In Proceedings of the 34th Annual International Symposium on Computer Architecture, June 2007

Mechanisms for Store-wait–free Multiprocessors

Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi and Andreas Moshovos†
http://www.ece.cmu.edu/~stems

Computer Architecture Laboratory (CALCM)
Carnegie Mellon University

†Dept. of Electrical & Computer Engineering
University of Toronto

Abstract

Store misses cause significant delays in shared-memory multiprocessors because of limited store buffering and ordering constraints required for proper synchronization. Today, programmers must choose from a spectrum of memory consistency models that reduce store stalls at the cost of increased programming complexity. Prior research suggests that the performance gap among consistency models can be closed through speculation—enforcing order only when dynamically necessary. Unfortunately, past designs either provide insufficient buffering, replace all stores with read-modify-write operations, and/or recover from ordering violations via impractical fine-grained rollback mechanisms.

We propose two mechanisms that, together, enable store-wait–free implementations of any memory consistency model. To eliminate buffer-capacity–related stalls, we propose the scalable store buffer, which places private/speculative values directly into the L1 cache, thereby eliminating the non-scalable associative search of conventional store buffers. To eliminate ordering-related stalls, we propose atomic sequence ordering, which enforces ordering constraints over coarse-grain access sequences while relaxing order among individual accesses. Using cycle-accurate full-system simulation of scientific and commercial applications, we demonstrate that these mechanisms allow the simplified programming of strict ordering while outperforming conventional implementations on average by 32% (sequential consistency), 22% (SPARC total store order) and 9% (SPARC relaxed memory order).

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Design, Performance

Keywords

Memory consistency models; store buffer design

ISCA'07, June 9–13, 2007, San Diego, California, USA.

1 Introduction

Store misses pose a significant performance challenge on shared-memory multiprocessors [7]. In uniprocessors, store miss latency can be hidden with small, simple store buffers that allow other accesses to proceed past store misses [3].[1] Unfortunately, in multiprocessor systems, application correctness often depends on the precise order of accesses to distinct memory locations (e.g., acquiring a lock before accessing a shared variable) [18]. To enable programmers to reason about the correctness of shared memory accesses, multiprocessor system designers have developed elaborate programming models, called memory consistency models, which establish minimal ordering guarantees that hardware must enforce [1]. To implement these guarantees, conventional processors often stall other accesses while in-flight store misses complete.

Today, programmers must choose from a spectrum of consistency models that trade increasing programming complexity for hiding more store-related stalls. At one extreme, strictly-ordered consistency models provide the most intuitive programming interface, but current implementations expose substantial store miss latency. At the other extreme, relaxed consistency models allow unrelated accesses to proceed past store misses, but require programmers to explicitly annotate accesses where precise ordering is required. Ideally, we would like to provide programmers with the simplest programming models and yet, as in uniprocessors, never stall as a result of store misses [16]. We call such designs *store-wait–free multiprocessors*.

Store delays arise from two sources: (1) insufficient store buffer capacity during store bursts, and (2) ordering constraints that stall execution until all in-flight stores complete. Under strictly-ordered consistency models, ordering stalls dominate, as loads may not retire if a store miss is outstanding. Even in aggressive implementations that hide some stalls via prefetching and speculative load execution [10], substantial stalls remain [7,11,13,26,27], accounting for nearly a third of execution time in commercial applications [25]. In models that relax load-to-store order [7], the key bottleneck shifts to capacity stalls. Conventional store buffer capacity is limited because every load must associatively search the buffer for a matching store, and this search is on the load's critical path.

1. There is much confusion regarding the terms "store queue," "store buffer," and "write buffer." We use "store queue" to refer to storage that holds stores' values prior to retirement and "store buffer" to refer to storage containing retired store values prior to their release to memory. We avoid the term "write buffer."

Fully-relaxed models alleviate store buffer capacity pressure by allowing stores to drain in any order, but still incur substantial ordering delays at synchronization instructions [7,13,25,36].

To bridge the performance gap among consistency models, researchers have proposed systems that speculatively relax ordering constraints and/or retire instructions past synchronization operations [12,13,16,21,23,24,27,36]. If a memory race exposes an ordering violation, these systems roll back execution to the violating instruction. Speculation improves performance because races are rare in well-behaved concurrent applications, and hence most ordering delays are dynamically unnecessary. Unfortunately, existing designs address only specific classes of synchronization stalls [21,23,24,36], provide insufficient speculation depth [27], replace all stores with read-modify-write operations [12,13], and/or implement impractical fine-grained rollback mechanisms that recover precisely to the violating instruction [12,13].

We propose two mechanisms to address limited store buffer capacity and relax ordering constraints with practical hardware. Together, these mechanisms enable store-wait–free implementations of any consistency model.

We eliminate store-buffer-capacity stalls through the *scalable store buffer* (SSB). The SSB eliminates the non-scalable associative search of conventional store buffers by forwarding processor-visible/speculative values to loads directly from the L1 cache while maintaining globally-visible/committed values in L2. The SSB maintains store order in a FIFO structure that does not require an associative search, but enables a properly-ordered commit to L2.

We eliminate ordering-related stalls through *atomic sequence ordering* (ASO). Rather than enforce ordering constraints on each memory access, ASO enforces ordering at coarse granularity over access sequences that appear atomic to other processors. If a data race exposes an ordering violation, ASO recovers to the start of the nearest atomic sequence. Because it uses coarse-grain rollback, ASO can be implemented with efficient checkpoint-based rollback mechanisms [2,8,20]. ASO leverages the SSB to provide sufficient speculation depth and recover correct memory state using the ordered FIFO in the event of a violation.

With these mechanisms, we construct store-wait–free implementations of sequential consistency (SC) and SPARC total store order (TSO). We compare these to a conventional implementation of SPARC relaxed memory order (RMO). Using cycle-accurate full-system simulation of scientific and commercial multiprocessor applications, we demonstrate:

- **Substantial store stalls under all consistency models.** In even the most aggressive conventional implementations, we observe an average of 31% of execution time on store-related stalls under SC, 16% under TSO, and 14% under RMO.
- **Scalable buffering bridges the TSO-RMO gap.** We demonstrate that the SSB provides store buffer capacity exceeding application demands, closing the performance gap between TSO and RMO. The SSB improves performance by as much as 45% over conventional TSO.
- **Store-wait–free implementations.** ASO further eliminates dynamically unnecessary ordering constraints, including synchronization stalls. Our mechanisms enable SC or TSO guarantees while outperforming RMO by 6% to 9%.

The remainder of this paper is organized as follows. In Section 2, we provide motivation and high-level descriptions of our mechanisms. We present hardware details in Section 3 and evaluate our designs in Section 4. In Section 5, we discuss related work. We conclude in Section 6.

2 Store-wait–free mechanisms

Our goal is to design multiprocessor systems that, like uniprocessors, never incur store-related stalls. We begin by analyzing the causes of store-related stalls in conventional implementations of various memory consistency models.

2.1 Store stalls in conventional systems

In conventional systems, stores stall instruction retirement for two reasons: (1) capacity stalls—a store instruction may not retire because no store buffer entries are free, and (2) ordering stalls—a load, fence, atomic read-modify-write (RMW), or synchronization access may not retire until all prior stores complete. The precise conditions that lead to these stalls vary across consistency models. Figure 1 illustrates these conditions under three broad classes of consistency models [1,16], and identifies the name for each stall category used in our analysis.

Strongly-ordered consistency models. Under strongly-ordered models (e.g., sequential consistency (SC)), retirement must stall at the first load following an outstanding store miss. Hence, such systems expose nearly the entire latency of store misses on the execution critical path [11,13,26,27]. These systems may contain a small store buffer to manage L1D port contention [3], but increasing buffer capacity provides no benefit, because few stores occur between consecutive loads. Store prefetching and speculative load execution improve performance drastically under these models, as they allow several misses to be serviced concurrently [10,26], but remaining store stalls still account for up to a third of execution time [25].

