Journal of Instruction-Level Parallelism 5(2003) Submitted 10/02; published 4/03

Speculative Sequential Consistency with Little Custom Storage

Chris Gniady GNIADY@CMU.EDU
Babak Falsafi BABAK@CMU.EDU
Computer Architecture Laboratory,
Carnegie Mellon University, 5005 Forbes Ave, Pittsburgh, PA 15213, U.S.A.

Abstract

This paper proposes SC++lite, a sequentially-consistent system that relaxes memory order speculatively to bridge the performance gap among memory consistency models. Prior proposals to speculatively relax memory order require large custom on-chip storage to maintain a history of speculative processor and memory state while memory order is relaxed. SC++lite uses the memory hierarchy to store the speculative history, providing a scalable path for speculative SC systems across a wide range of applications and system latencies. We use cycle-accurate simulation of shared-memory multiprocessors to show that SC++lite can fully relax memory order while virtually obviating the need for custom on-chip storage. Moreover, while demand for storage increases significantly with larger memory latencies, SC++lite's ability to relax memory order remains insensitive to memory latency. An SC++lite system can improve performance over a base SC system by 26% with only 1.7KB of custom storage in a system with 16 processors. In contrast, speculative SC systems with custom storage require 51.4KB of storage to improve performance by 29% over a base SC system.

1. Introduction

Sequential Consistency (SC) is the most intuitive programming interface for shared-memory multiprocessors. A system implementing SC appears to execute memory operations one at a time and in program order [11]. A program written for an SC system requires and relies on a specified memory behavior to execute correctly. Implementing memory accesses according to the SC model constraints, however, would adversely impact performance because memory accesses in shared-memory multiprocessors often incur prohibitively long latencies (tens of times longer than in uniprocessor systems). Researchers and vendors have alternatively relied on relaxed memory consistency models that augment the shared-address space programming interface with directives enabling software to inform hardware when memory ordering is necessary [1]. By otherwise allowing hardware to relax and overlap multiple memory accesses, systems implementing relaxed consistency models achieve high performance.

Recent research indicates that through hardware support for speculative execution, an SC-compliant system, referred to as SC++, can speculatively relax memory order to achieve the performance of a Release Consistent (RC) system, the consistency model previously enabling the highest performance [8]. The intuition behind this result is that an SC system must only *appear* to execute memory accesses in order. SC hardware on one processor can relax memory order from that processor as long as other processors do not observe the relaxed order [5,8,10,15]. Much as in speculative instruction execution in modern processors, an SC++ system requires to buffer the history of processor and memory state while speculatively relaxing memory order. Using this history, the hardware must roll back to an SC-compliant state if one processor attempts to access memory that has been accessed out-of-program-order by another.

While the results in [8] serve as a proof of concept that SC systems can achieve the performance of RC systems, SC++ requires a custom on-chip queue (to store the processor/memory state history) proportional in size to the maximum memory access latency incurred in the system. Unfortunately, memory latencies drastically vary within and across applications, resulting in infrequent but large bursts of history, and requiring a large custom queue that is mostly underutilized. Moreover, memory latencies also largely vary across systems depending on the memory subsystem and interconnect speeds, the system size, and the contention induced due to the workload. Because multiprocessor servers are typically built using commodity microprocessors, providing the right size custom queue to satisfy the requirements of a wide spectrum of applications and systems would be prohibitively difficult.

This paper proposes *SC++lite*, a speculative SC system that virtually eliminates the custom storage needed to support SC++. SC++lite spills the processor/memory history information generated by SC++ into each processor's local memory hierarchy, offering a scalable storage across a wide spectrum of applications, system sizes and latencies. We use cycle-accurate simulation of distributed shared-memory (DSM) multiprocessors running scientific and engineering applications to compare SC++lite's performance and storage requirements against SC++'s.

The contributions of this paper are:

- **Detailed history characterization:** We present a detailed characterization of SC++ history information and corroborate that: (1) queue requirements vary between 16 to 8192 entries for applications and systems we studied, and (2) the history information is quite bursty, on average leaving the queue empty 85% of the application execution time.
- **Speculative SC with little custom storage:** Our results indicate that SC++lite on average performs 26% better than a base SC system with *only* 1.7KB of storage. In contrast, SC++ requires 51.4KB to achieve a 29% average speedup over a base SC system. Moreover, SC++lite's performance relative to SC++ remains unchanged with a four times increase in memory latencies, while SC++'s storage requirements double to 101.4KB.
- **Sensitivity to L1/L2 bandwidth & L2 size:** We show that on average there is little interference with an application's L2 footprint and processor's L1/L2 traffic even when using small L1 caches due to the bursty nature of history. Applications with high L2 bandwidth requirements benefit from an additional L2 port in SC++lite.

The rest of the paper is organized as follows. In Section 2., we describe the current high-performance SC systems. In Section 3., we present a design for SC++lite. In Section 4. we present the experimental methodology. In Section 5., we present the results. Finally, we conclude the paper in Section 6..

2. Background: High-Performance SC

Sequential consistency (SC) provides the most intuitive programming interface by requiring that all memory operations appear to execute in program order and atomically [11]. In conventional SC implementations, the processor would faithfully implement SC's ordering constraints, performing memory operations atomically and in program order one memory operation at a time, blocking on cache misses. Such a memory system would preclude non-blocking caches and overlapping accesses among multiple memory operations.

Figure 1 illustrates an example of a memory bottleneck in an SC system. The figure illus-

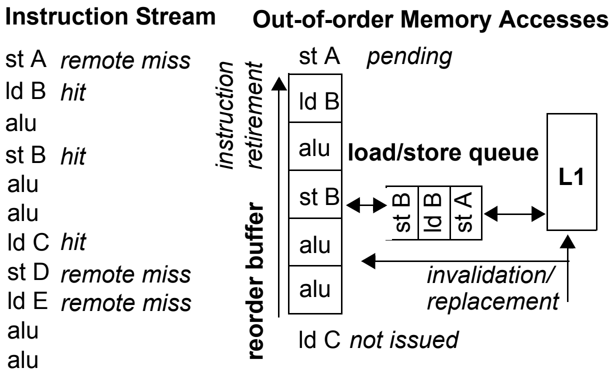

FIGURE 1. Example execution in SC.

trates instruction flow and ordering of memory accesses in an out-of-order processor pipeline (e.g., MIPS R10000 [19]). The figure also illustrates the common case of a program segment in which memory accesses to distinct addresses are independent, and do not require program ordering. In the example shown, while all memory accesses are independent and the cache blocks containing address *B* are present, a naive SC system would wait for the store to address *A* (waiting for either a missing cache block or a write permission to a read-only cache block) to complete, before executing the load to address *B* and its corresponding computation. Unfortunately, store latency in a DSM is typically hundreds of processor cycles because each remote access includes multiple network transactions and may require invalidating several sharers of a cache block. Therefore, the store to address *A* in this example potentially blocks the flow of instructions for hundreds of processor cycles.

