zFENCE: Data-less Coherence for Efficient Fences

Shaizeen Aga Abhayendra Singh Satish Narayanasamy
University of Michigan, Ann Arbor
{shaizeen, ansingh, nsatish}@umich.edu

ABSTRACT

Efficient fences will not only help improve the performance of today’s concurrent algorithms, but could also pave the way for the adoption of stronger memory models such as Sequential consistency (SC). However, the cost of fences in commodity processors remains prohibitively expensive.

A hardware fence only requires that all memory accesses preceding a fence in the program order are performed before the fence and its following memory accesses are performed. But, it does not require that these operations are completed in that order.

In this work we observe that a significant fraction of fence overhead is caused by stores that are waiting for data from memory. We propose the zFENCE architecture that exploits this observation for efficiently implementing a fence by introducing the capability to grant coherence permission for a store much earlier than servicing its data from memory. We show that zFENCE eliminates fence overhead in a majority of scenarios, and helps bridge the performance gap between SC and TSO runtime memory models for a low design cost.

Categories and Subject Descriptors
C.1.4 [Computer Systems Organization]: PROCESSOR ARCHITECTURES—Parallel Architectures

Keywords
Memory Consistency; Sequential Consistency; Fences; Data Less Coherence; Parallel Programming

1. INTRODUCTION

While the merits of the sequentially consistent (SC) memory model are well known, the concern for its performance overhead has prevented its adoption in mainstream languages. To guarantee language-level SC, both compiler and hardware optimizations need to be restricted to be SC-preserving. Recent studies have produced encouraging results showing that it is indeed possible to preserve SC during compiler optimizations for a fairly low performance cost (less than 4% on average) [32]. However, preserving SC at runtime in the presence of hardware optimizations continues to be a challenge.

Today’s mainstream concurrent programming languages, including C++ [8], Java [31] and OpenCL [25], have adopted variants of the data-race-free (DRF) memory model [2]. While these languages guarantee SC for correctly written data-race-free programs, they guarantee either no semantics [8] or weak and complex semantics for programs with data races [31].

While existing hardware does not guarantee SC, in order to support the DRF-0 model, their ISAs provide support for a full fence. Fences are used by a DRF-0 compliant compiler to implement memory accesses to synchronization variables (annotated as volatile in Java or atomic in C++). Completion of a full fence instruction in hardware guarantees that all the instructions in the same thread that precede the fence in the program order have been performed, before performing any instruction after the fence.

Compiler can guarantee SC on relaxed DRF-0 compliant hardware by conservatively inserting a fence for each memory access [13]. Memory model constraints enforced on hardware by these compiler-inserted fences are equivalent to the constraints that a hardware would have to anyway honor to guarantee SC.

Unfortunately, fences are prohibitively expensive in today’s commodity hardware (about 20 cycles even when store-buffer is empty, and an order of magnitude higher when it contains store misses). We believe that if fences become inexpensive, language developers would be more willing to adopt stronger memory model standards, such as SC. Therefore, instead of asking for SC hardware, proponents of a language-level SC memory model should simply ask for efficient hardware support for fences. As efficient fences are useful to improve the performance of existing DRF-0 compliant concurrent programs, processor industry has a clear short-term incentive to optimize fences.

In this paper, we present zFENCE, a simple optimization that can provide near zero overhead hardware fences. We believe that zFENCE can remove the remaining performance hurdle that is holding back language designers from adopting SC, without requiring complex changes to commodity hardware designs.

Past solutions for realizing efficient SC hardware and fences employed complex out-of-window speculation [7, 4, 18, 34, 24, 20, 10, 37], global mechanisms that continuously...
check and prevent memory ordering violations while reordering memory accesses [30, 13, 22], or additional hardware and system support that identify and optimize safe data-race-free accesses [35].

We observe that long latency store misses serviced by main memory are by far the dominant cost of fences in a modern TSO multi-core processor. This cost can be easily optimized using hardware mechanisms that are much simpler than past solutions for efficient fences and SC.

In an efficiently implemented out-of-order processor with in-window speculation [18], the only intrinsic performance cost due to a fence is the cost of draining the store buffer before completing a fence. Furthermore, a huge fraction of the cost of draining the store buffer is due to long latency store misses serviced by main memory. It is less critical to optimize the low-latency store misses that are serviced either by an on-chip second-level cache or by another processor core's cache. Multi-processor SC designs proposed before the multi-core era perhaps did not consider this cost-benefit trade-off, as off-chip communication between multiple processors could be as expensive as fetching data from main memory.

We discuss a simple hardware design, zFENCE, which significantly reduces the impact of store misses serviced by main memory in enforcing fence constraints. zFENCE enables the directory to grant the exclusive coherence permission for a store before servicing the data. As we show in this paper, the coherence permission to a cache block can be serviced much faster than data for critical long latency store misses. A fence can be completed as long as all stores in the store buffer have acquired and retain their exclusive permission [16]. It is not necessary for those stores to have completed by writing their new values to data in cache or memory.

zFENCE optimization is fairly simple. On receiving a store miss request that needs to be serviced by the slow main memory, instead of waiting for the data fetch from the memory to complete, the on-chip directory cache responds immediately by sending a message granting coherence permission (but without data) for the requested block to the requester. Fences stall at the head of the re-order buffer only if there are pending stores in the store-buffer without the coherence permission. The only additional constraint that zFENCE needs to enforce to guarantee fence semantics is that a processor core does not relinquish the coherence permission to a cache block before any pending store in the store buffer to that cache block is completed. Fortunately, holding coherence permissions for pending stores cannot cause a deadlock, because they are not waiting for other processor cores but simply waiting for the data fetch from the main memory to complete.