Models that relax load-to-store order. Under consistency models that relax load-to-store order (e.g., processor consistency, SPARC total store order (TSO), and Intel IA-32 processor-ordering), conventional implementations buffer all retired stores (including hits) that follow a store miss, but allow loads to bypass these buffered stores. Stores may not retire when the store buffer is full. Store buffer capacity is limited because all loads must search the buffer for a matching store. This search is on the load critical path, and is typically implemented using a non-scalable content-addressable memory (CAM) [2,9]. Consecutive stores to the same cache block may coalesce in the store buffer, but this optimization provides minimal benefits [7]. High demand for store buffer capacity creates a significant performance gap between these and more relaxed consistency models. Although they do not stall retirement of loads, these systems do stall RMW and memory fence instructions until the store buffer drains.

Models that relax all order. Consistency models that fully relax ordering (weak ordering, release consistency, SPARC relaxed memory order (RMO), Intel IA-64, HP Alpha, and IBM PowerPC) alleviate store buffer pressure, as buffer entries may be freely coalesced and committed in any order. However, memory fence instructions and/or annotated synchronization accesses still stall

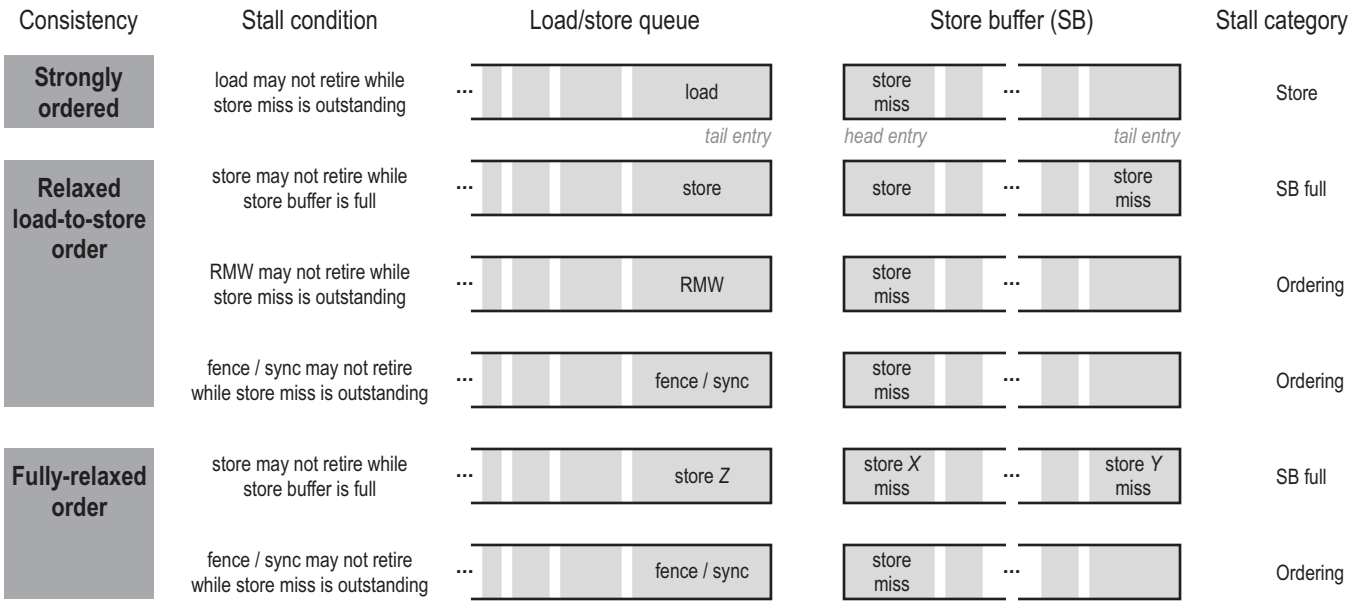

FIGURE 1. Store-related stalls under various classes of consistency models.

instruction retirement until the store buffer drains. Because of the frequent thread synchronization in commercial applications, these ordering delays are significant [7]. When no data race occurs, these stalls are unnecessary [13,36].

Throughout the remainder of this paper, we focus our study on the SC, TSO and RMO consistency models. However, our mechanisms also apply to other consistency models.

2.2 Stalls on RMWs

Atomic read-modify-write instructions have the semantics of both loads and stores. When an RMW incurs a cache miss, it stalls retirement much like a load miss, which neither of our mechanisms address. However, in most cases, RMWs are used to acquire locks. In the absence of contention, these locks are usually available and the RMW read stall is dynamically unnecessary. In Section 4.6, we consider value prediction as an approach to avoid RMW read stalls, allowing our other mechanisms to hide the write latency of the RMW.

2.3 Scalable store buffering

We propose the *scalable store buffer* (SSB) to eliminate the capacity constraints of conventional store buffers. By providing capacity exceeding application needs, the SSB closes the performance gap between the TSO and RMO consistency models. Furthermore, it provides a scalable buffering solution on which we can build more elaborate mechanisms to achieve store-wait–free execution.

Conventional store buffers perform two largely independent functions under TSO: (1) they forward values to matching loads, and (2) they maintain order among stores for memory consistency. The key idea of the SSB is to perform value forwarding through L1, eliminating the need for the conventional store buffer's CAM search. To maintain consistency, stores are tracked in a FIFO structure that does not support associative search, called the Total Store Order Buffer (TSOB). Stores drain from the TSOB into L2 in order, and coherence requests return values from L2. Hence, L1 contains CPU-private values, while globally-visible (consistent) values are maintained in L2 and beyond.

The SSB effectively increases store buffer capacity to the L1's capacity (with a victim buffer to address conflicting writes). Although this capacity is sufficient for all the applications we study (see Section 4.3), if a particular application does overflow the L1, the SSB ensures correct execution by stalling until stores drain and capacity is freed.

In the rare event that two processors simultaneously update a cache block (e.g., contention), a block cached in L1 may be partially invalidated—that is, CPU-private writes (uncommitted stores still in the TSOB) must be preserved, but the globally-visible portion of the cache line is invalidated. To enforce TSO, the memory system must merge the uncommitted stores with any remote updates. In conventional TSO systems, private values remain in the store buffer, which ignores invalidations, while the remainder of the cache block is discarded from L1. In designs with an SSB, we construct the correct value of a partially-invalidated line by reloading the affected line from the memory system and then replaying uncommitted stores to the line from the TSOB. (Note that this replay mechanism does not roll back execution.)

2.4 Atomic sequence ordering

We propose *atomic sequence ordering* (ASO) to address the ordering-related stalls under conventional implementations of any consistency model. ASO dynamically groups accesses into *atomic sequences*, and provides coarse-grain enforcement of ordering constraints over these sequences while relaxing ordering among individual accesses. To meet consistency requirements, ASO ensures that each sequence appears atomic to other processors. As long as an access sequence appears atomic, the actual order of individual accesses is not observable by other CPUs and, hence, does not violate ordering constraints under any consistency model. In particu-

