Atomic SC for Simple In-order Processors

Dibakar Gope and Mikko H. Lipasti
University of Wisconsin - Madison
gope@wisc.edu, mikko@engr.wisc.edu

Abstract

Sequential consistency is arguably the most intuitive memory consistency model for shared-memory multi-threaded programming, yet it appears to be a poor fit for simple, in-order processors that are most attractive in the power-constrained many-core era. This paper proposes an intuitively appealing and straightforward framework for ensuring sequentially consistent execution. Prior schemes have enabled similar reordering, but in ways that are most naturally implemented in aggressive out-of-order processors that support speculative execution or that require pervasive and error-prone revisions to the already-complex coherence protocols. The proposed Atomic SC approach adds a light-weight scheme for enforcing mutual exclusion to maintain proper SC order for reordered references, works without any alteration to the underlying coherence protocol and consumes minimal silicon area and energy. On an in-order processor running multithreaded PARSEC workloads, Atomic SC delivers performance that is equal to or better than prior SC-compatible schemes, which require much greater energy and design complexity.

1. Introduction

Sequential consistency (SC) [1, 2] is considered the most intuitive memory model to programmers. A straightforward naïve implementation of SC disallows all possible reordering of shared memory operations in program order. Consequently, many of the architecture and compiler optimizations used in uniprocessors are not safely applicable to sequentially consistent multiprocessors.

To achieve better performance, alternative memory models have been proposed that relax some of the memory ordering constraints [1] imposed by SC and enable increased concurrency (i.e. *memory-level parallelism, MLP)*. The complexities of specifying relaxed memory models, however, combined with the additional burden it places on the programmers make relaxed memory models less appealing. While many widely-used instruction sets have adopted relaxed models, the MIPS ISA continues to require SC, is widely deployed in the embedded space and has seen renewed development in the Godson and Loongson projects [21]. Furthermore, many proposals on deterministic replay and debug work best with SC. As a result, a large body of prior research strives to retain the programmer's productivity by designing SC hardware either by using in-window speculation [3, 15] or post-retirement speculation [5, 6, 7, 9, 10, 11, 12, 19].

While the above approaches are promising, the hardware complexity associated with them is inconsistent with the prevailing trend towards many-core chips consisting of simple energy-efficient cores and is likely to hinder their continued adoption in commercial processors. Instead, a lightweight hardware solution to enforce SC is required for in-order low-power many-core processors, as they cannot afford energy-hungry speculation support.

On a conventional in-order processor, cache misses stall the execution pipeline. Enforcing SC necessitates each pending miss to complete before servicing any subsequent memory requests. In our Atomic SC framework, we allow younger memory operations to non-speculatively complete past pending writes and reads (pending due to a write permission upgrade or cache miss, hereafter referred to as a *miss*). Atomic SC enables this by forcing the miss to obtain a fine-grained mutex to the memory block it is requesting before initiating any coherence transaction on that block. As soon as the miss acquires a mutex, the pipeline resumes and the memory system can initiate the next request from the same core. Furthermore, before the memory reference of a younger operation is committed in shadow of a pending miss, the younger request must also obtain a mutex to its memory block. Mutex acquisition for the miss and the following memory requests effectively forces conflicting memory operations from other cores to the same addresses to wait until the pending miss completes. This ensures SC since all locally reordered requests complete *atomically* (hence the name *Atomic SC*). When the pending miss eventually completes, it releases all the mutexes acquired under its *miss shadow* and allows conflicting accesses in other cores to wake up and begin their coherence transactions. The mutexes we describe are not programmer-visible; programs use load-linked/store-conditional (LL/SC) to synchronize.

The key insight here is that younger operations in the shadow of a miss are not allowed to issue until the miss obtains a mutex for the requesting block, as opposed to stalling on the miss and waiting for it to finish. Whereas a mutex acquisition incurs some latency (as much as a round trip to the "home node" of the mutex pool), a miss's service latency can be much longer than this, if the request cannot be satisfied from the L2 cache or the request requires communicating with remote sharers. Hence, subsequent requests can issue earlier, leading to reduction in pipeline stalls, additional concurrency and MLP.

Figure 1(a) shows a code segment with two pairs of conflicting memory operations in two processor cores c0 and c1 in program order and Figure 1(b) describes our approach of ensuring SC. Initially, when store A incurs a cache miss at time t1, it grabs a mutex to A at t2 and allows load B to continue past the pending store A. Load B manages to acquire a mutex to B at t3 and safely completes at t4. On the other hand, when store B from c1 is issued at t5, it is forced to stall, since the store-miss B fails to obtain a mutex. Later at time t_6, when the pending store A from c0 completes, it releases mutexes to A and B, allowing store B and load A from c1 to issue their request. However, if load B from c0 is allowed to complete past the pending store A without acquiring mutexes, then the load A from c1 might have already been completed before the invalidation to address A hits c1's private cache, potentially violating SC. The act of acquiring mutex to A and B essentially forces store B and load A from c1 to wait until store A releases the mutexes, resulting in a execution order: load B (c0) → store A (c0) → store B (c1) → load A (c1) that honors SC. Moreover, if the conflicting pairs of accesses from c0 and c1 are staggered in time as is the common case in most parallel applications, then a store-miss B from c1 would not stall c1 as long as the prior holder (i.e. c0) releases the mutex to addresses A and B. In this scenario, the store-miss B will only need to wait for acquiring a mutex, before allowing load A to complete. Since the act of acquiring mutex to B from c1 ensures the possible completion of store A from c0, allowing load A to complete in shadow of the pending store B does not violate SC.

To summarize, our work makes the following contributions:

- We describe Atomic SC that mitigates most of the performance loss caused by cache misses in in-order multiprocessors and closes the performance gap with a weakly-ordered design (RMO). Atomic SC does not rely on speculation and its accompanying complexity.
- Experimental study on PARSEC shows that Atomic SC improves performance by 9.38% on average, ranging from 3.61% to 16.33% over the baseline SC, delivers slightly better performance than TSO (1.24% on average) and incurs a slowdown of 4.2% on average in comparison to RMO.

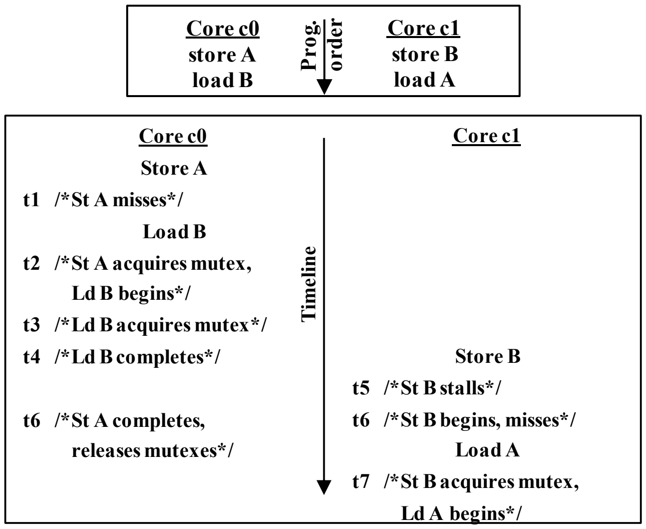
Core c0
store A
load B
Prog. order
Core c1
store B
load A
Core c0
Store A
t1 /*St A misses*/
Load B
t2 /*St A acquires mutex, Ld B begins*/
t3 /*Ld B acquires mutex*/
t4 /*Ld B completes*/
t6 /*St A completes, releases mutexes*/
Timeline
Core c1
Store B
t5 /*St B stalls*/
t6 /*St B begins, misses*/
Load A
t7 /*St B acquires mutex, Ld A begins*/

Figure 1. (a) Code segment in program order (top), (b) Execution order using Atomic SC: load B (c0) → store A (c0) → store B (c1) → load A (c1), ensuring SC (bottom).