Our optimization exceeds or equals most of the benefits of the optimizations that targets relaxing the memory ordering constraint for the private data [35], because, zFENCE optimizes any L2 store miss to private data, without relying on conservative compiler analysis or page-level tracking of private locations [35]. In addition, any store misses to shared data serviced by memory are also optimized. However, zFENCE does not optimize relatively low-latency store misses to cache blocks owned by other processor cores. We show that this is a worthy trade-off for realizing a complexity-effective design.

Unlike single-socket architectures, in a multi-socket architecture, coherence permissions cannot always be obtained through on-chip communication. Nevertheless, we find that zFENCE is still effective for multi-socket architectures for at least two reasons. First, a significant fraction of stores misses are to private data, whose home directory tends to be located on the same chip as the requesting core. Second, even when a store miss is serviced by a remote memory, zFENCE can be effective as the off-chip directory could grant the permission without waiting for the data fetch from memory to complete.

We study the use of zFENCE in guaranteeing SC at runtime for several x86 macro-benchmarks (SPLASH-2 [38], PARSEC [6], and Apache). We use our compiler to conservatively insert fences for all memory accesses to guarantee SC at runtime on an RMO processor. We show that the performance of this SC system is only 1.54% slower than TSO and 2.93% slower than RMO without the additional fences we insert. zFENCE based SC system also outperforms a recent SC hardware proposal [35], which is also more complex than zFENCE, as it requires store-to-load forwarding from two store buffers and hardware to determine safe accesses (private and shared read-only accesses).

This paper makes the following contributions:

- In modern multi-core processors, we observe that store misses serviced by main memory constitute the dominant cost of fence overhead. Optimizing this cost is largely adequate in eliminating memory ordering costs of fences.
- We propose zFENCE, which enables the directory to grant coherence permission before servicing data from main memory. This capability allows a fence to complete without waiting for data for stores to be fetched from main memory.
- We show that zFENCE optimization eliminates most of the fence overhead. We study both single and multi-socket systems, and compare SC implemented using zFENCE with TSO and RMO processors. We also show that our SC system outperforms a recent SC hardware design [35].

2. BACKGROUND AND MOTIVATION

In this section, we discuss the motivation for SC and the challenges in supporting efficient fences.

2.1 Need for SC

Programmers commonly take two programming abstractions for granted when analyzing a program snippet: program order, which requires that instructions in a thread execute one after the other in the order they appear in the program text; and shared memory that behaves as a map from addresses to values with each memory operation taking effect immediately. The memory model that preserves this intuitive program behavior for multithreaded programs is formally known as sequential consistency (SC) [26].

Before the arrival of commodity multi-core processors, the issue of memory models was mostly delegated to processor architects and assembly-level programmers. Ubiquitous adoption of parallel programming made it essential to standardize concurrency semantics at the language-level. Unfortunately, memory models of mainstream languages such as
C++ and Java chose to provide SC only for data-race-free programs [2, 31, 8]. For racy programs, these languages either provide no semantics or a weak semantics, which permit obscure thread interleavings that are hard for a programmer to reason about. Furthermore, proving the correctness and safety of various compiler and hardware optimizations under these weak memory models continues to be a challenge [36, 9].

2.2 Processor Actions

We clarify a few common terms that describe events in a processor core. An in-flight instruction is one that has been fetched into the processor pipeline but has not yet been committed. In an out-of-order processor, instructions may execute out-of-order but must commit from the reorder buffer (ROB) in the program order. Two memory operations in different threads are said to conflict if they access the same memory location and at least one of them is a write.

A load or a store is performed if its actions are observable to all the processors [14]. In most processors, a load is performed when it is committed from the ROB. In SC, stores are performed at commit. In TSO or RMO, a store may commit from the ROB and be placed in a store buffer without being performed. A store is completed when it eventually retires from the store buffer and its value is written to cache or memory.

2.3 Challenges in Realizing Efficient Fences

Compilers can guarantee SC on commodity processors by inserting fences between stores and loads. Thus, efficient fences can allow us realize efficient SC. However, realizing an efficient hardware fence design remains a challenge. It is not only sufficient to produce an efficient design but the modifications needed to a commodity processor should be simple enough that hardware vendors would be willing to adopt it in practice.

Though there are various flavors of fences in modern processors, we consider a full fence. A full fence requires that all the preceding stores and loads in the program order are performed before that fence and its following accesses (all in the same thread) are performed.