Modern SC systems implement a spectrum of optimizations to reduce the negative impact of a pending store on performance. Early acknowledgment of invalidation messages (e.g., in AlphaServer GS320 [7]) helps partially hide the store latency in systems with ordered network messages. Non-blocking caches allow for overlapping multiple fetch operations (including write permission for stores) in arbitrary order into L1 [5] while satisfying the SC constraints by performing L1 accesses atomically and in program order. Store buffering [6] allows pending stores and subsequent computation up to a load instruction to retire from the reorder buffer. Such optimizations, however, only partially reduce the exposed store latency and *can not* eliminate the performance gap between SC and relaxed memory systems such as Release Consistency (RC) [8,15].

A key technique (e.g., adopted by MIPS R10000) to further hide a pending store latency in an out-of-order processor core is *speculative load execution* [5]. In our example, the processor using speculative load execution would allow for the load to address *B* to hit in the memory hierarchy, and the corresponding computation to complete while a store to address *A* is pending. Relaxing the memory order speculatively does not violate SC's constraints as long as no other processor in the system observes the reordering — i.e., no other processor modifies the speculatively accessed data. To guarantee SC semantics, all requests for replacement or invalidation (from other processors) of cache blocks that are speculatively accessed roll execution back to the offending instruction. Unfortunately, the limited size of the reorder buffer prevents the system from realizing the full potential of relaxing memory order [8,15]. The reorder buffer is primarily designed to tolerate branch resolution latency which is in the order of tens of processor cycles and can not tolerate DSM store latencies that are one or more orders of magnitude larger.

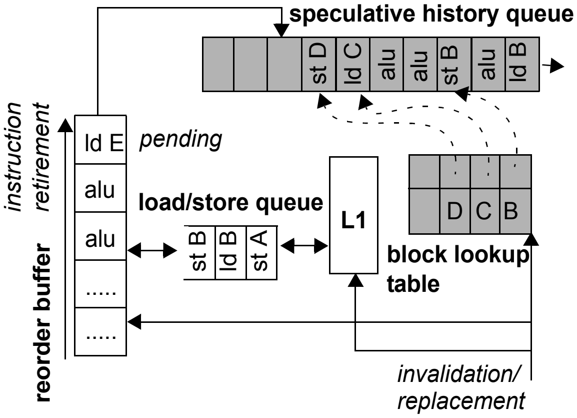

FIGURE 2. Example execution in SC++.

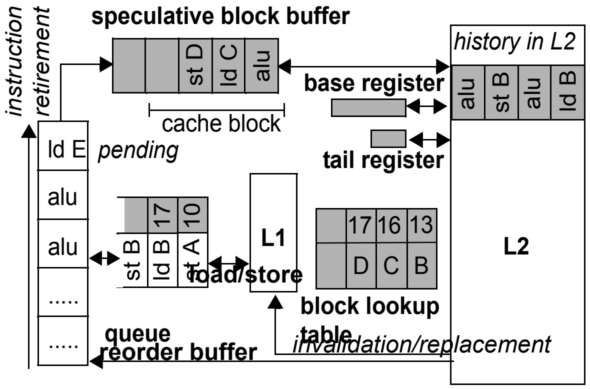

FIGURE 3. Example execution in SC++lite.

2.1 Speculative SC with RC Performance

In a recent paper Gniady, et al, [8] identified the requirements for an SC system to fully achieve the performance of an RC system to be: (1) allowing arbitrary re-ordering of (load/ store) memory accesses to distinct memory locations, (2) providing sufficient buffering to maintain a history of all instructions executed while an in-program-order store is pending, (3) providing fast mechanisms to look up remote processor requests for speculatively-accessed memory blocks and detect potential model violation, and (4) exhibiting infrequent rollbacks in workloads.

Gniady, et al. [8] also proposed a speculative SC system, SC++, that satisfies the above requirements. Figure 2 depicts how SC++ speculatively relaxes memory order. Upon a pending store, SC++ speculatively retires instructions and records the modified processor and memory state in a speculative history queue (implemented much like a history buffer [16]). In the example shown, SC++ retires the instructions up to the missing load to address *E* and stores them in the history queue.

Upon acknowledgment for the completion of the first in-program-order pending store, all entries up to the next pending store on the list are discarded; an acknowledgment indicates that the memory accesses maintained in the history while the store was pending were not observed by other processors and the SC constraints are satisfied. To allow for locating the portion of the history to be discarded, each load/store queue entry for a pending store also includes a pointer to the location of the instruction in the speculative history queue. The pointer also enables locating the entry within the queue to record the old memory value corresponding to the store address when a cache block for a missing store arrives. In the example shown, when the acknowledgment for the store to address *A* arrives, all entries up to the store to address *D* are discarded. An acknowledgment for a later in-program-order pending store (e.g., the store to address *D*) while an earlier store (e.g., the store to address *A*) is pending simply removes the corresponding entry from the load/store queue but does not discard any history.

A block lookup table provides a quick mechanism to verify speculation. The lookup table maintains a list of cache blocks that are speculatively accessed by instructions in the queue. In our example in the figure, the table keeps track of all speculatively-accessed blocks at addresses *B*, *C*, and *D* while the store to *A* is pending. A hit in the lookup table upon an invalidation/replacement request from L2 indicates a potential for violating SC semantics and triggers a rollback to guarantee SC's ordering constraints. Upon rollback, SC++ locates the earliest instruction accessing the block in the queue, and rolls back execution to this offending instruction. To guarantee forward progress, execution restarts after all pending stores are acknowledged and the queue and table are empty.

Each lookup table entry also keeps a pointer to the last (in-program-order) instruction accessing the block. The pointer helps clear the table entries for blocks without any corresponding instructions in the history queue. Upon discarding entries in the history queue, all table entries pointing to the discarded history entries are also cleared, indicating that speculation for the corresponding cache block addresses has been verified. In our example, the completion of the store to address *A* results in discarding the history entries corresponding to addresses *B C*, and *D* and clearing the corresponding table entries.

Gniady, et al. [8] showed that SC++ performs as well as an RC system for well-synchronized and scalable parallel programs. However, an SC++ system may require a prohibitively large history queue depending on the system size, memory and interconnect speeds, and application memory access characteristics. In this paper, we show that to achieve maximum performance, an SC++ system with 16 processors must incorporate queues with up to 101.4KB of storage per processor for the applications and interconnect speeds we studied. We also show that the speculative history is quite bursty, leaving the queue empty for over 65% of processor cycles in all applications we studied. The variation in history size precludes selecting a custom queue size that fits the demands of a large spectrum of applications and systems.

3. SC++Lite: SC++ with Minimal Storage

This paper proposes *SC++lite*, a speculative implementation of SC that spills the speculative history into the memory hierarchy (much as CNI [12] spills network messages into main memory). The key advantage of SC++lite is that the history can grow as large as the memory hierarchy can accommodate. SC++lite offers a scalable path for speculation because larger systems with longer latencies also incorporate larger caches that increase the capacity for speculative history. Moreover, SC++lite provides speculation with minimal impact on either an application's footprint in the memory hierarchy or bandwidth into the caches, because the history is typically bursty and accumulates infrequently [8]. Finally, SC++lite allocates history storage dynamically (through the caches), eliminating the need for large underutilized custom storage.