- Quantitative comparison to two prior non-speculative approaches, Conflict Order [15] and Ack-Delay [20], Atomic SC delivers equivalent or better performance, while resulting in about 3x and 2x reduction in area overhead, 4.65x and 3.65x saving in power requirement and does not require any modifications to the coherence protocol.

2. Atomic SC enforces SC order

In this work, we use constraint graph representation [13, 14] to prove that Atomic SC ensures SC. A constraint graph consists of a set of vertices representing memory operations in a multithreaded execution, connected by edges that dictate their correct ordering. For a sequentially consistent system, there are four edge types: program order (*po*) edges that place an order on all memory operations executed by a single processor and the dynamic ordering relations (*dyn-raw*, *dyn-war* and *dyn-waw*) that reflects the order among all memory operations that read or write the same location. The constraint graph can be used to test the correctness of an execution by checking the graph for a cycle. The presence of a cycle indicates that the execution violates SC, because the observed order of operations cannot be placed in an order dictated by SC. Figure 2 illustrates how the constraint graph is used to reason out the correctness of Atomic SC. Figure 2(a) shows an example constraint graph for an execution that violates SC, because core c1's second load should return the value of memory location A written by c0, instead of the original value of A. The constraint graph corresponding to this execution contains a cycle, thereby violating SC.

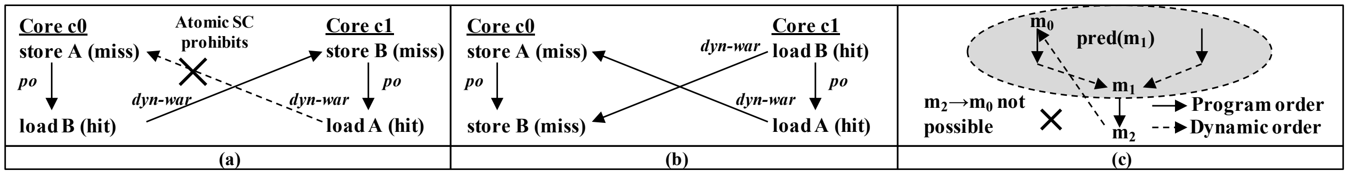

Figure 2. Constraint graph examples. (a) Sequentially inconsistent, (b) Sequentially consistent, (c) Atomic SC ensures SC.

We first informally describe the way Atomic SC enforces SC with two examples in Figure 2(a) and Figure 2(b). Then we prove the correctness of Atomic SC by building on the formalism detailed by Lin *et. al* in Conflict Order[15]. In Figure 2(a), Atomic SC requires the store-miss A and the following load-hit B in shadow of that to obtain mutexes before executing their request. Mutex acquisitions of the two requests and the load-hit B in c0 effectively forces conflicting request store B in c1 to find a miss, be issued to the directory and get served in completion of the pending store A and their subsequent release of the held mutexes. This forms a *dyn-war* edge from c0 to c1. Furthermore, until the store-miss B in c1 is able to grab a mutex, any younger request (load A) in shadow of that is not allowed to be served in Atomic SC. As a result, the completion of the pending store A in c0 before initiating to serve the store-miss B in c1 effectively eliminates the possibility of the load A in c1 to read an old value and form an *dyn-war* edge from c1 to c0, resulting in no cycle in the constraint graph and ensuring SC.

In Figure 2(b), Atomic SC requires the store-miss A and the following store-miss B in shadow of that to acquire mutexes before initiating their coherence action. However a cache hit (load-hit B in c1) is not required to obtain a mutex in absence of an outstanding cache miss in Atomic SC, since c1 with load B reading an old value has not communicated with c0 via a cache miss since the invalidation to shared copies of B was posted. Intuitively, therefore, SC is preserved, because these loads could have been done before the invalidation of store-miss B from c0 hit c1's private cache. As a result, load B (c1) precedes store B (c0) in the global memory order, therefore serving the cache-hits in c1 without acquiring mutexes in absence of a pending cache miss does not lead to a cycle in the constraint graph of Figure 2(b).

As described by Lin *et al.* [15], in order for a multiprocessor execution to be sequentially consistent, a memory operation m_2 (whose immediate predecessor in program order is m_1) is allowed to complete iff the memory operations from *pred*(m_1) which conflict with m_2 have completed, thus forbidding the dynamic ordering edge $m_2 \rightarrow m_0$ and subsequently a cycle to form in the constraint graph, as shown in Figure 2(c). In an Atomic SC system, m_1 is a cache miss and *pred*(m_1) refer to memory operations that include m_1 and those before m_1 in the global memory order. We argue that the definitions detailed in [15] to prove the correctness of Conflict Ordering holds for Atomic SC as well, since Atomic SC prevents a cycle in the constraint graph by forcing the cache miss m_1 and the following memory operation m_2 in shadow of that to acquire global mutexes before they are allowed to complete. The availability of the mutexes to m_1 and m_2 ensures that the conflicting memory requests from *pred*(m_1) in the global memory order must have completed. This in turn allows m_2 to safely commit past the pending m_1 without running into a risk of creating a dynamic ordering edge $m_2 \rightarrow m_0$ and closing the cycle in the constraint graph, violating SC. Conversely, the non-availability of the mutex to m_1 indicates that the coherence activity of the conflicting memory operations from *pred*(m_1) may have not yet completed. This in turn forces m_1 and subsequently m_2 to wait until the conflicting operations in *pred*(m_1) complete.

3. Atomic SC

In order for a processor to proceed in presence of a pending miss while still enforcing SC, the effect of the subsequent accesses of the processor should not be made visible to any other processor in the system, creating an illusion to other processors as if these accesses are actually performed after the pending miss completes. In this section, we describe at a high level the way Atomic SC ensures that by completing all locally reordered requests *atomically*. We assume a system consisting of in-order processors, augmented with store buffers. The shared L2 cache (LLC) and coherence directories are interleaved across the nodes. The shared mutex pool is equally partitioned into sets among the directories. Thus, a memory request maps its addresses to a directory and then hashes its address (excluding the bits for the offset and directory) into a mutex mapping. Furthermore, all processors observe writes to a given location in the same order.