In-window speculation [18] allows in-flight loads to speculatively execute (i.e., read from memory) out-of-order ahead of preceding memory operations and fences, while still guaranteeing that they appear to perform at the time of their commit. Since TSO disallows load-load reordering, most modern x86 processors also implement in-window speculation to allow out-of-order execution of loads. In-window speculation is also useful in a relaxed memory-order (RMO) processor as it would allow loads to execute ahead of memory fences in the pipeline. Hence, we assume support for in-window speculation in all the hardware designs we study.

In window speculation, however, does not help us hide the overhead of fence constraint for stores, as stores have to be performed before performing a fence. To guarantee this fence constraint, a conservative hardware stalls the commit of a fence from the ROB when the store buffer is not empty. This store buffer drain is the primary overhead in enforcing fence constraint.

3. zFENCE OVERVIEW

As we described in the previous Section 2, waiting for the store buffer drain is the primary cost associated with a fence. We present an overview of the zFENCE optimization that addresses this cost.

3.1 Store Buffer Drain is not Necessary to Preserve Fence Semantics

To implement fences efficiently, we exploit the crucial observation [16]: it is not necessary to complete a store in order to perform it. A store is performed at an instant when a load issued to the same address by any processor returns that store’s new value. To perform a store, it is not necessary to have completed the store actions by merging its new value to the architectural memory state. In past this observation has been used to reduce the latency of store by sending eager acknowledgments [19] for invalidation requests. However, past approaches do not optimize for the case where stores go to main memory.

In zFENCE we propose to exploit the above observation for stores that do not require any invalidation and receive data from memory. To perform a store, it is sufficient to acquire the coherence permission for that store’s address, provided it is not released until its actions are completed.

Figure 1 provides an example to clarify the above observation. In a conventional fence implementation (middle figure in 1), processors P1 and P2 each have to complete their respective stores before their fences can be committed. This overly constrained fence implementation unnecessarily stalls the pipeline, delaying the commit of instructions following the fence (loads in our example).

In zFENCE, as shown in the rightmost figure in 1, a processor stalls a fence commit only till the pending stores in the store-buffer acquire their coherence permissions (without waiting for data). While stores are waiting for the data fetch from memory to complete, later fence and loads are allowed to complete out-of-order.

Even though the completion of write W_P is delayed, fence semantics are not violated. Once W_P is performed by acquiring its coherence permission, any later read to P is guaranteed to observe the new value of W_P, as it would have to wait for W_P to complete and release the coherence permission. This is ensured as we do not relinquish the coherence permission acquired for W_P until it is completed. This will not cause a deadlock as pending stores are waiting for data fetch from memory and not on an action by any other processor. In summary, once W_P is performed, the dotted edge in Figure 1 representing the execution order that is necessary to violate fence semantics can never happen.
3.2 Decoupling Coherence Permission from Data Fetch

For zFENCE to commit fences faster, acquiring coherence permissions should be faster. We observe that most fence overhead is caused by pending stores serviced by main memory. zFENCE specifically targets such long latency stores and expedites acquisition of their coherence permission by decoupling coherence permission acquisition from data fetch.

To accomplish this, when a directory receives an exclusive coherence request to a cache block that needs to be serviced by the main memory (i.e., the cache block is in Invalid state), the directory immediately responds back with a special coherence reply, zMESSAGE, which grants exclusive coherence permission to the requester without sending the data. Simultaneously, it instructs the memory controller to fetch the required data from memory and forwards it to the requester once it is fetched. Thus, coherence permission acquisition is a magnitude faster since we take off-chip data access off the critical path.

We did not attempt to decouple the coherence permission from data for store misses that are serviced by either the on-chip L2 cache or by another processor core’s private cache. Coherence transactions necessary to service these stores are relatively low latency operations as they do not involve a main memory access. As a result, the coherence permissions for a large fraction of these store misses is acquired even before the store reaches the head of the ROB due to the coherence prefetch request optimization [18]. Modern x86 TSO processors commonly support this optimization where a processor core issues a coherence read-exclusive prefetch request as soon as a store’s address is resolved in the processor pipeline.

4. zFENCE ARCHITECTURE

In this section, we describe our proposed zFENCE architecture that enables fences to commit without requiring expensive store buffer drain. zFENCE mostly relies on the directory based coherence infrastructure of a commodity processor.

4.1 Baseline Fence Design

We assume a single-socket processor that supports MOESI coherence protocol using distributed directories (we describe multi-socket design in Section 4.5). We assume directory caches are inclusive [12]. That is, a directory cache miss to a cache block implies that it is not cached by any processor core (i.e., cache block is in Invalid state).

Each processor core contains a private L1 cache and a private L2 cache. We also assume two optimizations that are commonly supported in TSO processors. The first optimization is in-window speculation [18] that enables out-of-order execution of loads across fences, while providing an illusion that they are performed in the program order (Section 2.3). The second optimization is issuing a coherence read-exclusive prefetch request [18] for stores as soon as a store’s address is resolved in the processor pipeline, which helps hide some part of the store miss latency.

In our baseline architecture, we assume that a fence is committed from ROB only when all preceding stores are completed. This is achieved by draining the store buffer. This may incur an expensive pipeline stall, especially if stores in the store buffer have missed in on-chip caches and are waiting for data to be fetched from the slow main memory. This overhead constitutes the bulk of the performance cost associated with fences.