Figure 3 depicts the anatomy of SC++lite. SC++lite maintains the speculative history in the memory hierarchy. The queue is allocated in main memory and assigned physical addresses at boot time (much as Sun WildFire [9,4], IBM Prism [3], and CNI [12] allocate DRAM pages to hardware managed caches and queues). A *Speculative Block Buffer (SBB)* accumulates the instructions retiring from the reorder buffer and stores them as cache blocks in L2. The number of entries in SBB is proportional to the pipeline's issue width. A *tail register* indicates the location of the queue's tail in physical memory. While memory locations of queue entries can be recorded as full physical addresses, in practice the queue will only at most occupy a small fraction of the main memory. To reduce the storage overhead for queue pointers (i.e., the tail pointer, and pointers from the load/store queue and the lookup table), all pointers only record a number of the low order address bits. A *base register* records the high order address bits and is concatenated to all queue pointers to form a full physical address.

As in SC++, SC++lite uses the lookup table to quickly look up speculatively-accessed blocks and detect a potential ordering violation. Much as the history queue, the lookup table size requirements grow with system size and workload. However, because of the high degree of locality in cache block addresses and the small size of lookup table entries, a lookup table with 256 entries (~1.3KB) sufficed for all application and system sizes we studied. Alternatively, a scalable lookup table implementation could use state bits in the cache hierarchy to eliminate the auxiliary table. In this paper, however, we focus on eliminating the custom history queue which accounts for the substantial storage overhead in speculative SC systems. In the rest of this section, we describe SC++lite's queue insertion, deletion, lookup, and rollback operations. We also present L2 optimizations that help enhance performance in SC++lite.

3.1 Spilling & Discarding History

SBB behaves like a miniature version of the history queue as long as the accumulated history fits in it. Upon packing a complete cache block, the SC++lite logic queues a request for an L2 port (see Section 3.3). Upon receiving a free port, SBB ships the packed history for storage into L2, and shifts the contents of the buffer forward. The logic updates the queue's tail address upon insertion. The history stored in L2 can spill all the way down to the lowest level of the memory hierarchy.

Upon spilling into L2, the history blocks may encounter speculatively-accessed blocks in the cache. Unfortunately, blindly replacing such blocks triggers inadvertent rollbacks to guarantee correct speculation [8] and may result in high performance overheads in SC++lite due to frequent history spilling. In Section 3.3, we propose a speculation-aware replacement policy in L2 to avoid such rollbacks by selecting non-speculative blocks for replacement.

As in SC++, an acknowledgment for the first in-program-order pending store requires discarding of the corresponding history. Discarding history requires updating the queue's head address and the lookup table. As in SC++, SC++lite records the position of every pending store in the load/store queue. Unlike SC++, the recorded locations are physical addresses in memory. Upon acknowledgment of the first in-program-order pending store, the corresponding entry is removed from the load/store queue and the next pending store entry in the load/store queue points to the head of the history queue. It is also necessary to identify the cache blocks containing the discarded history in the memory hierarchy. Section 3.3 discusses the necessary L2 mechanisms to remove these blocks.

Similarly, the lookup table records the position in memory of the last instruction that has speculatively accessed a given cache block. Upon updating the queue's head address, all lookup table entries with locations outside the new queue address range are cleared. In the example in Figure 3, the store to address *A* points to the physical address 10 in memory. Upon acknowledgment of the store, all history up to the store to address *D* is discarded and the queue's head address in memory becomes 17. Similarly, the lookup table entries corresponding to blocks *B*, *C*, and *D* are cleared because the history corresponding to the last instruction accessing them is discarded.

3.2 Misspeculation & Rollback

Gniady, et al., [8] showed that in well-synchronized and scalable applications, misspeculations are extremely infrequent. The intuition behind such an observation is that a misspeculation only happens as a result of a true data race among processors on a specific address. In well-synchronized applications, data races typically only occur on synchronization addresses (e.g., a lock guarding entry into a critical section) which account for a small fraction of all memory accesses. Moreover, frequent data races on synchronization addresses result in high contention for critical sections and are not characteristics of scalable parallel applications. As such, while a speculative SC system must implement rollback correctly, rollback speed and efficiency is not a key design concern and does not significantly impact overall system performance.

As in SC++, when a replacement/invalidation message probes and hits in the lookup table, there is a potential for violation of SC's ordering semantics and the system must roll processor/memory state back to the offending instruction (i.e., the earliest in-program-order speculative instruction accessing the block). Rollbacks in SC++lite are potentially much slower than those in SC++ because the history must be retrieved from the memory hierarchy as compared to a custom hardware queue in SC++. Our results indicate that because rollbacks are very infrequent, SC++lite can achieve SC++'s performance even with a higher rollback latency and overhead.

3.3 L2 Footprint Optimizations

SC++lite can benefit from a few optimizations in L2. Upon store acknowledgment, invalidating the cache blocks containing the discarded history would be prohibitively expensive and would require multiple probes to the cache hierarchy. Because the head and tail pointers can always distinguish the valid history entries upon rollback, it is possible to leave the discarded history as valid/dirty cache blocks in the cache hierarchy. However, these blocks may generate unnecessary writeback traffic from L2 if replaced by the application's L2 footprint. Our results indicate that speculative history is quite bursty resulting in an insignificant probability of conflict in L2 with an application's footprint. Nevertheless, to entirely eliminate inadvertent L2 writeback of discarded history, we propose that the L2 controller check the queue head and tail pointers prior to writing back cache blocks.

L2 treats SBB write requests as L1 writebacks with the following exception. It is assumed that L2 does not maintain inclusion with respect to queue addresses so that write requests from SBB always write allocate and store the entire block as provided by SBB. As in L1 writebacks, an SBB request in L2 may generate an L2 writeback of a dirty block to lower levels. Upon an SBB write request, L2 writebacks of history blocks interfering with the SBB request will either proceed as is or will be optimized away by the L2 controller optimization discussed above.

The key to correct speculation in speculative SC systems is to hold on to all speculatively-accessed data in the memory hierarchy until speculation is verified. Upon an L2 miss either due to a data/instruction L1 miss (in SC++ and SC++lite) or SBB spill (in SC++lite), an attempt to replace a speculatively-accessed block (in L2) would result in a rollback. Such a rollback will not compromise forward progress [8] because execution starts at the offending instruction (i.e., the instruction accessing the block speculatively prior to replacement) which will execute non-speculatively. Therefore, in speculative SC systems, it would be desirable to avoid selecting speculatively-accessed blocks as candidates for replacement in a set-associative L2 cache. Avoiding these replacements may be critical in SC++lite because spilling speculative history into L2 may increase contention and trigger frequent rollbacks.

Gniady et al. [8] assumed a conventional LRU replacement policy because in the systems and applications they studied, rollbacks were too infrequent to justify a modified replacement policy. In this paper, we propose a modified LRU policy called *LRU Non-Speculative (LRUNS)* to minimize the frequency of rollbacks due to contention in L2 among regular blocks, speculatively-accessed blocks, and history blocks. Upon an L2 miss, LRUNS consults with the lookup table, prior to selecting a candidate for replacement, to identify blocks that are speculatively accessed. If all the blocks within a set are speculatively-accessed, LRUNS inhibits the L2 miss handling, and stops instruction retirement from the pipeline until either: (1) speculation is resolved and a block can be replaced, or (2) speculation is rolled back due to an invalidation triggered by another processor. These two conditions guarantee that execution eventually restarts L2 miss handling to proceed.