3.1. Overview

In Atomic SC, before a write-miss allows a younger request to proceed, it requests a mutex to the home node. If the mutex is available, the home node sends it to the requestor and continues on to serve the write-miss by invalidating all shared copies and also accessing memory in case of a miss in L2 cache. Once a positive acknowledgement to the mutex acquisition request reaches the requestor, the store is moved to the store buffer and the processor can initiate processing the following request in program order in shadow of the pending write. The availability of the mutex to the write address ensures that all pending memory operations in the global memory order that can potentially conflict with younger memory operations in shadow of that and violate SC have completed. However if the mutex is not available, the home node buffers the request and responds to that later when the current holder releases the mutex. Furthermore, before any younger request accesses the cache in presence of the pending write, the request must also obtain mutex to the memory block in order to prevent any conflicting memory operation from occurring in other cores. Subsequently, when the pending write completes, a multicast message is sent to the corresponding home nodes to release mutexes that were acquired in shadow of the pending write. Releasing mutexes also indicates that any coherence activity in regard to the write-miss has finished, ensuring no coherence interference with subsequent conflicting requests from other cores. Consequently any requests from other cores that are queued up on the responsible nodes to acquire those unavailable mutexes are also woken up and serviced. Thus it appears to all processors as if the younger request actually executes after the pending write completes. Figure 3 illustrates the performance advantage of Atomic SC. We assume that St X incurs a cache miss and Ld Y is the following request in program order.

In shadow of a pending write, if the mutex acquisition of a younger request is interrupted by a conflicting request, it means that the mutex must have already been claimed by another core. In that case, the memory system will not continue to process any following requests from that core until the mutex becomes available or the pending write completes. Furthermore, if a younger write request in shadow of a pending write also experiences a cache miss, then the in-order core would stall again. In order to avoid this stall, Atomic SC design facilitates non-speculative completion of memory operations past multiple pending writes, thus reducing memory ordering stalls even further without violating SC. In a similar way, Atomic SC allows younger requests to complete past pending reads once the read-miss and younger requests grab mutexes.

3.2. Mutex release

Atomic SC allows memory operations to complete past multiple pending writes. Depending on when Atomic SC releases held mutexes leads to two mutex release policies.

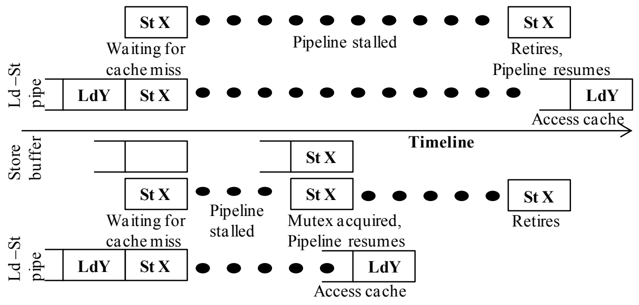

Figure 3. Execution timeline: baseline SC (top) and Atomic SC (bottom).

One policy is to delay releasing of held mutexes until all the pending writes complete, which we term as *lumped-release* policy. This has the advantage of reducing hardware overhead, as detailed in section 4, but at the cost of increasing mutex hold time and pressure on finite mutex resources. Considering that conflicting requests are staggered in time in most parallel applications, this lazy release of mutexes should not impair the performance to a great extent unless an application has very frequent write-misses. If an application keeps incurring write-misses before the preceding write-misses complete, then this policy can end up holding mutexes indefinitely, leading to many conflicts and unnecessary processor stalls. In that case a timeout[1] mechanism in our design can release mutexes and ensure prompt forward progress. Note that deadlock cannot occur, since the processor will eventually encounter a write-miss for which it cannot acquire a mutex (since some other processor holds it) and as a result the write-miss will not be moved into the store buffer until it acquires a mutex. While waiting for that mutex, it's older pending writes will finish, bring down the pending write count to zero and release all of its held mutexes.

The detrimental impact of longer mutex hold time of the *lumped-release* policy on performance can potentially be recovered by releasing mutexes of a write-miss and any following requests that complete under its *miss shadow* as soon as the pending write completes. We name this as *precise-release* policy. However, in presence of multiple pending writes, if a younger write-miss completes before the oldest write-miss, then in order to preserve SC, *precise-release* does not allow releasing mutexes that the younger write-miss prompted acquisitions previously until all the older pending writes complete. This policy has the potential to maximize performance, but at the cost of considerable increase in hardware to track acquired mutexes, as detailed in section 4.

[1]Timeout: release mutexes when a timer expires with timeout value set to 600 cycles. Timer is set if a write-miss occurs in absence of an older pending write and reset at mutex release.

3.3. Criticality to mutex acquisition latency

Atomic SC allows requests to complete past pending misses, once they acquire mutexes. However, the act of acquiring a mutex is latency critical as it involves a round trip to the directory. Since this adds to the request's critical path, it can outweigh the performance benefit achieved by avoiding memory ordering stalls. Spatial locality in the reference stream mitigates this problem, since there are frequently multiple requests to the same block under the shadow of the same miss (mutexes are assigned per block in our design). In order to observe the impact of this mutex acquisition latency on performance, we consider a low latency mutex network [22] as one of the variants in Atomic SC's design. This network supports multiple nodes participate in administration of the system's same set of mutexes, thus reduces the average distance a mutex request travels to get to a node with the authority to respond.

4. Implementation and operation

4.1. Hardware structures

Figure 4 shows a processing node. First, each core is augmented with an up-down counter, *inflight-write-miss-count*, that is incremented on a write-miss and is decremented when the write-miss completes. The positive value of this indicates the presence of pending writes and requires younger requests to obtain mutexes before they complete.

The *lumped-release* policy uses a bit-vector array to track the status of all acquired mutexes so far under all pending writes. If the completion of a write-miss brings down the *inflight-write-miss-count* to zero, then the bit-vector is consulted and all mutexes are released, resetting the entire bit-vector.

On the other hand, the *precise-release* policy adds a *mutex-lookup* table in each core. It is a FIFO queue; each entry tracks the mutex-mapping of an acquired mutex. Each entry is further augmented with a store-color[2] field and a *mutex-release* bit. Whenever a request acquires a mutex in presence of a pending write, it inserts the mutex-mapping of the request into the *mutex-lookup* table, saves the store-color of the pending write that prompted its mutex acquisition and set the *mutex-release* bit to false. Each mutex is acquired only once, so before sending a mutex request, the *mutex-lookup* table must be searched associatively to see if the mutex has already been acquired by an earlier reference. When a pending write completes, it sets the *mutex-release* bit of the set of entries that have the same store-color as the completed write to true.

Furthermore, if it was the oldest pending write that just completed, then the *mutex-lookup* table is scanned

[2]Store-color is an up-counter, incremented in a write-miss

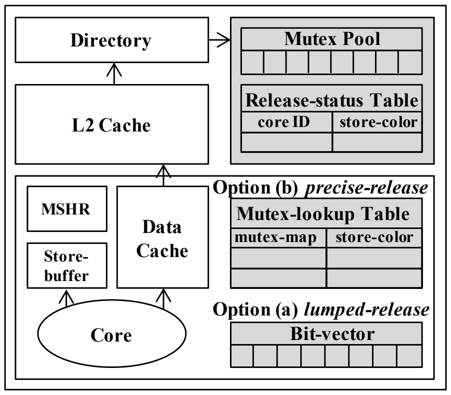

Figure 4. Processing node in Atomic SC.