4.2 Directory Controller Extensions

The directory controller is responsible for decoupling coherence permission from data only while servicing exclusive permission request to a cache block whose data resides in the main memory (that is, when the cache block is in the Invalid state). For the reasons described in Section 3.2, zFENCE directory does not decouple coherence permission from data while handling any other coherence request to blocks in any other state, and hence its actions to service them remain the same as in the baseline.

On receiving an exclusive coherence request to a cache block in the Invalid state, the zFENCE directory controller immediately responds with a special zMESSAGE granting exclusive coherence permission to the requester. This is the only addition to the baseline directory controller. Rest of the actions are the same as in the baseline, which includes sending a message to the memory controller instructing it to fetch the appropriate data, and blocking all later coherence requests to the same block till the data is fetched and sent to the requester.

4.3 Processor Core Extensions

In order to commit a fence in zFENCE, we need to test if all preceding stores have their coherence permissions. In zFENCE, we expedite coherence permission acquisition for long latency stores by decoupling coherence permission from data. Thus, processors could acquire coherence permission without data and as an effect we need to provision to track this state.

Modern out-of-order processor cores use a content addressable store queue (different from store buffer) to keep track of all in-flight stores. The store queue is used to enable store-to-load forwarding. In zFENCE, we propose to add an additional bit, zBit, to each store queue entry as well as to each store buffer entry. On receiving the zMESSAGE or the coherence permission with data for a cache block, processor core searches the store queue and store buffer for the stores to that cache block. Matching store entries get their zBit set to remember that the exclusive coherence permissions have been acquired.

When a store reaches the head of the ROB, it is committed to an unordered coalescing store buffer and its zBit is copied over to the corresponding store buffer entry. The store waits here for its cache block data to arrive and then it completes.

A fence is allowed to commit from the head of ROB only when all stores in the store buffer have their zBit set. Otherwise, the fence is stalled.

It is also necessary to ensure that exclusive permission to a cache block is not lost when there is a pending store to that block in the store buffer with its zBit set, as that could violate fence semantics. Note that in zFENCE we would commit a fence only when all pending stores have their zBit set and thus it is safe to relinquish coherence permissions for pending stores if their zBit is not set. A processor core could have to relinquish coherence permission either to service an incoming coherence invalidation request or to allocate space for a cache block requested by a later memory operation. Even in a blocking directory protocol, it is possible for a processor to get a forwarded request while there it has a pending store to same location as stores to same location...
could prefetch permissions for subsequent stores. zFENCE forces these two actions to a cache block to stall until the pending store to that cache block is completed.

Stalling the above two actions causes insignificant performance overhead. Stalling the eviction of a recently accessed cache block for which there is a pending store is an insignificant performance issue (and perhaps could be a performance win). Stalling a cache invalidation could potentially delay the operation in the requesting processor but in practice we observe that this case is not common.

Before evicting a cache block from L1 or downgrading its exclusive permission, it is necessary to search the store queue. The zBit for any matching entry in the store queue needs to be reset to indicate that the coherence permission for the corresponding store is no longer available. Note that stores in store queue are in-flight and resetting of their zBit does not effect their completion. When the store reaches the head of ROB, a binding cache coherence request will be issued for it.

4.4 Correctness and Deadlock Freedom

In this section, we provide an informal proof sketch for correctness and deadlock-freedom for zFENCE.

Loads preceding a fence are always committed before the fence. A load that follows a fence, even though could execute out-of-order with respect to the fence, in-window speculation [18] guarantees that it appears to have performed at the time of its commit. For stores, zFENCE guarantees fence semantics by ensuring that all stores preceding a fence in the program order acquire their coherence permissions before we commit that fence. Furthermore, coherence permissions to pending stores in the store buffer with their zBit set are not relinquished before they are completed. These are sufficient constraints to guarantee full fence semantics.

zFENCE is deadlock-free. The only stall it introduces is the actions that cause a processor core to relinquish the exclusive coherence permission to a cache block for which there is a pending store in the store buffer. Having already acquired the coherence permission, a pending store in the store buffer is either waiting for the data response or cache write port access in order to complete its actions. When these pending stores complete, any actions stalled on their behalf can be allowed to proceed.

4.5 Multi-Socket Systems

zFENCE is also effective in reducing the overhead of fences in a multi-socket architecture, though to a lesser degree. We study the multi-socket architecture shown in Figure 2. Each processor chip is a chip-multiprocessor (CMP). We assume two-level coherence for such systems: intra-cmp coherence and inter-cmp coherence. Intra-cmp coherence manages permissions between cores within a processor chip. This is supported by the first-level directories. In zFENCE, this remains unchanged.

Inter-cmp coherence is involved while managing coherence for cache blocks serviced by the main memory or by a remote processor core on another chip. This is supported by the second-level directories. Since the second-level directories are responsible for granting coherence permissions for cache blocks residing in main memory, we extend it to support zFENCE. That is, when a second-level directory receives request to a cache block that resides in the local memory it manages, it immediately responds back to the requester with the zMESSAGE.