Similarly, SBB spilling may encounter speculatively-accessed blocks in L2 (as discussed in Section 3.1). To determine whether there are evictable blocks in L2 prior to spilling, SBB consults the lookup table. If evictable blocks do not exist, SBB inhibits spilling and stops the pipeline until a block is available for replacement. In the rest of this paper, we assume LRUNS as the base replacement policy in L2 for all speculative SC systems we study. In Section 5.6, we study the impact of a conventional LRU policy on SC++ and SC++lite's performance.

Finally, SC++lite allows speculative history to be written back to lower levels of the memory hierarchy, and therefore rollbacks may require retrieving history from arbitrary memory levels. However, the history information during rollback is read-only, can be discarded immediately, and does not require allocation in higher cache levels. In SC++lite, speculative history only travels up the hierarchy upon rollback. Therefore, we propose a modification to the cache controllers in the memory hierarchy to prevent allocation of history blocks upon a fetch based on the physical address range of the queue (assigned at boot time).

3.4 L2 Bandwidth Optimizations

SC++lite's bandwidth demand on L2 depends on the pipeline retirement rate, and the size of L1 cache blocks. When history accumulates, the L2 must allow a steady stream of speculative history blocks spilling out of SBB. Fortunately, because the speculative history is highly bursty (as we show in Section 5.), the average demand on L2 is quite low. Nevertheless, in applications with high L1 i-cache and/or d-cache miss rates, the history traffic may interfere with the application's L1/L2 traffic. Multiporting/multibanking or pipelining can help improve L2's available bandwidth at the cost higher implementation overhead. Recent proposals for chip multiprocessors include shared L2 caches with high degrees of interleaving/multibanking offering high bandwidth. Similarly, increasing the L1 block size and the L1/L2 bus size would allow for a higher rate of speculative history spill from the SBB. Finally, because history sizes are variable even within an application, a larger SBB can help reduce a large fraction of spills into L2 by filtering short history sequences, spilling only larger sequences.

4. Experimental Methodology

We use RSIM [14], a state-of-the-art DSM simulator to compare the speculative SC systems. We simulate a 16-node DSM with every node including a MIPS R10000 like processor [19] with a local memory hierarchy interconnected by a high-bandwidth/low-latency 2-D mesh. The memory controller on each node implements a 3-hop full-map directory cache coherence protocol [15]. Table 1 shows the base system parameters used throughout the experiments unless specified otherwise. The L2 local fill latency corresponds to the minimum cache fill latency from local memory on every node. The L2 remote fill latency is the average of the minimum cache fill latencies for one node from all remote nodes. The L2

CPU clock	1GHz
Issue/retire width	8-issue per cycle
Instruction window/ Reorder buffer size	128/128 insts.
Load/store queue size	128 insts.
L1 cache	32KB, direct-mapped
L2 cache	512KB, 8-way, pipelined, 64 GB/s
L1 fill latency	10 cycles
L2 local fill latency	100 cycles
L2 remote fill latency	200 cycles
Cache line size	64 bytes

Table 1: System configuration.

Application	**Input Parameters**	**RC/ SC**	**SC++ /SC**
appbt	12x12x12 cubes, 40 iter.	1.39	1.33
barnes	4K particles	1.10	1.12
em3d	16K nodes, 15% remote	1.23	1.24
fft	64K points	1.14	1.14
radix	512K keys	1.79	1.80
tomcatv	128x128, 50 iter.	1.14	1.13
unstructured	mesh 2K	1.50	1.50
water-ns	343 molecules	1.21	1.33
water-sp	343 molecules	1.18	1.18
Average		1.28	1.29

Table 2: Applications, inputs, and base speedups.

cache configuration and bandwidth correspond to those in Pentium 4 at 2.0 GHz [2]. We assume an 8-entry SBB in the base SC++lite system to allow for maximum retirement width in one cycle, but present performance and spill bandwidth sensitivity to larger SBBs. All systems implement MIPS R10000's SC optimizations including multiple (non-blocking) pending fetches for cache misses, store buffering, and speculative load execution.

Table 2 shows the nine shared-memory applications that we use in this study. *Appbt* is a shared-memory implementation of the NAS benchmark. *Em3d* is a shared-memory implementation of the Split-C benchmark. *Barnes, fft, radix*, *water spatial* and *nsquared* are from the SPLASH-2 benchmark suite [18]. *Unstructured* is a shared-memory implementation of a fluid dynamics computation using an unstructured mesh [13]. *Tomcatv* is a shared-memory implementation of the SPEC benchmark [17].

Table 2 also shows the base RC and SC++ speedups over SC. The results show that RC improves performance over an optimized SC implementation in a base system with aggressive interconnect latencies on (geometric) average by 28%. SC++ fully benefits from speculatively relaxing memory order and on average performs 29% better than SC. SC++ actually outperforms RC in *water-ns* because the RC system conservatively enforces memory order at synchronization points even though processors rarely race for critical sections in these applications. SC++ always relaxes order for all memory accesses in these applications even at synchronization points, improving performance over RC.

5. Results

In this section, we will first characterize history in speculative SC systems and show that speculative history size varies largely across applications and system latencies and is bursty. Next, we show that our proposed SC++lite system performs as well as an SC++ system while requiring an order of magnitude less custom storage. Next, we show that SBB causes, on average, little to no interference with L1/L2 traffic allowing SC++lite to efficiently exploit the memory hierarchy as storage for history. Finally, we show that SC++lite significantly improves performance in all applications even under a high contention in L2.

5.1 History Characteristics

Figure 4 shows the impact of fixing the history queue size on SC++'s performance. The figure plots performance under SC++ with a finite queue size as a fraction of performance

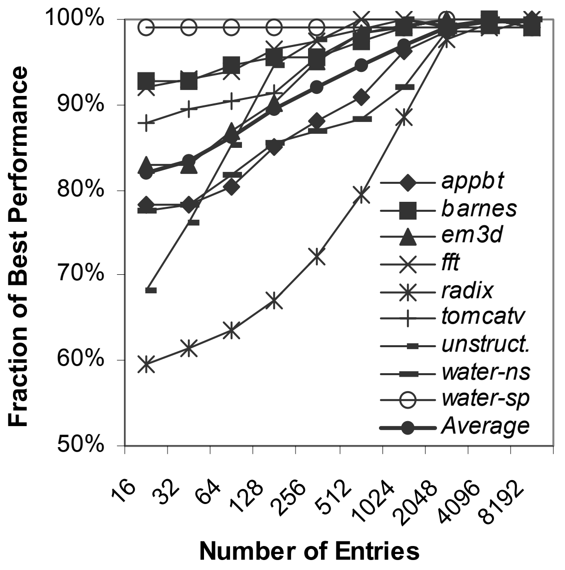

FIGURE 4. Performance sensitivity to queue size.

This figure plots the performance of SC++ given finite queue sizes as a fraction of SC++'s best performance given an infinite queue.