Core c0	Core c1
1. (3) store A (miss)	1. (4) store B (miss)
2. (1) load B (hit)	2. (5) load C (hit)
3. (6) store C (miss)	
4. (2) load B (hit)	

Figure 5. SC Violation; operation is labeled by program order, issue order (in parentheses).

sequentially, releasing mutexes (by communicating to the directory through messages, as described later) for all the entries whose *mutex-release* bit is true until the scanning hits an entry with *mutex-release* bit still set to false. *Mutex-release* bit in the *mutex-lookup* table ensures in-order releases of held mutexes.

Mutex release in *precise-release* policy requires some additional precautions to be taken to avoid a potential SC violation. Let us consider the scenario shown in Figure 5. The issue order (shown in parentheses) might occur and violate SC due to inappropriate release of held mutexes. The first load B in c0 is allowed to complete past pending store A, once it acquires the mutex, updates the *mutex-lookup* table entry and the corresponding store-color field with the store-color of the pending store A. Later, store C incurs a cache miss before pending store A completes. Now before a mutex acquisition request to second load B is issued under pending store C, the *mutex-lookup* table is searched associatively for a mutex-mapping that matches with address B. Upon finding a matching entry, the core services the cache hit to second load B immediately. Nevertheless, the mutex-mapping of this second load B is inserted into the *mutex-lookup* table and the corresponding store-color field is updated with the store-color of the pending store C. The act of mutex acquisition to B blocks conflicting request store B in c1 to proceed. Now if pending store A completes before pending store C, it will attempt to release mutexes for A and B, pass on the mutex to store B in c1, thus potentially allowing load C in c1 to complete, leading to an execution order (second load B (c0) → store

B (c1) → load C (c1) → store C (c0)) that violates SC. So in completion of an oldest pending write, before a mutex is released, the *mutex-lookup* table is searched associatively for a matching mutex-mapping with a different store-color whose mutex-release is still set to false. If none is found, the mutex can be released immediately, otherwise it is left to the younger pending write to release later and so on.

It is worth noting that the bit-vector array in *lumped-release* does not require any associative search during mutex acquisition or release, as required in *precise-release*. Furthermore, since the per-core bit-vector or the shared mutex pool is decoupled from the L1 or L2 cache, evictions or replacements of lines with mutex held is allowed and correct in our design.

In addition to a per-core bit-vector (*lumped-release*) or *mutex-lookup* table (*precise-release*) and a shared mutex pool, Atomic SC requires the addition of three new messages to the memory system. First, the mutex request message carries the store-color of a write-miss that prompts the acquisition of mutexes. Second, the mutex response message conveys the status of a mutex to a requestor node. Third, the mutex release message carries the mutex-mappings to release held mutexes in the shared mutex pool. We describe the usage of the *release-status* table as shown in Figure 4 towards the end of this section.

4.2. Complexity of interaction with coherence protocol

Atomic SC requires only three additional messages for managing mutexes, but these do not add any extra stable or transient coherence states and do not interact with the coherence protocol in any way. Checking the mutex status either in the per-core mutex bit-vector or in the shared mutex pool and initiation of mutex acquisition or release request only add some trivial encoding and decoding logic to the L1 and L2 controller. Shaded components in Figure 6 highlight the logic added into the L1 controller. Mutex requests and memory requests from different cores cannot race against each other, as the L2 controller serializes the requests. Waiting for mutexes in the L2 controller eliminates the possibility of a resource cycle and hence deadlocks because mutexes are acquired in program order.

Now we describe how Atomic SC interacts with the added structures for typical cache events.

4.3. Write-miss actions

The write-miss attempts to acquire a mutex first. The write request is tagged with the store-color of the write-miss and is issued to the home node. When the L2 node receives the request, it checks the status of the mutex in the mutex pool and accesses the directory in parallel with retrieving the block's coherence states. If the mutex is available, it sends a positive acknowledgement and

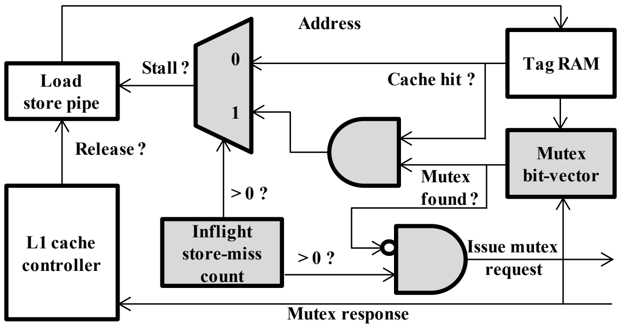

Figure 6. Atomic SC L1 cache controller.

continues on to satisfy the write request. If the mutex is unavailable, the responder buffers the write request and responds to that later when the current holder releases the mutex. Upon receiving a positive response, the requestor updates the bit-vector (*lumped-release*) or inserts the mutex-mapping of the write-miss into the *mutex-lookup* table with appropriate store-color number (*precise-release*) and begins the next request in the program order. If the write request requires only upgrading the block's cache permission and the mutex to that block is already acquired by an earlier reference of the core, then the core issues an upgrade request to the L2 node and begin processing the next request in the program order immediately. Nevertheless, the *precise-release* inserts the mutex-mapping of the write request into the next FIFO entry of the *mutex-lookup* table with the store-color of the current write-miss. Once the pending write completes, the requestor releases the mutexes according to the *precise or lumped-release* policy as described earlier.

In a similar way, Atomic SC requires a read-miss to obtain a mutex and update the added structures before allowing any younger request in the program order to proceed.

4.4. Cache-hit actions

If the value of *inflight-write-miss-count* is zero, then the cache-hit completes immediately. In absence of an outstanding cache-miss, a cache-hit is not required to obtain a mutex, as discussed in section 2. A positive value of *inflight-write-miss-count* necessitates the request to check the direct-mapped mutex bit-vector in *lumped-release* design. If the mutex is already acquired by an earlier reference, then it satisfies the cache-hit immediately. Otherwise, the cache-hit issues a mutex request to the home node. Upon receiving a positive mutex response, the cache-hit completes, the local bit-vector is updated and the requestor initiates the next request in the program order. In *precise-release* design, if the current request does not find a matching entry in the *mutex-lookup* table, it issues a mutex request. Upon receiving a positive response, the requestor updates the *mutex-lookup* table as described before.

Now we describe a race condition that occurs due to the lack of point-to-point ordering between two message classes in the interconnection network of our simulation framework. If the underlying network can guarantee that, we do not need to have this overprovision in our design.

If the mutex request of a cache hit races with the mutex release of a completed write that prompted the mutex acquisition, then a potential exists that the mutex acquired for the cache-hit is never released by the requestor node. This results in the corresponding mutex being never given to another requestor in future, leading to a deadlock. Since mutex request and mutex release messages travel through different virtual networks to a responder, this can happen if a completed write can reach a responder and release mutexes before the mutex request seizes a mutex at the responder. In order to prevent these races to the mutex pool, Atomic SC implementation necessitates the store-color of few recent completed writes that prompted mutex acquisitions previously and subsequently released mutexes - to track alongside a responder's mutex pool. We name this as *release-status* table (Figure 4). The *release-status* table is a direct-mapped table having one entry for each core. With this support, when a responder node receives a mutex request, it checks the local *release-status* table to verify if the particular requestor already released mutexes for the write-miss that prompted this mutex acquisition. If the store-color of the write-miss tagged with the mutex request matches with the store-color of the last completed write from that requestor, then the responder does not perform mutex acquisition to that request. This handles the possibility of a pending write being completed in the meantime that eliminates the need to acquire a mutex for the younger request.