We assume that the operating system allocates pages in the memory that is local to the first processor that requests it. This ensures that a significant fraction of the private data accessed by threads executing in a processor are allocated in the local memory. Any store miss to a private cache block stored in a processor’s local memory would enjoy all the performance benefits in zFENCE discussed before.

If a store miss request is serviced by a remote memory, it would entail off-chip communication with the second-level directory located in a remote processor chip. Nevertheless, zFENCE is still effective in reducing the fence overhead caused by such stores as the remote directory can respond with the zMESSAGE much faster when compared to servicing data from the remote memory.

One limitation of zFENCE is that it does not help reduce the overhead of store misses that are serviced by other processor cores, as we do not decouple data from coherence for servicing such stores. Coherence prefetch optimization is largely effective in fetching permissions before the stores get to the head of the ROB in single-socket systems where the permissions are serviced by on-chip cores. However, in multi-socket systems, when permission to a store is serviced by remote off-chip cores, it may result in some overhead as it requires off-chip communication.

4.6 Illustration

Figure 3 (b) illustrates the program execution shown in Figure 3 (a) under two different architectures: baseline RMO with traditional fences, and RMO with zFENCE. In this example, a program executes two threads on two processor cores P1 and P2 as shown in Figure 3. The initial state of all three variables P, D and F is zero. Under fence semantics, it is not possible for loads in P2 for P and D to have values 0 and 1 respectively. Fence semantics are violated only if both conflict orders shown in Figure 3 are observed in an execution. A memory operation that results in a cache miss is represented using a gray shade.

RMO using baseline fences conservatively stalls the commit of a fence due to pending store miss STP and consequently all its following operations in P1 (●). Though required to preserve fence semantics, this enforces an unnecessarily strict constraint, which could result in long pipeline stalls, especially if STP is serviced by the main memory.

zFENCE, on the other, hand commits the fence after ensuring that P1 owns the exclusive coherence permission for the cache block P which is the only pending store. While
ST\(P\) is waiting for its block to be fetched from memory, later operations in \(P_1\) are allowed to commit from the ROB (3). This allows \(P_1\) to commit and complete ST\(D\) whose value is then observed by LD\(D\) in \(P_2\). Thus, solid conflict edge in Figure 3 (a) is observed in this execution. However, zFence preserves fence semantics by ensuring that the dotted conflict edge is not observed. This is ensured by enforcing a constraint that \(P_1\) does not release its coherence permission for \(P\) when there is a pending store to \(P\). When \(P_1\) receives a coherence request for \(P\) from \(P_2\), it simply stalls the response till ST\(P\) is completed, which in turn stalls LD\(P\) in \(P_2\) (4). When \(P_1\) eventually releases the coherence permission for \(P\), \(P_2\) is guaranteed to see the new value written by ST\(P\), preserving fence semantics.

5. STRONGER MEMORY MODELS USING zFENCE

In this section we highlight the performance benefits that zFence enables for stronger memory models.

A compiler can preserve SC at runtime on RMO hardware by conservatively inserting a fence after every memory access. Note that this does not necessarily imply addition of more instructions. With the shift to 64-bit ISA we expect more freedom in finding unused bits in the ISA to convey that an instruction has fence semantics. Only if this is not an option, we can resort to adding fence instructions, which all DRF0-compliant ISAs (ARM, x86, PowerPC) support.

Compared to using traditional fences, zFence based SC system enjoys the following performance benefits:

- A load can commit even if there are pending stores in the store buffer waiting for the data response from the main memory. The fence associated with a load only needs to wait for the pending stores to acquire their coherence permissions.

- Stores can be committed to unordered coalescing store buffer as soon as preceding stores have acquired their coherence permissions.

- Pending stores in zFence’s unordered store buffer can be completed and retired from the store buffer out-of-order.

- Pending stores in the store buffer can be coalesced.

Notice that the last two benefits listed above are an advantage over a TSO processor, which needs to ensure in-order retirement of stores from its FIFO store buffer.

Relaxed memory model like TSO can also benefit from zFence. TSO mandates ordered store retirement. This can be accomplished with a fence between every two consecutive stores. zFence helps TSO as it enables an unordered coalescing store buffer for TSO which would let stores complete out-of-order while still honoring TSO constraints.

6. EXPERIMENTAL EVALUATION

We study in this section the efficiency of zFence and its use in supporting stronger memory model.

6.1 Configurations Studied

Our end goal is to enable a language-level SC memory model. Therefore, in all of our experiments we use the binaries compiled using the publicly available SC-preserving version of LLVM compiler [32]. This compiler preserves SC across compiler optimizations but does not guarantee SC on relaxed memory models such as TSO or RMO.

The goal of our experiments is to understand the performance overhead of enforcing SC at runtime and compare it to the performance of more relaxed Total Store Order (TSO) (similar to x86 model) and RMO. To evaluate the performance overhead of SC, we studied three configurations. The first configuration is a baseline SC processor design, where each load requires a store-buffer drain and stores are retired in the program order. We refer to this design simply as SC. The second design assumes an RMO processor with zFence support. We use the compiler to generate fences for each memory access in order to guarantee SC on this processor. Note that this does not require adding extra instructions as a single bit (lock-prefix like bit in x86) suffices to convey that a memory instruction is to be treated as a fence. Original fences produced by the baseline compiler (e.g., for atomic accesses) are also inserted. We refer to this system as zSC.