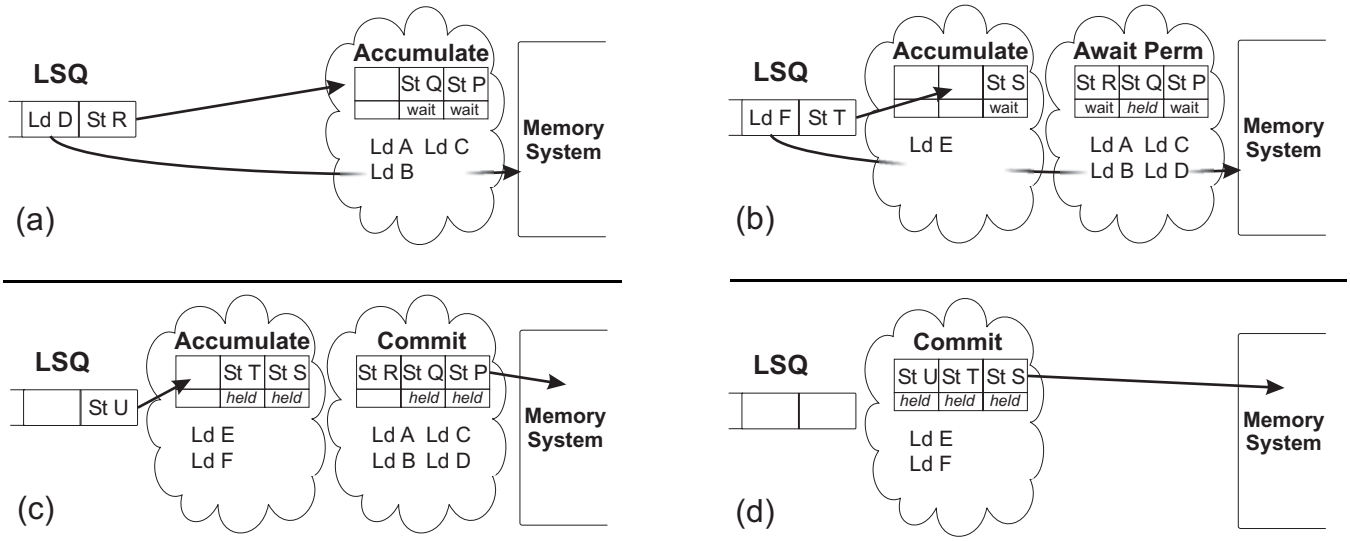

FIGURE 2. Atomic sequence ordering.

lar, ASO allows strictly ordered models (e.g., SC), to benefit from store buffering. Moreover, atomic sequences eliminate dynamically-unnecessary ordering stalls at the synchronization instructions required by fully-relaxed models.

As execution proceeds, atomic sequences either commit or are discarded if a race for one of the locations accessed in the sequence exposes an ordering violation. Because races are rare in well-behaved parallel applications [13,21,23,27], atomic sequence ordering improves performance.

Operation under ASO. Whenever an ordering constraint would stall instruction retirement, the system initiates ASO. Figure 2 illustrates the lifetime of two atomic sequences.

Each atomic sequence proceeds through three states. (a) ASO creates a processor checkpoint and begins a new atomic sequence, which is initially in the *accumulate* state. Loads, stores, RMWs and memory fence instructions retire speculatively, ignoring ordering constraints. As execution proceeds, retiring stores and loads are accumulated into the sequence. The hardware requests write permission for all cache blocks updated by the sequence. (b) When an atomic sequence reaches a predetermined size, the hardware creates a new checkpoint and begins a new atomic sequence. Meanwhile, the first sequence transitions to the *await permission* state, and waits for all write permissions to arrive. (c) Once all write permissions arrive, the first atomic sequence transitions to the *commit* state, and drains its writes to the memory system. Because the local processor holds write permission for all lines written by the sequence, it can assure commit atomicity by delaying requests from other processors until the commit is complete. The sequence is committed once all its writes are globally visible. (d) In this example, the second sequence transitions directly from accumulate to commit state, because it already holds all required permissions.

If another processor writes an address speculatively read by an atomic sequence (i.e., a data race), that and all later sequences are discarded and the processor state is rolled back to the start of the appropriate sequence. To guarantee forward progress, a processor must commit at least one store before initiating another sequence.

Implementation requirements. An ASO implementation must fulfill three requirements. First, it must detect when a read in an atomic sequence returns a value in violation of memory consistency constraints. Like many prior speculative techniques [19,23], we track speculatively-read blocks in the cache hierarchy and use the existing coherence notifications to detect conflicting writes. Second, when a violation occurs, the hardware must rewind execution and recover the correct processor state (i.e., register values, TLB/MMU, etc.) as of the start of the violating sequence. We apply previously-proposed register checkpointing techniques [2,8,20] to meet this requirement. Finally, the hardware must buffer all of a sequence's writes and either release them atomically upon commit or discard them upon violation. We use the SSB (Section 2.3) to address this requirement.

The SSB handles speculative writes (under any consistency model) much like the private writes that occur under TSO. An atomic sequence corresponds to a contiguous sequence of entries in the SSB's total store order buffer. When the atomic sequence commits (i.e., when all required write permissions have arrived), its stores drain from the TSOB into L2. To ensure atomicity, the L2 cache stalls external coherence requests that might intersect the sequence's reads or writes during commit.

In the infrequent case that some, but not all, outstanding sequences violate and must roll back, the L1 discards all speculatively-written lines and discarded sequences are erased from the TSOB. Then, TSOB entries from the remaining sequences are replayed into the L1 cache to reconstruct the cache state at the time of the checkpoint (much like the invalidation replay mechanism described in Section 2.3). This TSOB replay mechanism is necessary to reduce the performance impact of repeated violations in pathological cases of contention—older sequences (that do not access the contended address) continue to acquire permissions and make progress towards commit. If a violation rolls back the eldest outstanding

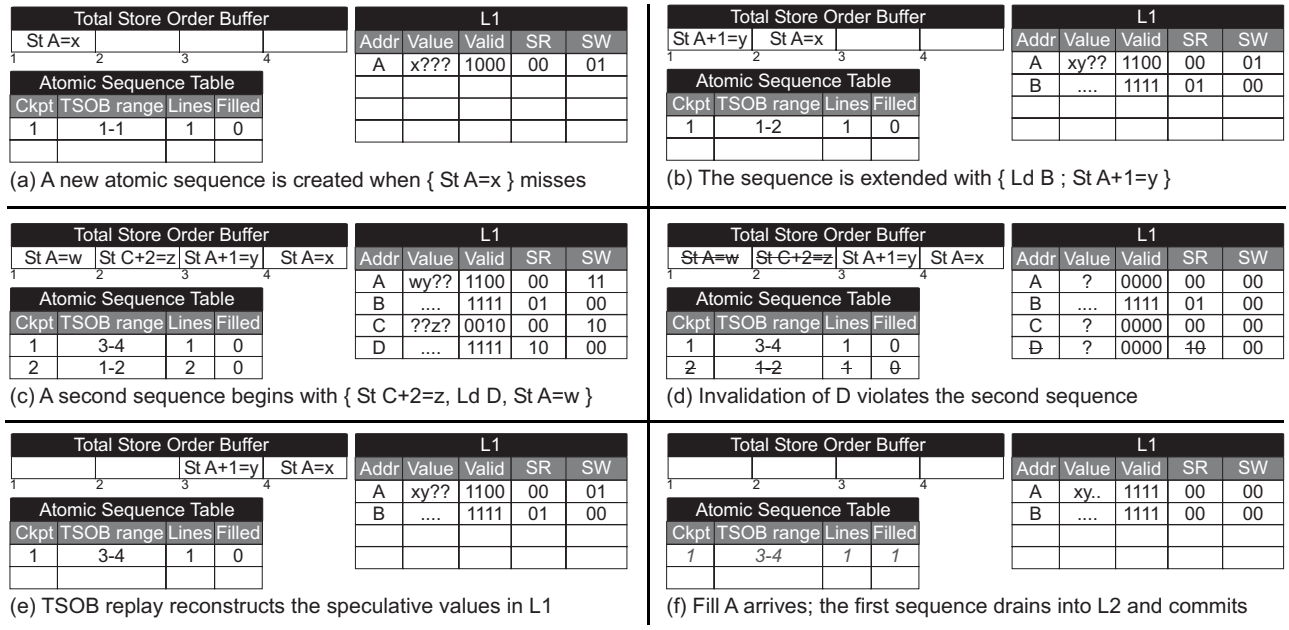

FIGURE 3. Example of store-wait–free hardware operation.

sequence, replay is unnecessary (i.e., all speculative writes are discarded).