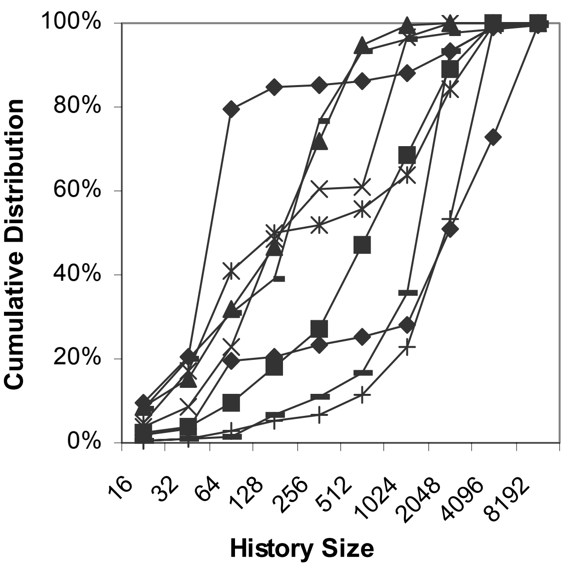

FIGURE 5. Queue utilization.

This figure plots the cumulative distribution of accumulated speculative history.

under an ideal SC++ implementation with an infinite queue. The figure indicates that there is a large variability in demand for queue size, ranging from an application that performs well with 16 queue entries (i.e., *water-sp*) to one (i.e., *radix*) that requires 8192 queue entries to reach maximum performance. The applications need about 4096 queue entries to reach, on average, the maximum performance. These results indicate a single queue size may not suffice to accommodate a wide spectrum of applications.

The required queue size for each application depends on the amount of exposed store latency in the processor pipeline. *Water-sp* primarily exhibits L2 store hits, and therefore generates little speculative history. In *radix*, however, loads and stores to remote memory are clustered respectively, and therefore there is little load latency or computation overlapping the store latencies. As such, much of the store latencies to remote memory are exposed.

Figure 5 shows the history queue utilization and Table 3 depicts the fraction of the execution time when the queue is empty. The table indicates that speculative history is quite bursty and, on average, 85% of the time the queue is empty. *Em3d, radix* and *water-ns* utilize the queue the most because there is a significant number of stores with exposed latencies in these applications. *Tomcatv*, and *barnes* use the queue infrequently. However, when a store is pending, there are large bursts of history falling within 2048 to 4096 entries to overlap the store latency. A dynamic allocation approach will be able to allocate resources only when needed, resulting in a better utilization of storage. Moreover, these results indicate the potential for storing the history in the memory hierarchy with little interference.

Application	*appbt*	*barnes*	*em3d*	*fft*	*radix*	*tomcatv*	*unstruct.*	*water-ns*	*water-sp*	Avg.
Fraction of execution time	85%	95%	68%	83%	73%	97%	77%	74%	99%	85%

Table 3: Fraction of execution time without any history.

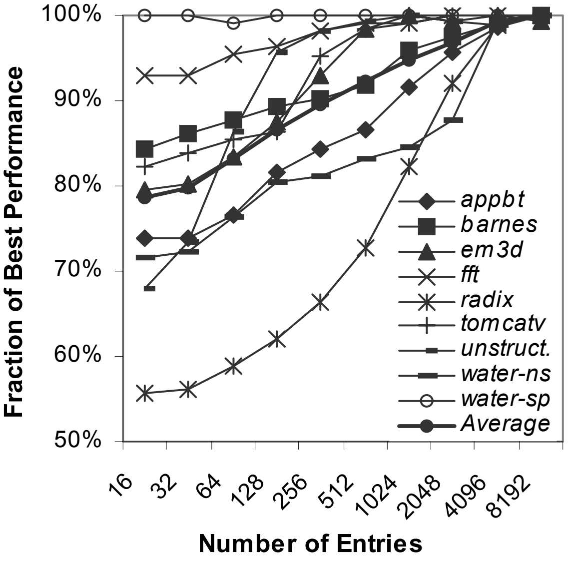

FIGURE 6. Performance sensitivity to queue size for 4x remote latency.

This figure plots the performance of SC++ given finite queue sizes as a fraction of SC++'s best performance given an infinite queue for systems with 4x L2 remote fill latency.

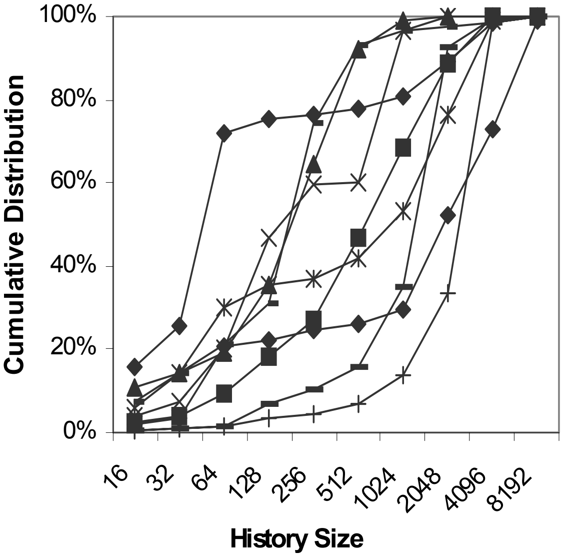

FIGURE 7. Queue utilization for 4x remote latency.

This figure plots the cumulative distribution of accumulated speculative history for systems with 4x L2 remote fill latency.

The distribution of generated history size also largely varies among applications. Figure 5 illustrates the cumulative distribution of history sizes when the queue is used. The history size is a function of both application and system characteristics. The key application characteristics that affects history size is the amount of a pending store latency overlapped after the store retires from the reorder buffer. Because a fetch for a store can be initiated as soon as the store enters the reorder buffer (using non-blocking caches), a fraction of the store latency can be overlapped before the store instruction leaves the reorder buffer. The extent to which the store latency, within the reorder buffer, can be overlapped depends primarily on any pending loads (missing in the cache) appearing prior to the store instruction in program order. Once a pending store retires from the reorder buffer, speculative history accumulates until a pending load reaches the top of the reorder buffer. In summary, the history size is a function of the distance in the program between a pending store and a prior or later pending load.

The main system characteristics are the pipeline retirement rate and the incurred memory latencies in the system. Therefore, if the application has a pending store and the pipeline is able to retire at a full rate, the resulting number of entries in the history will be the store latency multiplied by the retirement rate. This basic idea can be applied to classify the behavior of applications in Figure 5. To the first order of approximation, the knees in queue utilization around 64 to 128 entries are due to the L1 fill latencies, and the knees around 2048 entries are due to the L2 remote fill latencies that are affected by communication patterns and queuing. In case of *unstructured*, history size falls in the range of 128 to 512 entries that does not directly correspond to any system characteristics and shows us clearly the amount of remote store latencies that were partially overlapped by the subsequent or preceding loads. *Barnes*, *tomcatv*, and *water-ns* are able to overlap the L1 fill latencies resulting only in history generated due to remote store misses.

Figure 6, Figure 7 and Figure 8 illustrate the impact of longer L2 remote fill latencies on

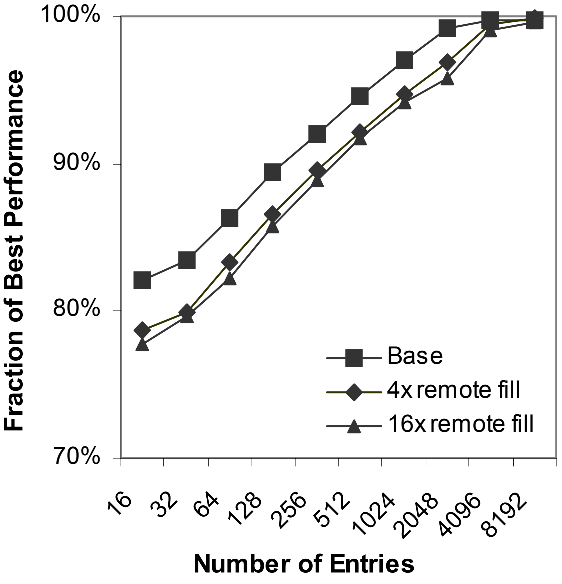

FIGURE 8. Average performance sensitivity to remote latency.