5. Discussion

5.1. Interaction with explicit synchronization

Atomic SC treats the explicit synchronization operations (LL/SC) as a regular memory request. This may potentailly hurt the performance, although correctness is not affected. Typically, at the beginning of a critical region, a successful atomic read-modify-write operation writes a lock variable. Before the end of the critical region, if another core attempts to acquire the lock, it will force the lock holder to relinquish exclusive ownership of the cache line containing the lock. Furthermore, if the read request to that cache line from another core attempts to complete in shadow of a pending write of that core, then Atomic SC will force the read to obtain a mutex for the cache line. This mutex acquisition will forbid the current holder to re-acquire ownership of the cache line and release the lock until the pending write of that core containing the mutex completes. However in our simulation, we found this subtle scenario to be rare, as a result considering the synchronization operations as a regular memory request does not hurt performance in Atomic SC.

5.2. No deadlock is possible

In Atomic SC, until a memory operation obtains a mutex, any following memory operation in the program order is not allowed to request for a mutex. Furthermore, a pending store is not moved to the store buffer until it acquires a mutex. As long as the underlying coherence protocol is deadlock-free, while waiting for that mutex, it's other older pending stores will complete and all mutexes acquired in their shadow will be released. Consequently, a core cannot stall the release of all of its held mutexes, while waiting for a mutex response to a younger pending store to arrive. This invariant effectively eliminates the possibility of a resource cycle, hence deadlocks.

5.3. Suitability to out-of-order (OoO) core

Atomic SC is relevant, but relatively less valuable, since the hardware savings and avoidance of associative structures that we provide using *lumped-release* policy over prior proposals are overshadowed by expensive OoO core. However, mutexes can be used to avoid associative queue searches and reduce or eliminate pressure on the load queue or out-of-window buffering structures. The benefit is less significant, since OoO cores require large associative buffers for other reasons.

5.4. Applicability to relaxed model (RMO)

We can extend our Atomic SC's idea to enable post-fence accesses in RMO programs and thus reduce fence overhead dynamically, as explored in [16] and [17]. In a RMO design, instead of stalling on a fence and waiting for the store buffer to drain, the post-fence references can be safely executed, as soon as the pending writes in the store buffer acquire mutexes. Mutex acquisitions of the pending writes effectively force conflicting post-fence references from other cores to wait until the pending writes complete.

6. Experimental results and analysis

6.1. Evaluation methodology

We evaluated our Atomic SC design using GEM5 [26] full-system simulator. We used GEM5's Ruby memory model. We modeled our processor in accordance to the ARM11 MP inorder core [25] that supports stall-on-use (hit under miss) policy for load misses *(*implemented in Alpha 21164 as well). When the MSHR fills, the pipeline stalls. Table 1 details the processor configuration. We utilized the existing request/reply virtual networks used for coherence messages to communicate mutex request, response and release messages. We used the PARSEC benchmarks [27] with their "sim-medium" input set. For vips, x264,

Table 1. Processor configuration.

Cores	16-core CMP, single-threaded, 2-wide In-order, 2GHz, 4-entry Store buffer for TSO, RMO, Atomic SC
L1 Cache	Private 64KB, 2 way, latency 3 cycles, 4 read MSHRs, 4 write MSHRs
L2 Cache, prefetcher	Inclusive shared 8MB, 16 way, 64B blocks, 12 cycle latency, Stream-based prefetcher (16 streams)
Coherence, Interconnect Memory	MESI, 4x4 2D Mesh, 2GB total memory, 150 cycles of DRAM lookup latency
Additional Hardware	Per-core bit-vector 128B (*lumped-release*), Per-core *mutex-lookup* table 16 2B entries (*precise-release*), Shared mutex pool 128B (Global)

freqmine we used smaller input set, as their simulation time is very long. We considered execution time to compare the performance of the benchmarks.

6.2. Results and analysis

The experimental evaluation of our design is divided into three sections. First, we compare the performance of our Atomic SC design to the baseline SC, TSO and RMO implementations. In our baseline SC design, processor stalls on every cache miss. Next, we show the breakdown of the execution time of Atomic SC to reflect the benefit obtained when we relax the store-to-load ordering, store-to-store ordering first (referred to as Atomic SC (store-load, store-store)) and then load-to-load ordering (referred to as Atomic SC (store-load, store-store, load-load)) in our Atomic SC design. We also evaluate another variant of our Atomic SC design, which we refer to as Atomic SC-low latency mutex (Atomic SC-llm), described earlier in section 3.3. Furthermore, in order to observe the impact of the mutex latency, we evaluate a hypothetical interconnect that facilitates access to mutexes with no latency and thus sets an upper bound on the performance. We name it as Atomic SC-ideal. Next, we perform sensitivity study to the *mutex-lookup* table, required in our *precise-release* design. The third subsection investigates the effect of *precise-release and lumped-release* on the overall performance, in conjunction with the sensitivity study to the shared mutex pool.

6.2.1. Performance summary. Figure 7 compares the normalized execution times (normalized to the baseline SC) of Atomic SC using *lumped-release* policy of releasing mutexes to the baseline SC, TSO and RMO designs. In TSO, writes are serialized. In contrast, Atomic SC eases pressure on the store buffer by relaxing store-store ordering. Atomic SC improves performance by 9.38% on average, ranging from 3.61% to 16.33% over the baseline SC, delivers slightly better performance than TSO (1.24% on average). Relative to the RMO design, Atomic SC incurs a slowdown of 4.2% on average. The benefit using Atomic SC is primarily attributed to the following key factors:

(a) Reduction in pipeline stalls. In our experiments, mutex acquisition incurs about 10-15 cycles (2-hop) of latency, whereas a write or read-miss incurs about 25-30 cycles (3-hop) of latency if the miss is required to consult remote sharers to either invalidate or get data on chip. So intuitively applications with either high off-chip misses or high on-chip misses that require 3 or more hops to service extract more benefit using Atomic SC. The 3rd column in Table 2 shows the average number of requests serviced in shadow of pending writes in Atomic SC.

(b) Spatial locality. Spatial locality in PARSEC workloads reduces the fraction of requests that requires fetching a mutex from the directory under pending writes substantially. The 2nd column in Table 2 shows that on average about 20% of the requests under pending writes incur mutex acquisition latency to the directory, while the remaining fraction finds a mutex that has already been acquired by an earlier reference in the local bit-vector. As a result they service their cache hits immediately.

(c) MLP extraction. If a younger memory request in shadow of a pending miss also finds a miss, it can effectively hide its miss latency in shadow of the first miss. Atomic SC extracts MLP in those cases.

Figure 8 shows the breakdown of the execution time of Atomic SC. The first bar shows the performance gain with Atomic SC (store-load, store-store) and the next bar shows the additional benefit with Atomic SC (store-load, store-store, load-load). The Atomic SC-llm design improves the performance by about 1.5% on average over the Atomic SC (store-load, store-store, load-load) design. The performance loss of Atomic SC over RMO can be further reduced by avoiding mutexes for private and shared read-only accesses, as explored in SC-preserving compiler [23] and End-to-end SC [24]. We leave this exploration for future work.

