficiency of the issue logic. In particular, when operating in a nonspeculative phase, ROCK is able to take a checkpoint on a branch instruction, thus causing all younger instructions to write to a speculative register file. As a result, it is possible to execute instructions younger than the branch before the branch, and to handle the misprediction of such a branch as though it were a failure of speculation.

7. CONCLUSIONS

This paper introduced Simultaneous Speculative Threading (SST), which was shown to be an effective architecture for extracting both ILP and MLP. Furthermore, it is efficient, as it requires only a very small (2-entry per execution pipeline) issue window, no reorder buffer, no register renaming, and no memory disambiguation buffer. SST was shown to deliver higher performance on commercial applications than a traditional OoO core in a CMP. Due to their greater ability to extract MLP, certain SST configurations also outperform OoO on integer applications as the number of cores on the chip is increased. Sun's ROCK processor implemented 16 SST cores, with 2 checkpoints each, in a 65nm process.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank the entire ROCK team for their great effort and dedication to making the SST concept a reality.

9. REFERENCES

[1] Akkary, H., Rajwar, R., and Srinivasan, S. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO'03)* (Dec. 2003), pp. 423–434.

[2] Barnes, R., Patel, S., Nystrom, E., Navarro, N., Sias, J., and Hwu, W. Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining. In *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO'03)* (Dec. 2003), pp. 287–398.

[3] Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. Piranha: A Scalable Architecture based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)* (June 2000), pp. 282–293.

[4] Chaudhry, S., Caprioli, P., Yip, S., and Tremblay, M. High-Performance Throughput Computing. *IEEE Micro 25*, 3 (2005), 32–45.

[5] Chaudhry, S., Cypher, R., Ekman, M., Karlsson, M., Landin, A., Yip, S., Zeffer, H., and Tremblay, M. ROCK: A High-Performance SPARC CMT Processor. *IEEE Micro 29*, 2 (2009).

[6] Cristal, A., Santana, O., Cazorla, F., Galluzzi, M., Ramirez, T., Pericas, M., and Valero, M. Kilo-Instruction Processors: Overcoming the Memory Wall. *IEEE Micro 25*, 3 (2005), 48–57.

[7] Dundas, J., and Mudge, T. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)* (July 1997), pp. 68–75.

[8] Gallagher, D. M., Chen, W. Y., Mahlke, S. A., Gyllenhaal, J. C., and Hwu, W. W. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)* (Oct. 1994), pp. 183–193.

[9] Hammond, L., Willey, M., and Olukotun, K. Data Speculation Support For a Chip Multiprocessor. *SIGOPS Operating Systems Review 32*, 5 (1998), 58–69.

[10] Herlihy, M., and Moss, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Computer Architecture News 21*, 2 (1993), 289–300.

[11] Kessler, R. The Alpha 21264 Microprocessor. *IEEE Micro 19*, 2 (1999), 24–36.

[12] Kongetira, P., Aingaran, K., and Olukotun, K. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro 25*, 2 (2005), 21–29.

[13] Krishnan, V., and Torrellas, J. A Chip-Multiprocessor Architecture With Speculative Multithreading. *IEEE Transactions on Computers 48*, 9 (1999), 866–880.

[14] Lebeck, A. R., Koppanalil, J., Li, T., Patwardhan, J., and Rotenberg, E. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)* (May 2002), pp. 59–70.

[15] Martinez, J., Renau, J., Huang, M., Prvulovic, M., and Torrellas, J. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proceedings of the 35th IEEE/ACM International Symposium on Microarchitecture (MICRO'02)* (Nov. 2002), pp. 3–14.

[16] Mutlu, O., Kim, H., and Patt, Y. N. Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance. *IEEE Micro 26*, 1 (2006), 10–20.

[17] Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architeture (HPCA'03)* (Feb. 2003), pp. 129–140.

[18] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. The Case for a Single-Chip Multiprocessor. *SIGPLAN Notices 31*, 9 (1996), 2–11.

[19] Palacharla, S., Jouppi, N. P., and Smith, J. E. Complexity-Effective Superscalar Processors. *SIGARCH Computer Architecture News 25*, 2 (1997), 206–218.

[20] Quiñones, C. G., Madriles, C., Sánchez, J., Marcuello, P., González, A., and Tullsen, D. Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. *SIGPLAN Notices 40*, 6 (2005), 269–279.

[21] Rundberg, P., and Stenström, P. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *Journal of Instruction-Level Parallelism 3*, 1 (2001), 2002.

[22] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)* (June 1995), pp. 414–425.

[23] Srinivasan, S. T., Rajwar, R., Akkary, H., Gandhi, A., and Upton, M. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)* (Oct. 2004), pp. 107–119.

[24] Tremblay, M., Chan, J., Chaudhry, S., Conigliam, A., and Tse, S. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro 20*, 6 (2000), 12–25.

[25] Tremblay, M., and Chaudhry, S. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *Proceedings of the 2008 International Solid-State Circuits Conference (ISSCC'08)* (Feb. 2008), pp. 82–83.

[26] Wenisch, T., Wunderlich, R., Falsafi, B., and Hoe, J. Simulation Sampling with Live-Points. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software (ISPASS'06)* (Mar. 2006), pp. 2–12.