This figure plots performance of SC++ with finite queue sizes as a fraction of performance with an infinite queue for systems with base, 4x, and 16x L2 remote fill latencies.

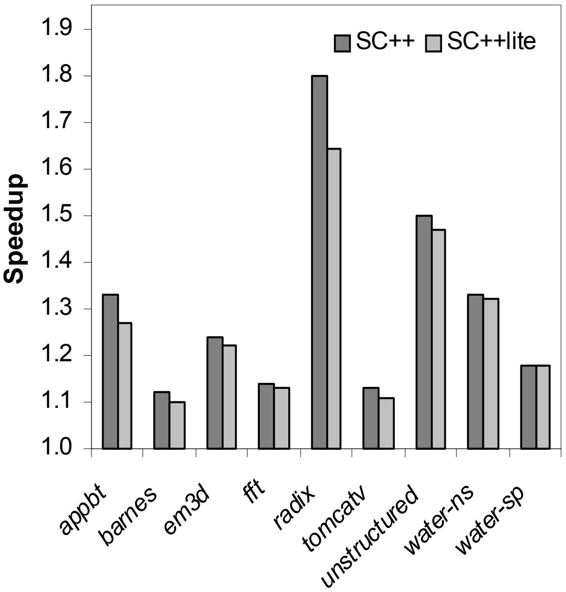

FIGURE 9. Base system performance.

This figure compares the performance of SC++lite against SC++, normalized to SC. SC++ uses 4K-entry custom history queue and SC++lite uses a 8-entry SBB.

the performance sensitivity to queue size. Quadrupling the remote fill latency results in approximately doubling of the execution time in the applications. Therefore, we can expect the queue size requirements to double in order to overlap the longer latencies and match the performance. In Figure 8, we see a right shift showing doubling in average queue size requirements to match the same performance range as compared to the base system. The latter results in an average queue size requirement of 8192 entries. Figure 7 shows the cumulative distribution of queue utilization for the 4x remote fill latencies, and the results follow the shape of those in Figure 5. Not surprisingly, the L1 fill latency region remains unaffected. However, the longer L2 remote fill latency results in a right shift of utilization knees in the remote latency range.

Figure 8 also includes the average for 16x latency network Figure 8. We observe that the history requirements do not grow significantly since there is a limit to the amount of exposed store latency dictated by a application characteristics.

5.2 SC++Lite Performance

Figure 9 compares the performance of SC++lite against SC++. The graph plots speedups with respect to our base SC system (Section 4.). We compare the performance and storage requirements of an SC++ system with 4096 custom queue entries, because most of our applications achieve their maximum performance given this custom queue size (Figure 4). SC++lite numbers assume a 8-entry SBB. The graphs indicate that SC++lite's performance matches closely the performance of SC++. On average, SC++ improves performance over SC by 29%, while SC++lite improves performance by 26% with little custom storage.

Figure 10 compares the performance of SC++lite and SC++ with the quadrupled L2 remote fill latency. The SC++ numbers assume a 8192-entry custom queue. On average,

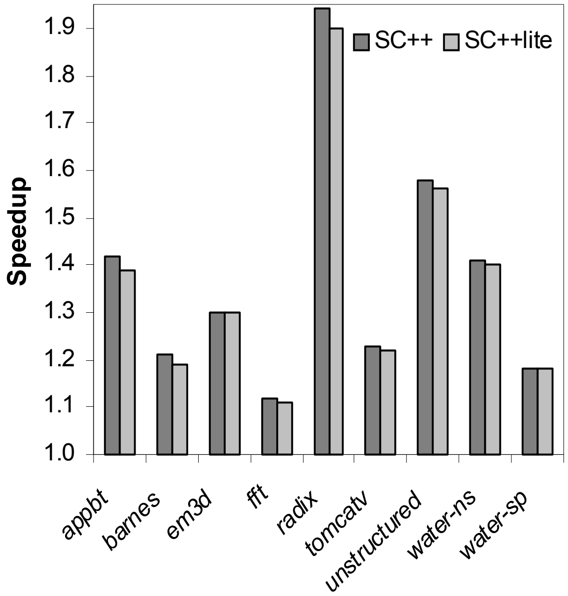

FIGURE 10. Performance for 4x remote latency.

This figure shows the performance of SC++lite against SC++, normalized to SC for 4x L2 remote fill latency. SC++ uses a 8K-entry custom history queue and SC++lite uses a 8-entry SBB.

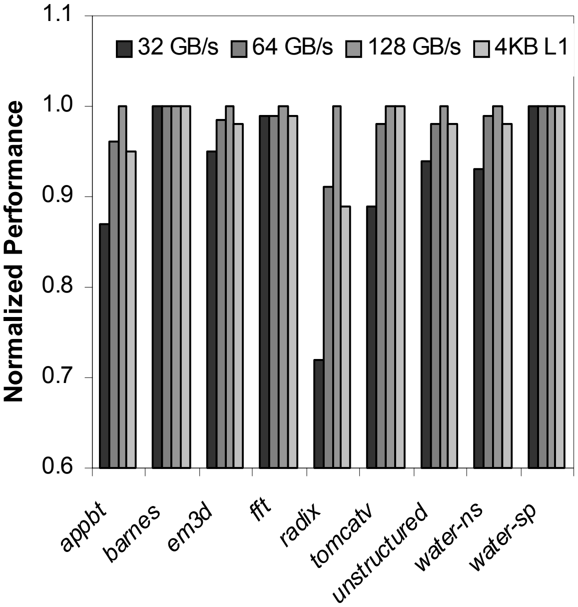

FIGURE 11. Performance sensitivity to varying offered/required L2 bandwidth.

In this figure we compare execution times of SC++ and SC++lite with different L2 bandwidth, normalized to the SC++ with corresponding L2 bandwidth. We also present the impact of 4K L1 on the competition for 64 GB/s bandwidth.

SC++ achieves a speedup of 36% over SC, while SC++lite is able to achieve a 35% speedup over SC with no changes to its base hardware configuration. The key factor contributing to the performance gap, between SC++lite and SC++, is limited L2 bandwidth. The gap can be eliminated by providing an additional L2 port, which we will study in Section 5.4.

5.3 History Storage Requirements

Table 4 compares the storage requirements for SC++ and SC++lite. The extra storage required in speculative SC systems comes from three sources: the history queue (in case of SC++) or the SBB (in case of SC++lite), the lookup table, and the pointers to the history queue embedded in the load/store queue. The entries in the history queue and the SBB are identical and are 100 bits in size. The entry size is dominated by store instructions which have the largest storage requirements. The storage for a store instruction includes 64 bits of modified data to record the old memory value, 32 bits of store address, 1 bit to distinguish store from other instructions, 2 bits to encode the store size, and 1 bit to distinguish between pending and completed store.