3 Hardware implementation

In this section, we present the hardware required to implement our mechanisms. We organize our presentation around the major functionality each mechanism requires. We illustrate the contents of new structures and cache modifications in Figure 3, along with an example of operation, described in Section 3.3.

3.1 SSB implementation

Scalable store buffering for conventional TSO. As store instructions retire, they update the L1 data cache, and are appended to the *Total Store Order Buffer* (TSOB). We augment each L1 line with per-word "valid" bits much like a sub-blocked cache. These bits allow the processor to read private data while a store miss is outstanding and facilitate merging with incoming data when the line is filled. The TSOB records the address, value, and size of all stores in a RAM structure organized as a circular buffer. Because it does not require associative search, we can roughly match TSOB capacity to L1 capacity (1024 entries, or about 10 KB of storage, in our experiments). A 16-entry victim buffer addresses conflicting stores.

Handling invalidations. If an invalidation matches a line with any of the per-word valid bits set, the line may be partially invalidated (i.e., another processor is writing part of the line). All valid bits are cleared and a new request for write permission and the updated value is issued. When the memory system returns the updated line, the SSB reconstructs the correct value of the line by traversing the entire TSOB to replay stores to the affected line, setting per-word valid bits as appropriate. Although this process is inefficient, the replay is sufficiently rare to cause negligible performance loss.

Handling sub-word writes. Because our design provides valid bits on a per-word rather than per-byte basis, it cannot place writes smaller than a word in the cache unless the line is valid (with or without write permission). We handle sub-word store misses by leaving them in a small (e.g., 8-entry), conventional *mini store buffer* until the line is filled. When multiple sub-word stores write the same word (a common case in unaligned memcpys) we create individual entries for each store in the TSOB, but allow the stores to coalesce in the mini store buffer.

3.2 ASO implementation

Tracking atomic sequences. We initiate atomic sequence ordering when an ordering constraint prevents instruction retirement because a store is outstanding (i.e., the SSB is not empty and a load reaches retirement under SC, or an RMW or fence reaches retirement under TSO). The *Atomic Sequence Table* (AST) tracks active atomic sequences. For each sequence, the AST records the corresponding range of TSOB entries and two additional counters. "Lines" counts the distinct cache lines written by the sequence. "Filled" counts lines for which write permission has been obtained. A sequence may commit when all permissions arrive ("Filled" equals "Lines"). We augment each L1 line with a speculatively-written ("SW") bit per supported sequence to indicate which sequences' "Filled" counters to update when the line arrives. Because violations are rare, we have found that there is little sensitivity to the precise heuristics used to allocate a new atomic sequence. Shorter sequences enable faster commit, but require hardware support for more active sequences to achieve store-wait freedom. We allocate a new sequence whenever the current sequence's "Lines" count reaches 16, allowing rapid sequence commit while requiring support for few (~4) sequences.

Detecting atomicity violations. To track the speculative reads in each atomic sequence, we augment each L1 and L2 cache line with per-sequence speculatively-read ("SR") bits. If an invalidation

message matches a speculative line, or the line is evicted from L2, the corresponding atomic sequence must roll back.

Committing atomic sequences. When all permissions arrive, the TSOB range corresponding to a sequence is drained into the L2 cache. To assure atomicity, external requests are delayed during the commit process. Both the "SR" and "SW" bits corresponding to the sequence are cleared via a bulk-clear operation [20]. To support the bulk operations, these fields may be implemented in a specialized array rather than augmenting the existing cache tag array. For the cache sizes we study, these additional bits incur ~3% storage overhead in L1, and less than 1% in L2.

Rollback on violation. To support rollback, we require a register checkpointing mechanism that provides one checkpoint for each supported atomic sequence. Several appropriate mechanisms have been proposed [2,8,20]. Because we speculate in supervisory code, the register checkpointing mechanism must record all supervisory registers and recover TLB/MMU state. To correct cache state, all lines with one or more "SW" bits set are bulk-invalidated, and the TSOB ranges corresponding to discarded sequences are erased. Then, the remaining TSOB contents are replayed into L1.

3.3 Hardware operation example

Figure 3 shows an example of the operation of our design.

(a) The hardware collects a CPU checkpoint and allocates a new atomic sequence upon an L1 store miss. The store writes its value into the cache, setting the appropriate valid and "SW" bits, and increments the sequence's "Lines" counter. The cache issues a request for the line.

(b) While the miss is outstanding, execution continues, and loads and stores are added to the atomic sequence. Loads set the sequence's "SR" bit. Stores are appended to the TSOB. In this example, {St A+1=y} updates a line that has already been written by this sequence (the "SW" bit is already set), so the "Lines" counter is left unchanged.

(c) The atomic sequence hardware collects a second checkpoint and allocates atomic sequence 2. Subsequent loads and stores are added to this sequence. Note that {St A=w} overwrites a value produced by sequence 1, which is prohibited in some prior designs [9].

(d) An invalidation arrives for address D, which has been speculatively read in sequence 2, resulting in a violation. Sequence 2's AST and TSOB entries are discarded and the CPU state is recovered to checkpoint 2. All lines that have been speculatively written by any sequence are discarded.

(e) Before execution may continue, the remaining TSOB entries (from sequence 1) are replayed into the L1 cache to reconstruct the values, valid bits, "SW" bits, "Lines" and "Filled" counts for the remaining sequences.

(f) When the fill for address A arrives, it increments the "Filled" count for sequence 1. Because "Filled" equals "Lines", sequence 1 may commit. The sequence's TSOB entries are drained into L2 (all accesses will result in hits), and all resources associated with the sequence are cleared.

4 Evaluation

In the following subsections, we evaluate the effectiveness of our mechanisms relative to conventional implementations.

4.1 Methodology

We evaluate our mechanisms using cycle-accurate full-system simulation of a shared-memory multiprocessor using *FLEXUS* [37]. FLEXUS models the SPARC v9 ISA and can execute unmodified commercial applications and operating systems. FLEXUS extends the *Virtutech Simics* functional simulator with models of an out-of-order processor core, cache hierarchy, protocol controllers and interconnect. We simulate a 16-node directory-based shared-memory multiprocessor system running *Solaris 8*. We implement a low-occupancy directory-based NACK-free cache-coherence protocol. Our system performs speculative load execution and store prefetching (at execute under SC, at retirement under other consis-

TABLE 1. System and application parameters.

Processing Nodes	UltraSPARC III ISA 4 GHz 8-stage pipeline; out-of-order 4-wide dispatch / retirement 96-entry ROB, LSQ 32-entry conventional store buffer
L1 Caches	Split I/D, 64KB 2-way, 2-cycle load-to-use 3 ports, 32 MSHRs, 16-entry victim cache
L2 Cache	Unified, 8MB 8-way, 25-cycle hit latency 1 port, 32 MSHRs
Main Memory	3 GB total memory 40 ns access latency 64 banks per node 64-byte coherence unit
Protocol Controller	1 GHz microcoded controller 64 transaction contexts
Interconnect	4x4 2D torus 25 ns latency per hop 128 GB/s peak bisection bandwidth

Online Transaction Processing (TPC-C)	
DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web Server	
Apache	16K connections, fastCGI, worker threading model
Zeus	16K connections, fastCGI
Decision Support (TPC-H on DB2)	
Qry 1	Scan-dominated, 450 MB buffer pool
Qry 6	Scan-dominated, 450 MB buffer pool
Qry 16	Join-dominated, 450 MB buffer pool
Scientific	
barnes	16K bodies, 2.0 subdiv. tol.
ocean	1026x1026 grid, 9600s relaxations, 20K res., err tol 1e-07