We also compared against a recent non-speculative SC design [35] which we refer as type-SC. Type-SC delivers high SC performance without employing complex out-of-window speculation by relaxing SC constraints for safe memory op-
Table 1: Simulator Parameters

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CMP: 16 cores on chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Multi-chip: 4 CMP's each with 4 cores on chip</td>
</tr>
<tr>
<td>Fetch, Execute, Commit</td>
<td>4 instructions (max 2 loads and 1 store) per cycle per core</td>
</tr>
<tr>
<td>Store Buffer</td>
<td>SC: TSO: 8-byte 64-entry FIFO buffer</td>
</tr>
<tr>
<td></td>
<td>RMO: 64-byte 8-entry unordered coalescing buffer</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64 KB per-core private, 4-way set associative, 64B block size, 2-cycle hit latency, write-back</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512KB per-core private, 4-way set associative, 64B block size, 10 cycle hit latency</td>
</tr>
<tr>
<td>Coherence</td>
<td>CMP: MOESI directory protocol</td>
</tr>
<tr>
<td></td>
<td>Multi-chip: Hierarchical MOESI directory protocol</td>
</tr>
<tr>
<td>Directory</td>
<td>Distributed, 10-cycle lookup latency</td>
</tr>
<tr>
<td>Interconnect</td>
<td>CMP: Torus-2D topology, 256-bit link width, 1ns link latency</td>
</tr>
<tr>
<td></td>
<td>Multi-chip: On-chip Ring, 256-bit link width, 1ns latency</td>
</tr>
<tr>
<td></td>
<td>Off-chip all-to-all, 32-bit link width, 20ns cycle latency</td>
</tr>
<tr>
<td>Memory</td>
<td>160-cycle DRAM lookup latency</td>
</tr>
</tbody>
</table>

We target a coverage ratio of 1.6X for our directory caches. If the cache block of an entry that is evicted from the directory cache is not in Invalid state, a recall [12] is issued to invalidate its permissions in processor’s private caches to ensure inclusivity.

6.3 Benchmarks

We study applications from PARSEC [6], SPLASH-2 [38] and a commercial web server benchmark, Apache. For the applications in PARSEC we used simlarge inputs. For barnes in SPLASH-2, we used 65536 n-body simulation, and for Apache we used the SURGE [5] benchmark.

For Apache, we warmed up the caches and micro architectural state for 10000 transactions and then collected simulation statistics over another 10000 transactions. For barnes, we simulated the entire parallel section after warming the L2 cache with an execution that produced at least as many L2 cache misses as the number of L2 cache lines per core.

For benchmarks in PARSEC, due to long simulation time it was not feasible to simulate the entire parallel section and hence sampling was employed. We took checkpoints for five samples spanned across most of the parallel section. In each simulation run we warmed up the caches by restoring cache state at the end of the last checkpoint. To warm-up rest of the micro-architectural state we also ran each checkpoint in the warm-up phase for 1 million user stores per core. After the warm-up, we measured the performance by simulating 160 million user stores across all cores.

6.4 zSC compares favorably to TSO and RMO

We study applications from PARSEC [6], SPLASH-2 [38] and a commercial web server benchmark, Apache. For the applications in PARSEC we used simlarge inputs. For barnes in SPLASH-2, we used 65536 n-body simulation, and for Apache we used the SURGE [5] benchmark.

For Apache, we warmed up the caches and micro architectural state for 10000 transactions and then collected simulation statistics over another 10000 transactions. For barnes, we simulated the entire parallel section after warming the L2 cache with an execution that produced at least as many L2 cache misses as the number of L2 cache lines per core.

For benchmarks in PARSEC, due to long simulation time it was not feasible to simulate the entire parallel section and hence sampling was employed. We took checkpoints for five samples spanned across most of the parallel section. In each simulation run we warmed up the caches by restoring cache state at the end of the last checkpoint. To warm-up rest of the micro-architectural state we also ran each checkpoint in the warm-up phase for 1 million user stores per core. After the warm-up, we measured the performance by simulating 160 million user stores across all cores.

6.4 zSC compares favorably to TSO and RMO

We study applications from PARSEC [6], SPLASH-2 [38] and a commercial web server benchmark, Apache. For the applications in PARSEC we used simlarge inputs. For barnes in SPLASH-2, we used 65536 n-body simulation, and for Apache we used the SURGE [5] benchmark.

For Apache, we warmed up the caches and micro architectural state for 10000 transactions and then collected simulation statistics over another 10000 transactions. For barnes, we simulated the entire parallel section after warming the L2 cache with an execution that produced at least as many L2 cache misses as the number of L2 cache lines per core.

For benchmarks in PARSEC, due to long simulation time it was not feasible to simulate the entire parallel section and hence sampling was employed. We took checkpoints for five samples spanned across most of the parallel section. In each simulation run we warmed up the caches by restoring cache state at the end of the last checkpoint. To warm-up rest of the micro-architectural state we also ran each checkpoint in the warm-up phase for 1 million user stores per core. After the warm-up, we measured the performance by simulating 160 million user stores across all cores.