Structure	SC++ (KB)	SC++ 4x fill (KB)	SC++lite (KB)
Custom history storage (history queue or SBB)	50.0	100.0	0.1
Lookup table	1.2	1.2	1.3
History pointers in load/store queue	0.2	0.2	0.3
Total	51.4	101.4	1.7

Table 4: Custom storage requirements.

	appbt	*barnes*	*em3d*	*fft*	*radix*	*tomcatv*	*unstructured*	*water-ns*	*water-sp*
Increase in L2 traffic	663%	45%	119%	40%	852%	13%	68%	86%	16%

Table 5: Fraction of traffic increase in L2 due to history.

Each lookup table entry includes 26 bits of block address and a dirty bit used in optimizing and eliminating rollbacks upon downgrade rather invalidation requests. The number of pointer bits in the lookup table and the load/store queue vary depending on the queue sizes. Our base SC++ and our SC++ system with 4x remote fill latency use 4K- and 8K-entry custom queues and therefore require 12 bits and 13 bits for queue pointers respectively. We aggressively assume a memory queue of 64K entries for SC++lite and therefore require 16 bits for pointers.

Table 4 indicates that the total custom storage for SC++, in the base system and the system with 4x L2 remote fill latencies are 51.4KB and 101.4KB respectively. SC++lite, however, reduces the custom storage requirement by an order of magnitude and only requires 1.7KB custom storage for all studied L2 remote fill latencies.

5.4 L2 Bandwidth Requirements

Figure 11 compares the performance of SC++lite under varying L2 bandwidth, affecting the SBB spill rate. In the base system, which corresponds to the 64 GB/s bandwidth available in Pentium 4, the SBB can spill, on average, five history entries per cycle. Decreasing the L2 bandwidth to 32 GB/s, results in the SBB being able to retire, on average, only 2.5 history entries per cycle, effectively reducing the graduation rate. Doubling the L2 ports results in the SBB being able to spill ten history entries per cycle and eliminates the L2 bottleneck seen by the SBB.

The performance gap between SC++ and SC++lite becomes significant, in some applications, for bandwidth of 32 GB/s. The SBB can only send on average 2.5 instructions per cycle into L2 which is 31% of the history throughput in SC++. For the base 64GB/s system, SC++lite is 2% slower, on average, than SC++ and it reaches maximum of 9% for *radix*. In the system with 32 GB/s bandwidth, the SC++lite is, on average, 18% slower than SC++, and 50% slower in case of *radix*. Even with 32 GB/s bandwidth the SC++lite is on average 18% faster than SC. Increasing the L2 bandwidth to 128 GB/s eliminates the performance gap between SC++ and SC++lite. This increase in the L2 bandwidth does not result in performance gains for SC++, and therefore an additional L2 port cannot be justified, given the studied applications. Future processors will provide 128 GB/s or higher bandwidth and therefore the SC++lite will result in the best performance and resource utilization.

Figure 11 also shows the impact of 4KB L1 on the performance of SC++lite. By reducing the L1 size to 4KB, the competition for the L2 bandwidth between the SBB and L1 requests increases. Since the L1 misses are handled before the SBB requests, the available L2 bandwidth for the SBB is reduced. As a result, the performance gap between SC++lite and SC++ increases, but only by less than 1%, on average, as compared to the base system. The insensitivity to the L1 cache size can be explained by dependence between the L1/L2 traffic and the amount of generated history. When the L1/L2 traffic is low, the processor hits in L1 and therefore is generating history at a full rate in a presence of a pending store, but in this case the SBB has the entire L2 bandwidth available. On the other hand, when the L1/L2 traffic is very high the available L2 bandwidth for the SBB is lower, but because the processor is missing in L1, it stalls for the L1 read misses and as a result the amount of generated history

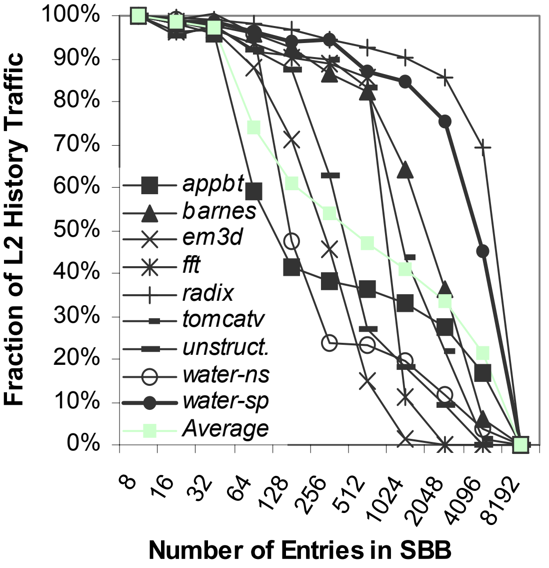

FIGURE 12. History traffic sensitivity to SBB size.

This figure plots the normalized L2 history traffic that was generated for each SBB size.

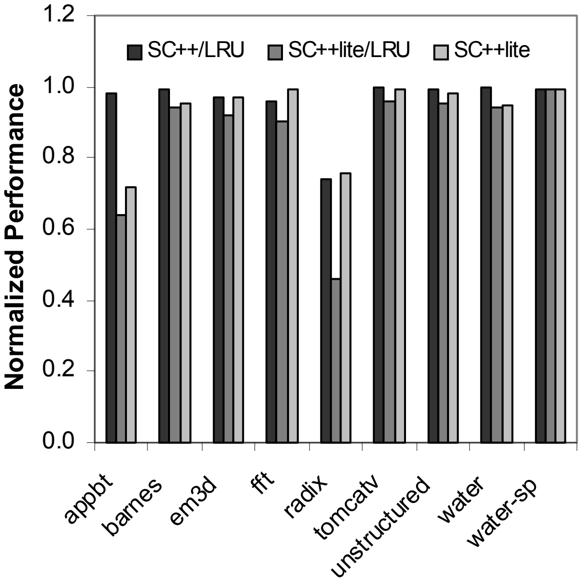

FIGURE 13. Performance sensitivity to L2 replacement policy.

This figure studies the competition for L2 space between data and history. We reduce L2 to 64KB and compare execution time of SC++ and SC++lite with standard LRU (/LRU) and also SC++lite, normalized to SC++ with a 64KB L2.

is small.

Table 5 table shows the relative increase in L2 traffic. The traffic increases only by 13% in *tomcatv*, while in *radix* it increases by over a factor of 9 of the base-case traffic, and on average triples for the studied applications. Table 5 shows that *appbt* and *radix* result in a largest increase in traffic. Although, the increase in L2 traffic in these applications is substantial, Figure 11 indicates that the overall impact on execution time is small. Because these application exhibit low L1 data miss rates, there is little L2 data traffic and as such traffic increase due to history does not impact performance. In contrast, when using 4KB L1 data caches, these applications exhibit lower performance because the increased history traffic competes for L2 bandwidth with L1 data misses. Other applications show moderate increases in traffic and therefore are less impacted by the smaller L1 cache.