Among the benchmarks shown in Figure 7 and in Figure 8, blackscholes has low cache miss rates, resulting in little performance gain with either of the memory models. Canneal has low write-miss rate and so the baseline TSO does not gain that much benefit. However, high L1 and L2 read miss-rates (12.2% and 22.7% respectively) in canneal lead to significant benefit from Atomic SC, as demonstrated in Figure 8, since the off-chip read misses now only require a round trip to the directory before the pipeline resumes. Streamcluster, ferret and freqmine benefit substantially from high write misses (4.54%, 4.02% and 3.1% respectively), major fraction of those require minimum 3 hops to get service. Furthermore, freqmine has high off-chip write misses (35.1%). The reasonable benefit with relaxing load-to-load ordering in streamcluster in Figure 8 is attributed to its high read misses that require 3 hops to get data from remote sharers. Although swaptions and dedup have moderate write misses (1.67% and 1.51% respectively), a noticeable fraction

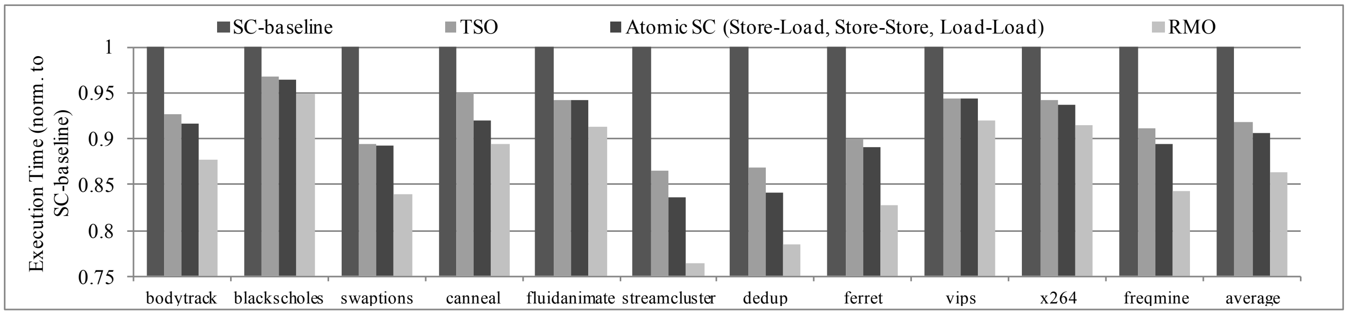

Figure 7. Performance comparison of Atomic SC to baseline SC, TSO and RMO.

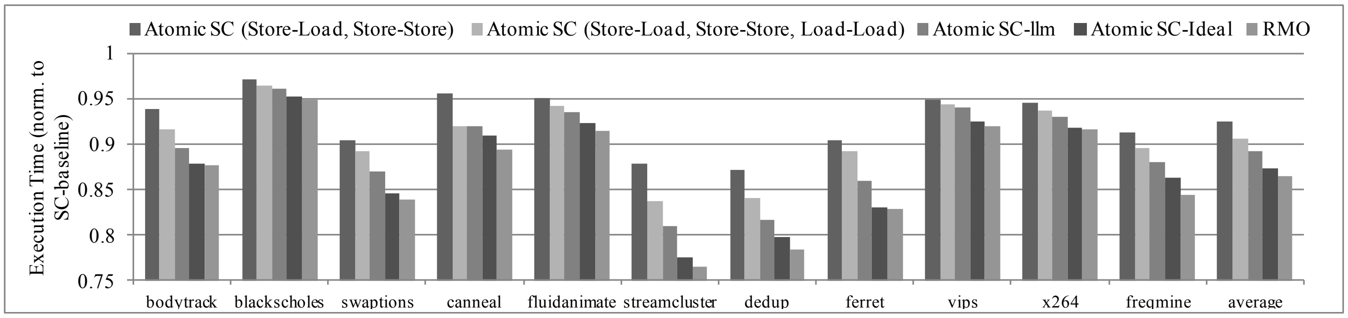

Figure 8. Time breakdown of Atomic SC and comparison to low-latency mutex, ideal version and RMO.

Table 2. Benchmark characterization.

Benchmark	% memory reqs. under write-misses issuing mutex reqs.	Avg. reqs. serviced under write-misses	Address conflicts
bodytrack	14.59	1.78	8218
blackscholes	16.22	2.47	6093
swaptions	15.26	1.68	7751
canneal	28.70	1.87	4148
fluidanimate	23.37	1.60	11759
streamcluster	57.53	1.38	508593
dedup	12.50	1.45	137147
ferret	10.90	1.95	25349
vips	17.68	2.01	9474
x264	12.96	3.33	4327
freqmine	31.77	3.16	987543

of those write misses entail 3 hops to get service. Furthermore, the off-chip write misses in dedup are reasonably high (12.9%). Bodytrack has lower write miss rate than swaptions, however the off-chip write and read misses in bodytrack is substantially higher than in swaptions; Atomic SC benefits reasonably from those misses. The remaining workloads have low cache miss rates, resulting in little performance gain.

The performance gap between the Atomic SC and the RMO design in Figure 7 is primarily due to the mutex acquisition latency. Atomic SC-ideal design in Figure 8 further confirms the fact. Furthermore, the high address conflicts present in streamcluster, dedup and freqmine add significantly to the performance gap between Atomic SC and RMO. The 4th column in Table 2 reports the address conflicts in the shared mutex pool. Even with a mutex pool of size equal to the number of L2 cache entries, the block level false sharing in streamcluster, dedup and freqmine is significantly high. In our work, we assign mutexes in per block basis and do not attempt to eliminate false sharing between mutexes within a cache block. Furthermore, the lower spatial locality present among the memory requests under write-misses in streamcluster, as shown in Table 2, is also responsible for this performance gap.

6.2.2. Sensitivity to the *mutex-lookup* table (*precise-release*). Figure 9 shows that a *mutex-lookup* table with 16 entries is sufficient to achieve performance that matches an idealized design with unlimited *mutex-lookup* table entries.

6.2.3. Sensitivity to the shared mutex pool for *precise-release* and *lumped-release*. If a mutex in the global shared mutex pool is not available, it may be due to either (1) a memory request to the same block from another requestor is active (true positive) or (2) a memory request to a different block from the requesting or different requestor is active (false positive). Figure 10 shows how adding more mutexes, between 256–2048 reduces the false positive rates consistently for few representative benchmarks. However that does not translate to significant performance benefits (bodytrack, blackscholes, canneal,

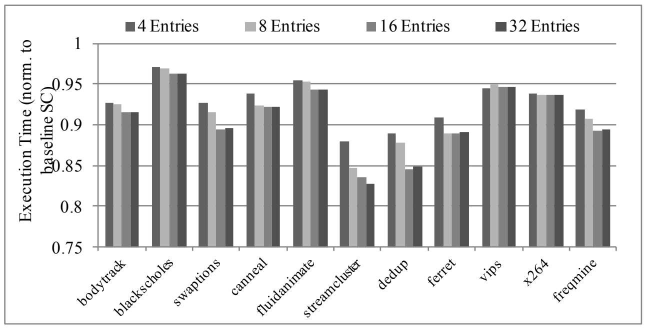

Figure 9. Sensitivity to the *mutex-lookup* table.