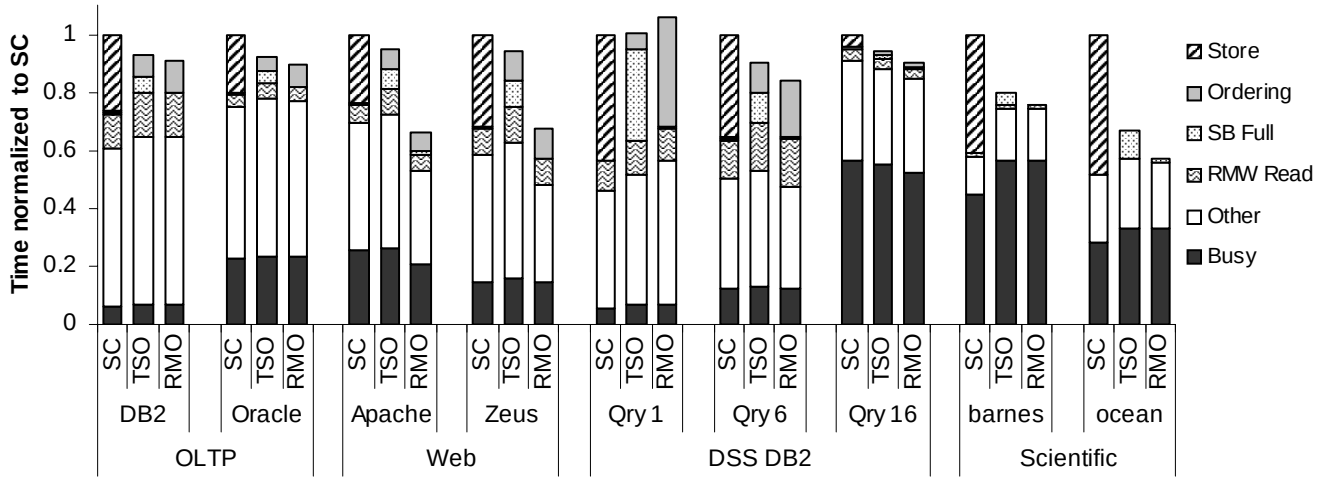

FIGURE 4. Execution time breakdown in conventional systems. Each bar shows an execution time breakdown under conventional implementations of SC, TSO, RMO, normalized to SC performance.

tency models) as described in [7,10]. Hence, our base system is similar to the system described in [25]. We configure our processor model to approximate the hardware resources of the Intel Core 2 microarchitecture. We list other relevant parameters in Table 1 (left).

Table 1 (right) enumerates our commercial and scientific application suite. We include the TPC-C v3.0 OLTP workload on *IBM DB2 v8 ESE* and *Oracle 10g Enterprise Database Server.* We run three queries from the TPC-H DSS workload on DB2, selected according to the categorization of Shao et al. [30]. We evaluate web server performance with the SPECweb99 benchmark on *Apache HTTP Server v2.0* and *Zeus Web Server v4.3*. We drive the web servers using a separate client system (client activity is not included in timing results). Finally, we include two scientific applications as a frame of reference for our commercial application results.

We measure performance using the SimFlex multiprocessor sampling methodology [37]. The SimFlex methodology extends the SMARTS [38] statistical sampling framework to multiprocessor simulation. Our samples are drawn over an interval of from 10s to 30s of simulated time for OLTP and web server applications, over the complete query execution for DSS, and over a single iteration for scientific applications. We launch measurements from checkpoints with warmed caches and branch predictors, then warm queue and interconnect state for 100,000 cycles prior to measuring 50,000 cycles. We use the aggregate number of user instructions committed per cycle (i.e., committed user instructions summed over the 16 processors divided by total elapsed cycles) as our performance metric, which is proportional to overall system throughput [37].

FLEXUS fully supports execution under the SC, TSO and RMO consistency models. Unfortunately, the commercial applications we study include code that requires TSO for correctness. Hence, while we can simulate all applications directly under SC and TSO, and scientific applications under RMO, our commercial applications lack the necessary memory barriers to run under RMO.

To work around this limitation, we dynamically insert memory barrier instructions when simulating our commercial applications under RMO. These barriers stall retirement until all preceding memory operations complete, but allow loads within the instruction window to execute speculatively. Because it is difficult to automatically identify all locations where barriers are required, we err on the side of inserting too few barriers, thus overestimating RMO performance (conservatively underestimating speedups from ASO). We introduce a barrier after each RMW operation (i.e., lock acquire). However, we do not introduce memory barriers at lock release because it is difficult to reliably identify the releasing store. As these barriers are not sufficient to guarantee correctness, we separately track TSO-consistent memory values and confirm that all loads return values that occur under TSO. In most cases, ordering differences between TSO and RMO do not affect the value returned by a load. If a memory ordering difference causes a load value mismatch, the affected load and its dependent instructions are re-executed to assure correct program execution. These mismatches are rare and have a negligible performance impact.

4.2 Store stalls in conventional systems

The frequent store misses in commercial applications result in a significant performance gap between conventional implementations of memory consistency models despite aggressive store prefetching and speculative load execution. Figure 4 shows an execution time breakdown for each application under SC, TSO, and RMO. The total height of each stacked bar is normalized to the application's performance under SC. The stall categories shown in the breakdown are described in Section 2.1.

Under SC, we observe 25%-40% of execution time stalled on stores. Our results corroborate previous results for OLTP [25]. We observe somewhat higher stalls in Qry 1 and Qry 6 because of significant lock contention in these two applications. The performance gap between SC and more relaxed models arises because of the ordering constraints on load retirement; loads stall until in-flight stores complete. Store stalls are an even larger problem without store prefetching; additional experiments (not shown in

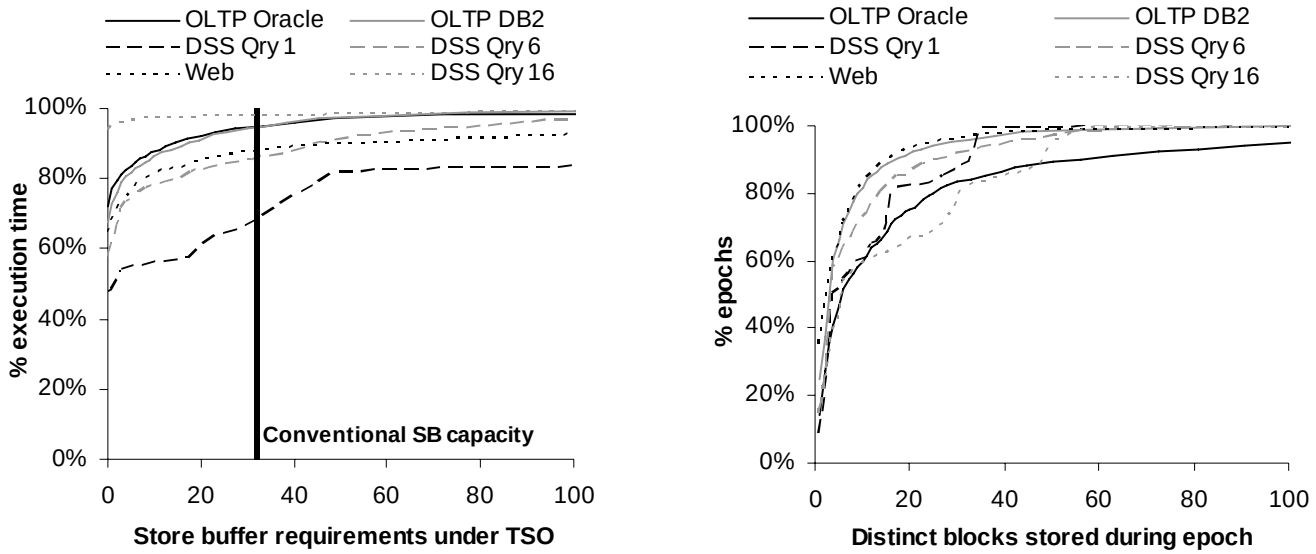

FIGURE 5. Capacity requirements. The left figure shows a cumulative distribution of SB capacity requirements under TSO. The right figure shows the distinct blocks written in the execution epochs between off-chip load stalls.