6.4 zSC compares favorably to TSO and RMO

We study applications from PARSEC [6], SPLASH-2 [38] and a commercial web server benchmark, Apache. For the applications in PARSEC we used simlarge inputs. For barnes in SPLASH-2, we used 65536 n-body simulation, and for Apache we used the SURGE [5] benchmark.

For Apache, we warmed up the caches and micro architectural state for 10000 transactions and then collected simulation statistics over another 10000 transactions. For barnes, we simulated the entire parallel section after warming the L2 cache with an execution that produced at least as many L2 cache misses as the number of L2 cache lines per core.

For benchmarks in PARSEC, due to long simulation time it was not feasible to simulate the entire parallel section and hence sampling was employed. We took checkpoints for five samples spanned across most of the parallel section. In each simulation run we warmed up the caches by restoring cache state at the end of the last checkpoint. To warm-up rest of the micro-architectural state we also ran each checkpoint in the warm-up phase for 1 million user stores per core. After the warm-up, we measured the performance by simulating 160 million user stores across all cores.
6.5 Comparison to Type-SC

Figure 6 also shows that zSC outperforms a recent non-speculative SC design [35] (type-SC). Average overhead of type-SC is about 4.91% on average for single-socket and 5.74% for multi-socket; which is brought down by zSC. The overhead in type-SC is slightly higher than the reported result in [35]. We believe this is due to longer samples in our simulations, and also we carry over the unsafe state for pages across different samples. As a program executes longer, more and more pages would be classified as shared unsafe locations. While this overhead could be optimized by resetting classifications periodically, it would entail additional complexity.

zSC compares favorably with type-SC not only with respect to performance but also in terms of complexity. Type-SC requires two store buffers (one each for safe and unsafe stores) which introduces complexities in the store-to-load forwarding logic. Also, additional software and hardware support is needed for statically and dynamically classifying memory operations as safe and unsafe.

6.6 Detailed Analysis of Memory Ordering Overhead

In Figure 5, we analyze various sources of memory ordering overhead for different processor designs we studied for the multi-socket system. We attribute each processor core’s cycle to one of the following categories and calculate their contributions to average CPI per core. Since we model a commit-width of four instructions, a cycle in which all four instructions at the head of the ROB are committed or commit is stalled for a reason other than ones being considered here is classified under other. Otherwise, the cycle is attributed to one of the following reasons that caused the commit stall. The first instruction in the program order that caused the stall is referred to as the Head. At the end of each stall category we indicate the processor configurations that could incur the stall.

- **cache-miss-ld:** Head is a load miss waiting for the data. (All configurations)
- **sb-full:** Head is a store and the store buffer is full. (All configurations)
- **sb-drain-fence:** Head is a fence waiting for the store buffer to drain. (TSO and RMO)
- **sb-drain-load:** Head is a load. It is ready to perform, but stalled for store buffer drain. (Baseline SC and Type-SC)
- **coherence-permission-stall:** Head is a lock-prefixed store waiting to acquire coherence permission as explained in Section 6.4. (zSC).
- **other:** None of the above or core committed commit-width instructions.

In Figure 5 the cycles attributed to the other categories varies across different contributions. This is due to an artifact of our classification strategy. A load stalled for store buffer drain will contribute to sb-drain-load in baseline SC and will mask any potential stalls contributing to other
bucket. Since loads do not contribute stall cycles to sb-drain-load in RMO or zSC configurations, we could have higher other stall cycles that were masked in SC.

Stalls caused by sb-drain-load dominate the commit stage stalls in both baseline SC and type-SC configurations. We see considerable sb-drain-load stalls in facesim, barnes and apache in baseline SC which type-SC helps eliminate to certain extent, while zSC eliminates this overhead completely.

Since baseline SC, type-SC, and TSO employ a FIFO store buffer, they suffer from commit stalls due to FIFO store buffer being full. On the other hand RMO and zSC employ unordered coalescing store buffers, and thus they do not incur these sb-full stalls significantly.

While zSC is successful in reducing the overhead of ensuring SC in most benchmarks, apache and swaptions are still left with considerable overhead. The factor which limits the performance of zSC as compared to RMO and TSO is coherence-permission-stall. For benchmarks under consideration this overhead tends to be negligible except for apache and swaptions where this overhead is considerable, about 6.59% and 10.82% of total execution time respectively.

7. RELATED WORK

For the set of benchmarks under study, even though a small percentage of total store misses get serviced by main memory, they are primary contributors to memory ordering overhead. Our proposed technique helps eliminate precisely this overhead by granting coherence permission without data for requests serviced by main memory. To our knowledge, earlier designs for efficient fences did not employ this optimization to amortize the overhead of enforcing fence constraints.

7.1 Store Miss Optimizations

A store miss waits for all sharers to acknowledge invalidation request. Store wait time can be reduced by early acknowledgment [3, 16, 27, 1] of an invalidation request as soon as it is queued at the recipient processor. The AlphaServer GS320 [19] exploits this optimization to perform stores faster. Early acknowledgment only targets store misses already cached in other processors, but not long latency off-chip store misses. We observe that store misses to blocks cached in other cores contribute far less to fence overhead as compared to off-chip store misses.