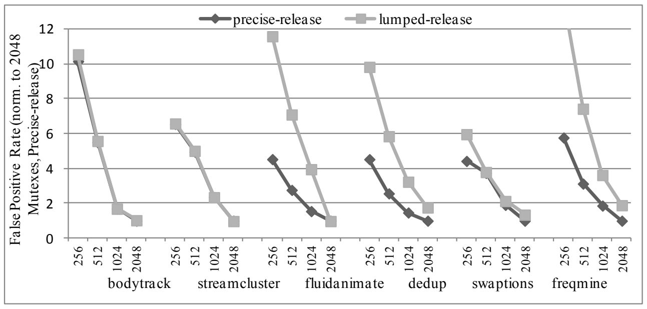

Figure 10. Comparison of false positive rate.

fluidanimate, x264) and can even lead to performance degradation (ferret, swaptions, vips) for *precise-release*, as shown in Figure 11. Any further increase in the pool size shows little or no improvement. Streamcluster, dedup and freqmine show considerable improvement in performance with increase in mutex pool size as shown in Figure 11, indicating that true positives are rare. However, for ferret, swaptions and vips, the slowdown with a larger mutex pool is counterintuitive. A smaller mutex pool adds more pressure on the finite mutex resources, leading to more false positives and stalling the cores more. Highly contended locks in those benchmarks can benefit from the reduced contention caused by the additional stalls from a smaller (mid-range) mutex pool [18].

Figure 12 shows the same sensitivity study to the mutex pool using *lumped-release*. As shown in Figure 10, *lumped-release* causes more false conflicts in contrast to *precise-release*. The false positives increase due to the additional pressure caused by the multiple overlapped write misses and any following requests in shadow of those misses holding mutexes for long periods of time. However *precise-release* eases pressure on the mutex pool and does not suffer as much from the false positives. Intuitively, the increase in false conflicts in *lumped-release* should make the performance worse than *precise-release*. However, there are quite a few cases in Figure 12, where *lumped-release* even leads to more speedup than *precise-release*. This is attributed to the fact that mutex release and false positives have competing impacts on the performance. Although the act of holding mutexes for long in *lumped-release* can cause more false conflicts and stall another core, however it can benefit from prefetching (by holding mutexes for longer) mutexes for the following requests of the core. Of particular note is Figure 12's impact of lazy release on dedup. The effect of prefetching mutexes is more pronounced for dedup than the increase in false conflicts with *lumped-release* across all pool sizes, resulting in additional performance gain in Figure 12 relative to *precise-release* in Figure 11. With a pool of 1024 mutexes, the performance of *lumped-release* very closely approximates the performance of *precise-release*.

6.2.4. Mutex release policy comparison. Overall, *precise-release* performs slightly better than *lumped-release* with a smaller mutex pool. However, this benefit comes at the cost of associated lookups in the *mutex-lookup* table during mutex acquisition and release. A reasonable increase in the mutex pool size in *lumped-release* delivers similar benefit that of a *precise-release*. Given these tradeoffs, we argue that *lumped-release* with a timeout mechanism is more complexity-effective policy of the two.

6.3. Bandwidth increase

Table 3 reports the increase in network bandwidth utilization for Atomic SC relative to the baseline SC design. The increase in link bandwidth is about 0.55%

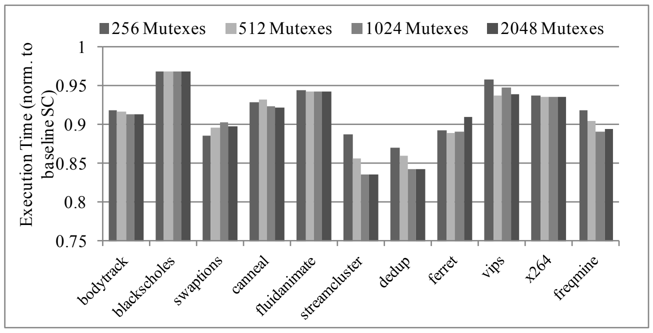

Figure 11. Sensitivity to mutex pool in *precise-release.*

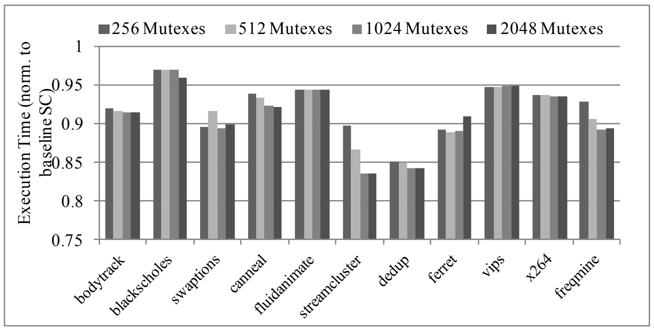

Figure 12. Sensitivity to mutex pool in *lumped-release.*

Table 3. Increase in bandwidth utilization.

Benchmark	% Increase in link b/w	Benchmark	% Increase in link b/w
bodytrack	0.43	dedup	0.55
blackscholes	0.14	ferret	0.19
swaptions	2.36	vips	0.15
canneal	1.89	x264	0.27
fluidanimate	0.45	freqmine	2.12
streamcluster	3.25	-	-

on average. The power reported in Table 5 does not include the additional interconnect power for mutex messages. Considering the fact that the interconnect traffic contributes about 5% [28] of the total chip power, the 0.55% increase will result in negligible power overhead.

6.4. Hardware cost

In our design, the *Release-status* table contains 16 entries to track the store-color numbers from 16 cores.

Table 4. Hardware cost.

Structures	*precise-release*	*lumped-release*
lookup table (per core)	16 entries * (10 bits for mutex-map + 5 bits for store-color + 1 bit for *mutex-release* bit) ~ 32 B	1 bit-vector array * 1024 bits = 128 B
Mutex pool	1024 bits = 128 B	1024 bits = 128 B
Release-status table per directory	16 entries * (4 bits for core-id + 4 bits for store-color) = 16 B	16 entries * (4 bits for core-id + 4 bits for store-color) = 16 B

7. Related work

There are decades of research on efficiently designing an SC hardware that matches the performance of relaxed memory models. Adve and Hill's [20] proposal to enforce SC by reserving cache lines that have been accessed out of order and delaying invalidate acknowledgments for such lines bears some similarity to Atomic SC. Reservation bits, loosely analogous to our mutexes, are kept at each local L1 cache, and upgrade requests to shared lines must visit these local caches to determine the status of the reservations before allowing further requests to proceed. A complex set of modifications are required to a baseline coherence protocol to ensure that requests are correctly forwarded, invalidate acknowledgments appropriately delayed, reservation bits set and flash-cleared at the right time and so on. Only a high-level description is provided, and a quantitative evaluation is missing, so we implemented their *Ack-delay* approach and found that it provides consistently worse performance than Atomic SC (Figure 13). The difference is due to the longer three-hop delay required to check reservation bits at remote sharers and guarantee ordering, vs. our two-hop mutex acquisition delay. Furthermore, reserved cache blocks cannot be replaced in *Ack-delay*, causing more pressure on the L1 cache and a potential increase in miss rates.