Figure 4) indicate that store stall cycles nearly double without store prefetching.

Under TSO, store delays manifest as SB full and ordering stalls. The SB full stalls indicate that a 32-entry store buffer is insufficient to hide the store bursts that arise in these applications. We examine the capacity requirements of our applications in more detail in Section 4.3.

As stores may coalesce and retire from the store buffer in any order under RMO, a 32-entry store buffer eliminates the capacity-related stalls. In the web applications, eliminating SB full stalls provides a drastic performance boost; in addition to direct savings of SB full time, the reduced stalls improve the out-of-order core's ability to hide load and RMW read stalls through speculative load execution.

The persistent lock contention in Qry 1 leads to a performance loss under RMO relative to SC. Under high contention, delaying memory requests sometimes improves overall application performance because of faster lock hand-offs.

Comparing RMO to TSO, despite substantial gains from eliminating capacity stalls, ordering stalls at barriers still account for an average of 16% of execution time. We will show in Section 4.5 that many of these stalls are dynamically unnecessary, and can be eliminated by ASO.

Finally, the overall proportion of memory stalls in these applications depends heavily on memory latency—in particular, on the communication latency for coherence activity. In the distributed-shared memory system we study, the latency of coherence misses (for read or write permission) is typically three times that of a local memory access, and more than 75% of off-chip accesses are the result of coherence in our workloads. Hence, the overall store stall time, and the gap among consistency models, will vary directly with memory access and interconnect latency.

4.3 Store buffer capacity requirements

TSO capacity requirements. Under TSO, conventional store buffers provide insufficient capacity during store bursts, resulting in SB full stalls. Figure 5 (left) shows that typical store buffer capacity (limited to ~32 entries because of the CAM search [7]) falls well short of commercial applications' requirements. Two constraints drive the high capacity requirements: (1) stores that will hit in the L1 must remain buffered behind store misses, and (2) overlapping stores cannot coalesce into a single SB entry because of intervening stores to other addresses. The scalable store buffer eliminates both of these constraints, allowing both store hits and misses to write to the L1 immediately. We examine the performance impact of eliminating these stalls in Section 4.4

ASO capacity requirements. Our ASO design requires that speculatively-written data remain in the L1 cache and victim cache. Hence, the capacity of these structures constrains maximum speculation depth.

To assess ASO's speculative-data capacity requirements, we analyze our workloads using the epoch execution model described in [6,7]. This approach models the execution of out-of-order processors with long off-chip access latencies as a series of execution epochs. Each epoch consists of a computation period followed by a long stall on one or more off-chip loads. During the stall period, all outstanding misses complete, and any atomic sequences can commit. Hence, we can estimate ASO's capacity requirements from the number of blocks stored during each epoch. The key advantage of this analysis approach is that its results are independent of precise access latencies and details of the execution core.

We present a cumulative distribution of the distinct cache blocks written during epochs with at least one store in Figure 5 (right). In every case except Oracle, nearly all epochs write fewer than 64 cache blocks, much less than the ~1000-line capacity of our 64KB L1. Oracle exhibits long epochs without an off-chip load miss, as it has been heavily optimized to maintain its primary working set

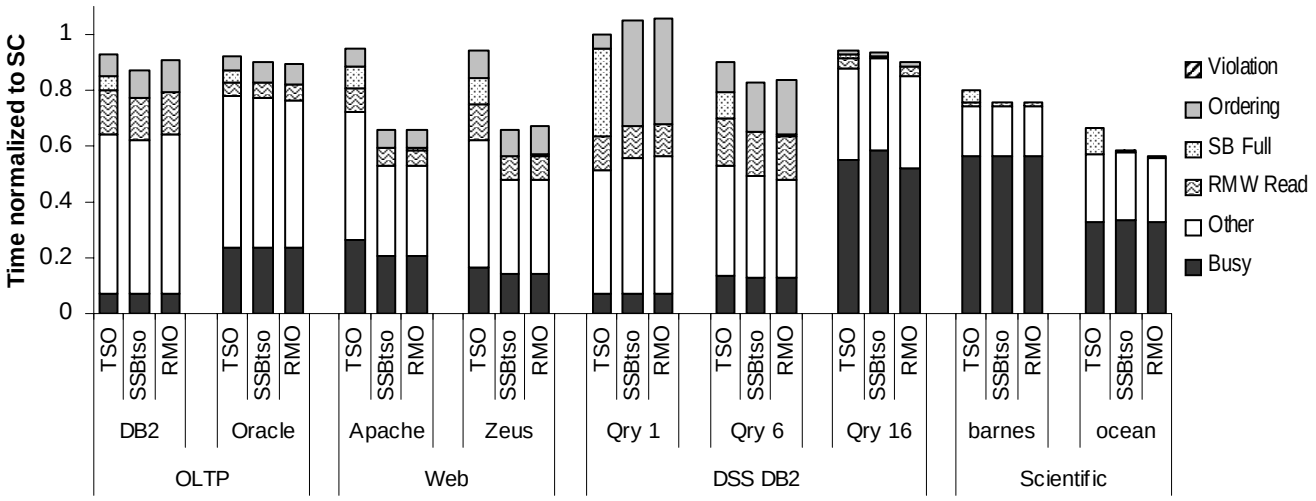

FIGURE 6. Scalable store buffer (SSB) performance impact.

within the 8MB L2. However, these epochs also contain few store misses. Hence, we conclude that L1's capacity is sufficient for ASO.

4.4 Impact of the scalable store buffer

Our capacity analysis indicates that conventional store buffers cannot eliminate SB full stalls. However, the scalable store buffer provides capacity far exceeding application demands. In Figure 6, we examine the performance impact of replacing conventional store buffers with our scalable design.

By itself, the scalable store buffer has no effect on SC or RMO execution. Under SC, ordering constraints at loads result in stalls regardless of store buffer capacity. Under RMO, a conventionally-sized store buffer is sufficient to eliminate capacity-related stalls. However, under TSO, conventional store buffers are insufficient, creating a significant performance gap between TSO and RMO. Adding the speculative store buffer to a TSO system (SSB_{tso}) eliminates all capacity stalls. As a result, SSB_{tso} provides performance similar to RMO.

4.5 Impact of atomic sequence ordering

Atomic sequence ordering eliminates ordering constraints and their associated performance penalties. As Figure 7 shows, adding ASO to SSB_{tso} (labelled ASO_{tso}) improves performance an additional 8%. ASO_{tso} outperforms conventional RMO implementations, which always stall retirement at memory barriers to ensure proper ordering. Under ASO, retirement continues, optimistically ignoring ordering constraints. ASO incurs delays only if an ordering violation is actually observed by another node, in which case execution must roll back. With the exception of high-contention decision support queries (Qry 1 and Qry 6), less than 2% of execution time is lost due to rollback on ordering violations. Although all ordering stalls are eliminated, order speculation sometimes exposes read misses previously hidden behind a memory barrier. Hence, only a portion of ordering stalls are recovered as performance improvement.