To address the overhead of off-chip store misses in a TSO processor, Chou et al. proposed a victim-cache-like structure (named SMAC) for the last-level L2 [11]. SMAC stores the addresses of evicted L2 blocks that had exclusive permission. If a store miss hits in SMAC, and if it is at the head of the FIFO store buffer, it is allowed to retire. SMAC optimization is complementary to ours in that it can help optimize store miss latency to cache blocks mapped to remote processors in a multi-socket system. While zFENCE can reduce store miss latency to any block managed by the directory, SMAC optimizes only for those that were recently evicted.

7.2 Efficient fences

There have been several proposals directed at making fences efficient. Techniques like Invisifence [7, 37] use out-of-window speculation to bring down the overhead of fences. Note that these proposals not only rely on checkpointing of large amount of processor state but also demand non-trivial amount of storage. Our proposed technique does not rely on any form of speculation and is thus free of any concomitant overheads of checkpointing or rollback.

There have also been non-speculative proposals to bring down fence overhead. Lin et al. [28] proposed conditional fences in which compiler communicates to the hardware about set of fences (called associate fences) for which fence ordering must be ensured. In other words, they ensure that all memory operations prior to a fence must be completed before its associate fence is issued. To keep track of associate fences, they propose to use a global table that maintains information about currently active fences in the system that have outstanding store misses. In contrast, our proposal does not need hardware support for tracking pending stores globally.

More recently, WeeFence [13] also uses global information about pending conflicting stores to allow memory accesses to skip fences. In contrast, our technique does not require keeping track of pending accesses at other cores and hence does not incur the additional complexity and storage overhead associated with it. SFence [29] aims to reduce fence overhead by letting programmers convey which memory operations need ordering guarantees provided by fences. and as such is orthogonal to our technique. SFence could use our optimization to further bring down the fence overhead and
also we could use SFence to limit the stores whose coherence permission need to be waited on before committing a fence.

7.3 Efficient SC Designs

In this section we describe prior work on efficient SC hardware. Techniques proposed in these prior work can be potentially used to reduce the cost of fences.

To enable store buffer optimization while preserving SC, past work proposed aggressive out-of-window speculation [34, 21, 20, 23, 10]. These designs allowed loads to speculatively commit even when the store buffer is not empty. To detect a memory ordering violation, incoming coherence invalidations need to be compared with addresses of all the loads committed after the oldest pending store. When a violation is detected for a speculatively committed load, execution needs to be rolled back to that load, which requires that the hardware checkpoints the program state at each speculatively committed store. Thus, out-of-window speculation requires fairly complex hardware to detect and recover from memory ordering violations.

Non-speculative SC hardware designs like Afek et al.’s “lazy caching” [3] and Landin et al.’s “race-free networks” [27] achieve SC but rely on special properties such as bus based ordering or race-freedom in the network. More recently, Lin et al. [30] proposed “conflict-ordering.” Their key insight is that a processor is allowed to perform a memory access as long as there are no pending conflicting accesses preceding it in the global total order. Thus, correctness in [30] was contingent on efficient conflict detection mechanism: a core has to be apprised of all pending conflicting memory operations at any other core before a memory operation could be committed from ROB. This constraint requires an augmented write buffer to track every memory operation until all preceding memory operations in global total order are complete. It further requires elaborate changes to coherence protocol for cores to communicate presence of pending memory operations to other cores.

Singh et al. proposed the End-to-End SC that supports SC efficiently by not enforcing SC constraints on provably safe accesses. We have discussed this design in Section 6.4. Recently, Gope et al. proposed Atomic-SC [22] a non speculative SC design for in-order multi-core systems. They implement another layer of functionality which they term as mutexes. Cores acquire mutex for every miss and for all hits under the shadow of a miss. Cores release acquired mutex for an operation only when all preceding operations in program order are complete. Thus, using mutexes they attain the net effect of obtaining a lock on a memory location and all subsequent operations before moving past a memory operation and keeping it pending. While implementing this technique for out-of-order multi-core systems is non-trivial, our proposed technique avoids the complexities of an additional mutex layer by harnessing the existing coherence infrastructure.

8. CONCLUSION

We observed that a significant fraction of fence overhead could be amortized simply by introducing a capability where the directory is able to grant the coherence permission before servicing the data. The only micro-architectural state that is added to a baseline hardware to take advantage of these early coherence permissions without data is a bit per store queue entry and store buffer entry. These early coherence permissions enable an effective fence design we term SFence that does not wait for long latency stores to be serviced by the main memory.

Our SC realization using SFence, zSC successfully eliminates most of the runtime SC overhead in single-socket systems by reducing it to 0.95% performance cost when compared to RMO hardware. Even for multi-socket systems, zSC is shown to be effective for a majority of programs. We also demonstrated that zSC outperforms a recently proposed SC hardware design.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments which helped improve this paper. This work is supported by the National Science Foundation under the CAREER-1149773 award.

10. REFERENCES