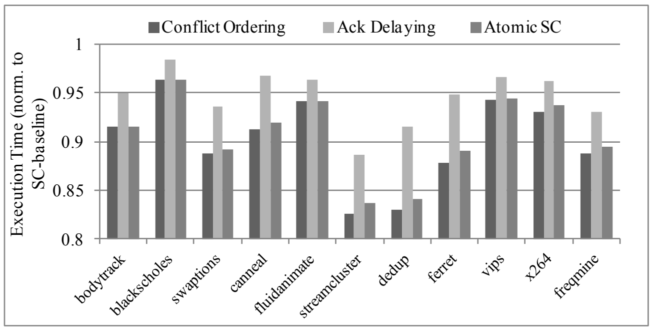

Figure 13. Performance comparison.

Table 5. Comparison of Atomic SC.

	Conflict Ordering [15]	Ack-Delaying [20]	Atomic SC (*lumped-release*)
Power (mW)	164.47	129.38	35.35
Area(μm^2)	250,266.02	185,364.2	85,044.95
Coherence Protocol Complicated?	Yes	Yes	No

A recent proposal for out-of-order processors, conflict ordering [15], is also amenable to implementation with in-order processors, since it does not require speculation and rollback. Instead, similar to Atomic SC, it allows memory operations to complete past pending misses and retire from the ROB, as long as all conflicting memory operations prior to them in the global memory order have completed. Conflict ordering uses an augmented write buffer (AWB) to track pending write sets and also keeps an active bit in each L1 block. The AWB must be searched associatively for each active and inactive invalidation acknowledgement and L1 replacement of active blocks is prohibited (similar to *Ack-delay* above). In addition, conflict ordering necessitates substantial and pervasive changes to the coherence protocol to correctly implement conflict detection. We adapted conflict ordering to an in-order processor and compared its performance to Atomic SC. As seen in Figure 13, both schemes provide comparable performance gains. Atomic SC avoids most of the overheads of these schemes, requiring only distributed mutex pool (128B RAM) and a simple RAM mutex bit-vector (128B RAM) at each core, with no associative lookups or active bits at each L1 block.

As summarized in Figure 13 and Table 5, detailed analysis of these three schemes shows that for comparable or better performance benefit, Atomic SC requires about 3x and 2x less area, while resulting in 4.65x and 3.65x saving in power requirement over [15] and [20] respectively. Considering the fact that recent mobile processors, such as ARM's Cortex-A9 MPCore with 4 processors [33] have stringent power budget (<1W), even a saving of about 100-

150 mW of power using our approach will ease considerably to maintain SC within tight power constraints. RAM and CAM energy and area were analyzed with the NCSU FABMEM tool [29], with event counts collected during simulation.

Among the other prior works, [30] and [31] introduce changes to the coherence protocol to maintain proper SC order non-speculatively for re-ordered references. [32] shows that massively-threaded throughput-oriented processors does not benefit much from weak consistency models when compared to SC due to the presence of SMT and vector operations.

8. Conclusions

In this work, we propose Atomic SC that attempts to reap the benefits of memory reference reordering in simple in-order processors without requiring any speculation or rollback support. The *lumped-release* policy derives a low-complexity hardware design for Atomic SC. The additional hardware resources required to support Atomic SC is considerably lesser in comparison to the prior schemes. Finally, Atomic SC is fully compatible with existing coherence protocols and does not require any protocol modifications, which are error-prone and often lead to hard-to-find bugs and design flaws. On account of simplicity in design and minimal hardware overhead, Atomic SC stands as a suitable alternative to deploy in power-limited in-order many-core designs (e.g., embedded space, mobile processors etc.).

Acknowledgments

We thank Mark Hill and the anonymous reviewers for their insightful feedback, which has improved the content and presentation of this paper. We also thank Abhishek Aggarwal and Pankhuri Taneja for their assistance during the initial phase of this work. This work was supported in part by NSF grants CCF-1116450 and CCF-1318298 and donations from Qualcomm and Oracle research.

References

[1] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor program. IEEE Computer, 1979.
[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A Tutorial. IEEE Computer, 1995.
[3] K. Gharachorloo et al. Two techniques to enhance the performance of memory consistency models. In *ISCA*, 1991.
[4] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.
[5] L. Ceze et al. BulkSC: bulk enforcement of sequential consistency. In *ISCA*, 2007.
[6] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *PACT*, 2002.
[7] C. Gniady, B. Falsafi and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, 1999.
[8] M. D. Hill. Multiprocessors should support simple memory consistency models. IEEE Computer, 1998.
[9] P. Ranganathan et al. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *SPAA*, 1997.
[10] T. F. Wenisch et al. Mechanisms for store-wait-free multiprocessors. In *ISCA*, 2007.
[11] H. Cain and M. H. Lipasti. Memory ordering: a value-based approach. In *ISCA*, 2004.
[12] L. Hammond et al. Transactional memory coherence and consistency. In *ISCA*, 2004.
[13] A. Condon and A. J. Hu. Automatable verification of sequential consistency. In *SPAA*, 2001.
[14] A. Landin et al. Race-free interconnection networks and multiprocessor consistency. In *ISCA*, 1991.
[15] C. Lin, R. Gupta, B. Rajaram. Efficient sequential consistency via conflict ordering. In *ASPLOS*, 2012.
[16] C. Lin et al. Efficient sequential consistency using conditional fences. In *PACT*, 2010.
[17] Y. Duan, A. Muzahid and J. Torrellas. WeeFence: toward making fences free in TSO. In *ISCA*, 2013.
[18] R. Rajwar et al. Improving the throughput of synchronization by insertion of delays. In *HPCA*, 2000.
[19] C. Blundell et al. InvisiFence: performance-transparent memory ordering in multiprocessors. In *ISCA*, 2009.
[20] S. V. Adve and M. D. Hill. Implementing sequential consistency in cache-based systems. In *ICPP*, 1990.
[21] Hu et al. Godson-3: A scalable multicore RISC processor with x86 emulation. IEEE Micro, 2009.
[22] S. Franey et al. Accelerating atomic operations on the GPU for broader applicability. In *NOCS*, 2013.
[23] D. Marino et al. A case for an SC-preserving compiler. In *PLDI*, 2011.
[24] A. Singh et al. End-to-End SC. In *ISCA*, 2011.
[25]http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php
[26] The Gem5 simulator: http://gem5.org/
[27] C. Bienia et al. The PARSEC benchmark suite. In *PACT*, 2008.
[28] M. Hayenga et al. Pitfalls of ORION-Based simulation. In *WDDD*, 2012.
[29] people.engr.ncsu.edu/ericro/research/fabscalar.htm
[30] C. Scheurich et al. Correct memory operation of cache-based multiprocessors. In *ISCA*, 1987.
[31] A. R. Lebeck et al. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *ISCA*, 1995.
[32] B. A. Hechtman et al. Exploring memory consistency for massively-threaded throughput-oriented processors. In *ISCA*, 2013.
[33] http://www.arm.com/files/pdf/armcortexa-9processors.pdf