Perhaps more importantly, however, ASO eliminates the load retirement constraint under SC, allowing an SC system (ASO_{sc}) to

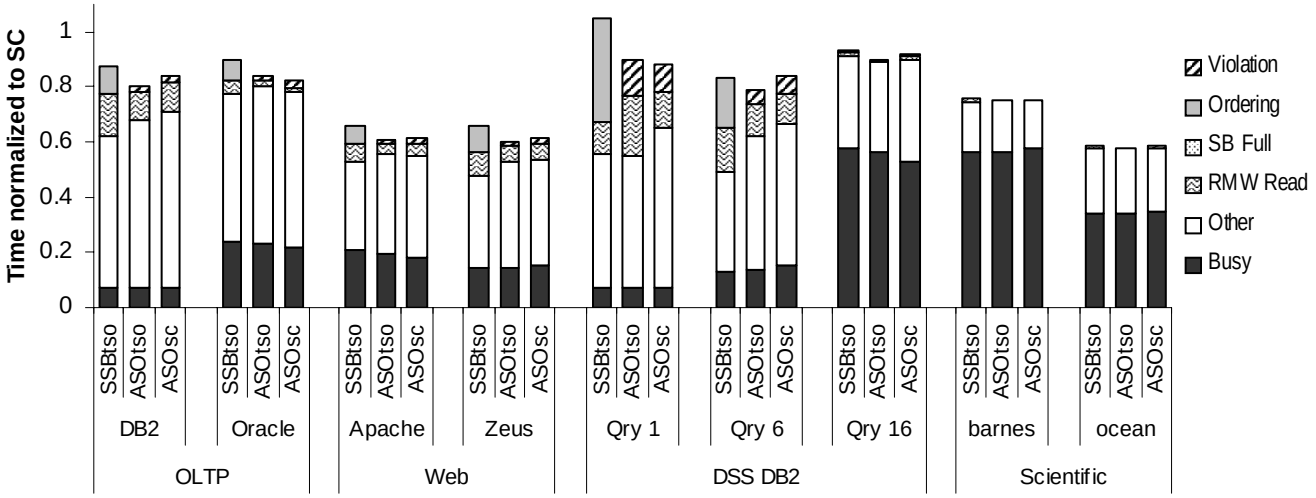

FIGURE 7. Atomic sequence ordering (ASO) performance impact.

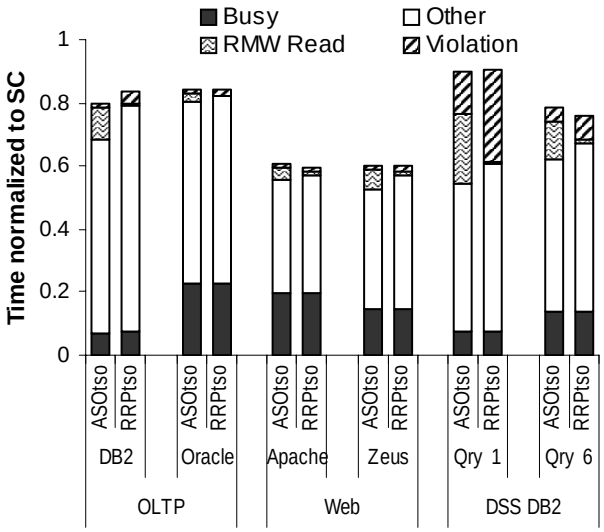

FIGURE 8. RMW Read Prediction (RRP) impact.

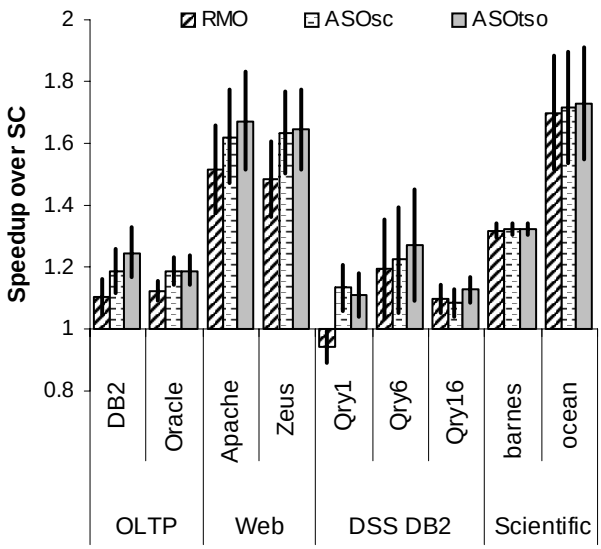

FIGURE 9. Speedup. Each bar depicts speedup relative to SC with 95% confidence intervals.

benefit from store buffering and avoid synchronization stalls. The minimal performance losses in ASO_{sc} relative to ASO_{tso} arise from increases in the average work discarded upon violations.

In general, we find that performance under ASO is insensitive to the precise heuristics used to choose sequence boundaries. These heuristics have minimal impact because violations are infrequent. We choose to allow a sequence to store at most sixteen distinct cache blocks before allocating a new sequence to enable rapid commit to L2. Four checkpoints are sufficient to achieve peak performance for this sequence length.

4.6 Impact of RMW read prediction

Even under $ASO_{tso,}$ considerable stalls on the read portion of RMWs remain. These RMW read stalls arise when a node must acquire a lock previously held by another node. In commercial applications, RMWs are typically used in simple locking protocols that use a unique value to indicate that a lock is free. Hence, we can predict this value [4] to avoid RMW read stalls when the lock is available. As ASO already requires rollback support, we can reuse this mechanism to recover from mispredictions. Unfortunately, our study shows that eliminating the lock acquisition latency has a minimal effect on performance even when the prediction is highly accurate. The prediction is ineffective because hiding lock acquisition simply exposes load stalls within the critical section.

We consider a perfect RMW read prediction mechanism that always predicts the current value of the memory location (i.e., as if the read had zero latency). Note that this algorithm can still lead to "misprediction" and rollback if the value is changed after the prediction is made (e.g., in cases of contention).

Figure 8 shows that RMW read prediction has minimal performance impact (we omit benchmarks with negligible RMW read stalls). Because we make a prediction any time an RMW would otherwise stall, all RMW read stalls are eliminated. Some of these gains are lost due to rollbacks in cases of contention. However, even in the applications with minimal contention, we observe a substantial increase in load stalls (reflected in the "Other" portion of the bars). We conclude that, when the RMW acquiring a lock misses, the loads within the critical section are also likely to miss.

4.7 Performance summary

By eliminating all ordering and store buffer capacity related stalls, ASO and the SSB provide better-than-RMO performance with the simplicity of stricter programming models. Our mechanisms can exceed the performance of relaxed programming models because they avoid conservative ordering stalls at memory barriers when precise ordering is dynamically unnecessary (i.e., when there is no race for a lock). Figure 9 summarizes the performance of the store-wait–free implementations, ASO_{sc} and ASO_{tso}, relative to conventional systems. All bars are normalized to the performance under conventional SC, and the vertical line attached to each bar indicates a 95% confidence interval on the speedup over SC given by our paired-measurement sampling methodology.

5 Related work

Adve and Gharachorloo present an overview of the differences between memory consistency models in [1]. Several prior studies investigate the performance gap between consistency models in scientific [11,13,26,27] and commercial [7,25] applications. Our results corroborate these studies; even under fully-relaxed consistency models, programs incur significant delays because of ordering constraints at synchronization operations.

Chou, Spracklen, and Abraham [7] introduce the epoch model for analyzing store misses in commercial applications. They evaluate several optimizations to reduce store stalls, including the store miss accelerator to reduce capacity misses, and hardware scouting to improve store prefetching effectiveness. These mechanisms can be combined with ASO and the SSB. Finally, they demonstrate that speculation techniques (from [23]) can bridge the performance gap among consistency models, but their analysis assumes speculation always succeeds and does not evaluate a hardware implementation.

Coherence protocol optimizations can eliminate store misses to shared data for special-case communication patterns, such as migratory sharing [33]. Our mechanisms hide the latency of all classes of store misses.

We extend prior research on speculative implementations of memory consistency [12,13,27]. These prior designs implement rollback on violation through large history buffers that undo the results of each speculatively retired instruction, and/or replace all stores with RMW operations to support rollback of memory state. In contrast, our design enforces ordering constraints at coarse granularity, enabling a checkpoint-based rollback mechanism.