5.5 Bandwidth & Performance Sensitivity to SBB Size

SBB contends for L2 bandwidth with L1 i-cache and d-cache. Because history size also varies within an application, SC++lite's L2 bandwidth requirements should vary with SBB size. The SBB, in a sense, can also act as a filter for speculative history. Figure 12 shows the impact the SBB size on the amount of history traffic sent to L2 in SC++lite. The figure plots normalized history traffic for each application. Similarly to Figure 5, there is a large variability in SBB size on L2 history traffic. We observe that almost all applications require 8K SBB entries to eliminate the L2 history traffic entirely. This observations confirms our earlier conclusion that when history is generated it is usually long and requires large amount of storage that SC++ provided in hardware.

We have to take into consideration the magnitude of the traffic increase shown in Table 5. The table shows the relative increase in L2 traffic for the 8 entry SBB. Therefore,

	appbt	*barnes*	*em3d*	*fft*	*radix*	*tomcatv*	*unstructured*	*water-ns*	*water-sp*
Fraction of best performance with 8-entry SBB	97%	97%	99%	100%	98%	99%	99%	99%	100%
SBB entries required for best performance	4K	64	16	8	4K	4K	1K	512	64

Table 6: Performance sensitivity to SBB size.

traffic increase only by 13% in *tomcatv*, while in *radix* it increased by 852% which is 9.5 times the traffic of the base case. On average the traffic tripled for the studied applications.

Table 5 shows that *appbt* and *radix* result in a largest increase in traffic for the base 8-entry SBB. This increase in magnitude would suggest a serious performance penalty but our study shows only 3% performance drop for *appbt* and 2% drop for *radix.* The conclusion is that these applications are able to fit their working set in L1 and therefore there is virtually no L2 data traffic. The conclusion is confirmed by Figure 11 that shows the impact of smaller L1 on the SC++Lite's performance. The 4KB L1 in those two applications resulted in degraded performance since the huge history traffic had to compete for the L2 bandwidth with data accesses. Other applications show moderate increase in traffic and therefore are not impacted by the smaller L1 cache.

Table 6 depicts performance with an 8-entry SBB as a fraction of performance achieved with an infinite-sized SBB, and the number of SBB entries required to achieve the best performance. The table indicates that there is little performance sensitivity to SBB size because: (1) the history is highly bursty with little stead-state L2 traffic due to history, and (2) the overall L2 bandwidth demand from is L1 is low and as such the average steady-state increase in L2 traffic has little overall impact on performance. The table also indicates that eliminating the slight performance degradation due to history traffic in most applications requires a substantial increase in SBB size.

5.6 L2 Replacement Policy

Figure 13 presents the impact of L2 contention on the performance of SC++lite. On average, SC++ improves performance over SC by 34%, while SC++lite improves performance by 30%. SC++lite stores a significant amount of history in L2 and therefore the potential of replacing the data that is currently or subsequently needed by the processor increases for the 64KB L2. The history size is not directly dependent on the L2 size, as long as the application is able to fit its working sets. In this case, the same history that resides in a 512KB L2 has to fit in the 64KB L2. Section 5.1 shows that history can grow and reach up to 4K entries, which requires 50KB of storage. 50KB corresponds to 10% of the 512KB L2, therefore the performance impact is limited, but in the case of the 64KB L2, the history can potentially occupy the entire cache, which can result in a competition for storage and resulting rollbacks.

The history-data competition for storage in L2 causes rollbacks that are not necessary, according to the SC requirements. They are required because SC++lite as well as SC++ is not able to monitor the blocks for mis-speculations after the replacement. After the rollback, the processor incurs stalls due to read miss on the replaced data. Figure 13 shows impact of a standard LRU replacement policy on the performance of SC++lite/LRU and also SC++/LRU.

The LRUNS policy, used throughout the experiments, results in reduction of replacement rollbacks and increase in the performance of SC++ and SC++lite. A significant performance increase is only present in *radix*, and the benefits diminish when more realistic cache sizes are used. Therefore, the implementation of LRUNS mechanisms in SC++ is not justified. SC++lite, on the other hand, has most of interconnect between the BLT and L2 already present and the implementation of the LRUNS replacement policy is a viable solution that provides higher performance with only small increase in design complexity.

The LRUNS replacement policy improves performance by preventing rollbacks due to cache conflicts, but it is not able to prevent replacements caused by capacity misses, which is observed in *appbt*. In case of *radix,* the performance gap is still due to limited L2 bandwidth. *Appbt,* on the other hand, is still exposing significant store latency which results in a large history replacing useful data. A smaller cache can also reduce the performance difference between SC++ and SC++lite, as observed in the remaining applications, by increasing the stall time due to load misses, which also reduces amount of generated history.

SC++lite is able to perform well even for relatively small caches. Moreover, the history is created and retired relatively fast as compared to cache operations, resulting in limited writebacks. The time that history is current corresponds to the time in which the system will service the pending store that created the history. Once the store completes, the history can be discarded. This short duration of history in L2 results in no significant history writebacks to memory and therefore will not create performance bottlenecks in the memory subsystem.

6. Conclusions

This paper proposed *SC++lite*, a system that uses limited custom hardware to speculatively relax memory order while maintaining Sequential Consistency's (SC's) memory order semantics. SC is attractive from a programming perspective because it obviates the need for programmers to annotate memory accesses in programs and enforce memory order through software. Prior research has shown that relaxing memory order speculatively can allow SC systems to achieve the performance of systems that relax order through software annotation. Unlike previous proposals for speculative implementations of SC, SC++lite is a low-overhead implementation that fully realizes the benefits of speculation across a wide range of applications and system latencies while requiring little custom storage. SC++lite uses the memory hierarchy on each processor to store a history of the modified processor/memory state during speculation.

We used cycle-accurate simulation of shared-memory multiprocessors running scientific and engineering applications to compare SC++lite's performance and storage requirements against SC++. We presented a detailed characterization of SC++ history information and corroborate that: (1) queue requirements drastically vary between 16 to 8192 entries for applications and systems we studied, and (2) the history information is quite bursty, on average leaving the queue empty 85% of the application execution time. Our results indicated that SC++lite on average performs 26% better than SC with *only* 1.7KB of storage. In contrast, SC++ requires 51.4KB to achieve a 29% average speedup over a base SC system. Moreover, SC++lite's performance relative to SC++ remained unchanged with a four times increase in memory latencies, while SC++ storage requirements almost double to 101.4KB. Due to the bursty nature of the history, our results indicated little interference with processor's L1/L2 traffic even when using small L1 caches.

Acknowledgments

We would like to thank the anonymous referees for their feedback on the submitted draft of this manuscript. This work was partially funded through an NSF CAREER award, an NSF Instrumentation award, and grants from Intel and IBM.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Intel Corporation. The Intel Pentium 4 processor. In *Product Overview at www.intel.com/design/Pentium4/prodbref/index.htm*, 2002.

[3] Kattamuri Ekanadham, Beng-Hong Lim, Pratap Pattanaik, and Marc Snir. Prism: An integrated architecture for scalable shared memory. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, February 1998.

[4] Babak Falsafi and David A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[5] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–355–364, August 1991.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[7] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of alphaserver GS320. In *Proceedings of the Nineth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[8] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[9] Erik Hagersten and Michael Koster. WildFire: A scalable path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[10] Mark D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8), August 1998.

[11] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[12] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[13] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[14] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Third Workshop on Computer Architecture Education*, February 1997.

[15] Parthasarthy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[16] J. E. Smith and A. R. Plezkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

[17] SPEC. SPEC benchmark suite release 1.0, Winter 1990.

[18] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[19] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.