Our SSB is similar to the Store Redo Log (SRL) of Gandhi et al. [9]. SRL is designed to disambiguate accesses for large-instruction-window microarchitectures (e.g., [2]). A sufficiently large instruction window hides the performance gap among consistency models [27]. However, we target conventional microarchitectures.

Like SRL, the SSB places private/speculative values into L1. However, the SSB replaces the processor's store buffer instead of its store queue. Unlike store buffers, store queues perform an age-based search to match each load to its nearest preceding store. Hence, the store queue must provide rapid access to many values for a single location. In contrast, for store buffers, only two values are of interest: the unique value currently visible to the processor, and the globally-visible (consistent) value. The SSB provides rapid access to the processor-visible value through L1, and displaces globally-visible values to L2. The SSB allows multiple overlapping writes, invokes the store replay mechanism infrequently (upon data races instead of dependence mispredictions), and does not require any of the other supporting structures of SRL. The SSB is complementary to other mechanisms that scale [9,22,28,35] or eliminate [29,34] load and store queues.

Rajwar and Goodman [23] and Martinez and Torrellas [21] propose schemes to proceed speculatively into critical sections without waiting for the lock to be acquired. We consider RMW read prediction, which is similar to these techniques. Unfortunately, we observe minimal performance gains from this approach in our applications because hiding the lock acquisition latency exposes read stalls within the critical section. Unlike their schemes, our approach still performs the lock acquisition, and does not provide the programmability benefits of allowing concurrent execution within critical sections.

A variety of architectural techniques exploit the appearance of atomicity and detection of conflicts among memory accesses to simplify programming models or improve performance. Transactional memory simplifies parallel programming by allowing programmers to specify regions of code for which the implementation guarantees the appearance of atomic execution [15,19]. Thread-level speculation enables parallel execution of sequential code provided that the speculative threads' writes appear to occur in order and reads do not conflict with out-of-order writes [14,17,31,32]. However, unlike these techniques, under ASO, hardware chooses which memory accesses to group into atomic sequences. Hence, atomic sequences can be chosen to assure that speculative data does not overflow L1, simplifying hardware requirements. Nonetheless, many of these proposals detect atomicity violations using the same underlying hardware mechanisms as ASO. Thus, by providing these fundamental mechanisms, future systems might support all these techniques for traditional parallel, transactional, and speculatively-threaded code, respectively.

Ceze et al. concurrently propose bulk enforcement of sequential consistency, which, like ASO, enforces memory ordering constraints at coarse granularity [5]. Their design builds the whole memory consistency enforcement based on coarse-grain operation with hash-based signatures. Their goal is to decouple consistency enforcement from processor design. Our study separately analyzes scalable store buffering and proposes store-wait–free implementations of relaxed consistency models in addition to SC.

6 Conclusion

In this paper, we have shown that store-related capacity and ordering stalls degrade the performance of commercial applications under all memory consistency models, including relaxed models. We propose two mechanisms to address these stalls, the scalable store buffer to eliminate capacity stalls, and atomic sequence ordering to eliminate ordering stalls. With these mechanisms, systems can provide the intuitive programming interface of strongly-ordered consistency models with performance exceeding fully-relaxed models. Furthermore, the underlying hardware mechanisms required (e.g., register checkpoints, tracking of speculative reads, speculative write data within the cache hierarchy) overlap strongly with the mechanisms required to implement emerging programming models like transactional memory. By implementing these underlying mechanisms in future designs, systems architects can provide a migration path to new programming models while enabling store-wait–free multiprocessing for legacy code.

Acknowledgements

The authors would like to thank Milo Martin, members of the Impetus research group at Carnegie Mellon University, and the anonymous reviewers for their feedback on drafts of this paper. This work was partially supported by grants and equipment from Intel, two Sloan research fellowships, an NSERC Discovery Grant, an IBM faculty partnership award, and NSF grant CCR-0509356.

References

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

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. *Proc. of the 36th Int'l Symposium on Microarchitecture*, Dec. 2003.

[3] R. Bhargava and L. K. John. Issues in the design of store buffers in dynamically scheduled processors. *Proc. of the Int'l Symposium on the Performance Analysis of Systems and Software*, Apr. 2000.

[4] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Transactions on Architecture and Code Optimization*, 3(2):182–208, 2006.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulk en-

forcement of sequential consistency. *Proc. of the 34th Int'l Symposium on Computer Architecture*, Jun. 2007.

[6] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. *Proc. of the 31st Int'l Symposium on Computer Architecture*, Jun. 2004.

[7] Y. Chou, L. Spracklen, and S. G. Abraham. Store memory-level parallelism optimizations for commercial applications. *Proc. of the 38th Int'l Symposium on Microarchitecture*, Dec. 2005.

[8] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. *Int'l Conference on Computer Design*, Oct. 2004.

[9] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai. Scalable load and store processing in latency tolerant processors. *Proc. of the 38th Int'l Symposium on Microarchitecture*, Dec. 2005.

[10] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. *Proc. of the Int'l Conference on Parallel Processing*, Aug. 1991.

[11] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. *Proc. of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[12] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. *Proc. of the 10th Int'l Conference on Parallel Architectures and Compilation Techniques*, Sep. 2002.

[13] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? *Proc. of the 26th Int'l Symposium on Computer Architecture*, May 1999.

[14] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report 92/07, Digital Equipment Corporation, Cambridge Research Laboratory, Dec. 1992.

[16] M. D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8), Aug. 1998.

[17] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

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

[19] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Publishers, 2006.

[20] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. *Proc. of the 35th Int'l Symposium on Microarchitecture*, Dec. 2002.

[21] J. F. Martinez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. *Proc. of the 10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[22] I. Park, C. Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. *Proc. of the 36th Int'l Symposium on Microarchitecture*, Dec. 2003.

[23] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. *Proc. of the 34th Int'l Symposium on Microarchitecture*, Dec. 2001.

[24] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. *Proc. of the 10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[25] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[26] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve. The interaction of software prefetching with ilp processors in shared-memory systems. *Proc. of the 24th Int'l Symposium on Computer Architecture*, Jun. 1997.

[27] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. *Proc. of the 9th Symposium on Parallel Algorithms and Architectures*, Jun. 1997.

[28] A. Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. *Proc. of the 32nd Int'l Symposium on Computer Architecture*, Jun. 2005.

[29] T. Sha, M. M. K. Martin, and A. Roth. NoSQ: Store-load communications without a store queue. *Proc. of the 39th Int'l Symposium on Microarchitecture*, Dec. 2006.

[30] M. Shao, A. Ailamaki, and B. Falsafi. DBmbench: Fast and accurate database workload representation on modern microarchitecture. *Proc. of the 15th IBM Center for Advanced Studies Conference*, Oct. 2005.

[31] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *Proc. of the 22nd Int'l Symposium on Computer Architecture*, Jun. 1995.

[32] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *Proc. of the 27th Int'l Symposium on Computer Architecture*, Jul. 2000.

[33] P. Stenstrom, M. Brorsson, and L. Sandberg. Adaptive cache coherence protocol optimized for migratory sharing. *Proc. of the 20th Int'l Symposium on Computer Architecture*, May 1993.

[34] S. Subramaniam and G. H. Loh. Fire-and-Forget: Load/store scheduling with no store queue at all. *Proc. of the 39th Int'l Symposium on Microarchitecture*, Dec. 2006.

[35] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia. Store buffer design in first-level multibanked data caches. *Proc. of the 32nd Int'l Symposium on Computer Architecture*, Jun. 2005.

[36] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu. Conditional memory ordering. *Proc. of the 33rd Int'l Symposium on Computer Architecture*, Jun. 2006.

[37] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, Jul-Aug 2006.

[38] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation through rigorous statistical sampling. *Proc. of the 30th Int'l Symposium on Computer Architecture*, Jun. 2003.