Address-aware Fences

Changhui Lin
CSE Department
University of California,
Riverside CA 92521
linc@cs.ucr.edu

Vijay Nagarajan
School of Informatics
University of Edinburgh
United Kingdom
vijay.nagarajan@ed.ac.uk

Rajiv Gupta
CSE Department
University of California,
Riverside CA 92521
gupta@cs.ucr.edu

ABSTRACT

Many modern multicore architectures support shared memory for ease of programming and relaxed memory models to deliver high performance. With relaxed memory models, memory accesses can be reordered dynamically and seen by other processors. Therefore, fence instructions are provided to enforce the memory orderings that are critical to the correctness of a program. However, fence instructions are costly as they cause the processor to stall. Prior works have observed that most of the executions of fence instructions are unnecessary. In this paper we propose *address-aware fence*, a hardware solution for reducing the overhead of fence instructions without resorting to speculation. Address-aware fence only enforces memory orderings that are necessary to maintain the effect that the traditional fence strives to enforce. This is achieved by dynamically checking a condition for when an execution of a fence must take effect and delay the memory accesses following the fence. When a fence instruction is encountered, first, necessary memory addresses are collected to form a watchlist, and then, only the memory accesses to addresses that are contained in the watchlist are delayed. The memory accesses whose addresses are not contained in the watchlist are allowed to complete without waiting for the completion of pending memory accesses from before the fence. Our experiments conducted on a group of concurrent lock-free algorithms and SPLASH-2 benchmarks show that address-aware fence eliminates nearly all the overhead due to fences and achieves an average improvement of 12.2% on programs with traditional fences.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*

General Terms

Design, Performance, Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'13, June 10–14, 2013, Eugene, Oregon, USA.
Copyright 2013 ACM 978-1-4503-2130-3/13/06 ...$15.00.

Keywords

Memory models, Fence instructions, Microarchitecture

1. INTRODUCTION

Multiprocessors are becoming ubiquitous in all computing domains, from mobile devices to datacenter servers. They are able to deliver high performance via parallelism, and hence the importance of parallel programming continues to grow. To simplify programming for multiprocessors, shared memory is widely used as the primary system level programming abstraction. However, problems arise when accesses to shared memory are reordered. Hence, many memory consistency models have been proposed to constrain memory behaviors with respect to read and write operations originating from multiple processors [1]. Each of these models offers a trade-off between programming simplicity and high performance. To achieve high performance, many manufacturers (e.g., Intel, IBM, Sun, etc) typically choose to support relaxed/weak consistency models, such as total store order (TSO), relaxed memory order (RMO), release consistency (RC), etc [1]. With relaxed memory consistency models, memory accesses can be reordered dynamically and seen by other processors. To enforce the memory orderings that are critical to the correctness of programs, fence instructions are provided. A fence instruction guarantees that all memory accesses prior to it are completed before the memory accesses following it are performed. The stalling of the processor resulting from implementing this fence semantics is costly and often unnecessary [17, 20, 36].

The traditional fence instructions are processor-centric [36], i.e., a fence instruction only controls the ordering of memory accesses for the processor executing the fence while being unaware of memory accesses in other processors. However, when the reordering of memory accesses across a fence is not observed by other processors, it is safe to allow the reordering. Researchers have observed that most dynamic fence instances are unnecessary because the execution of the program already conforms to the effect that fences strive to enforce and therefore they have developed techniques for eliminating the performance degradation due to unnecessary execution instances of fence instructions. These techniques include both speculative techniques [3, 5, 14, 15, 37] and non-speculative techniques [17, 20, 36]. While the latter are more desirable because of their lower hardware complexity, they have two limitations: (a) they are only able to eliminate a subset of unnecessary fence instances; and (b) when a fence instance is necessary, no memory accesses are allowed

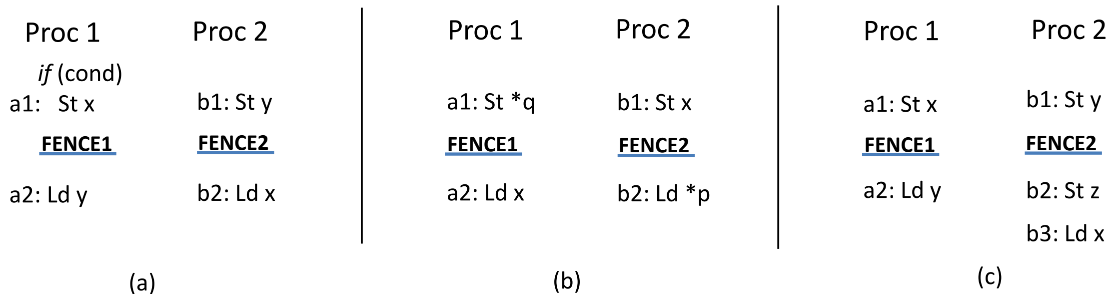

Figure 1: Unnecessary fence instances at runtime.

to be reordered across fence boundaries even though it may be safe to allow reordering of some of the memory accesses.

Fig. 1 shows several common scenarios which demonstrate the above limitations of existing techniques [17, 20, 36]. The example in Fig. 1(a) contains a conditional branch. Fence instructions are inserted to ensure that memory accesses to shared variables x and y are ordered properly. However, if at runtime the condition in Proc 1 evaluates to `false`, and hence $a1$ is not executed, then the shared variable x is not accessed concurrently by multiple processors. As a result, the fence instance is not necessary to effect the order of memory accesses of x and y. In Fig. 1(b), there are two pointers p and q pointing to some field of a shared object (e.g., a hash table [29]) respectively – since they may point to the same field, fence instructions are inserted. However, at runtime, if they point to different fields of the shared object, i.e., $a1$ and $b2$ do not access the same field concurrently, then the fence instances are not necessary. Finally, in Fig. 1(c), `FENCE2` orders both $b1 \rightarrow b2$ and $b1 \rightarrow b3$. However, if z is not accessed in any other processor concurrently, $b2$ need not be delayed and hence can be reordered across the fence. Thus, even though the instance of `FENCE2` is necessary, $b2$ need not be delayed. However, none of the existing techniques [17, 20, 36] can eliminate the above unnecessary fence instances.

We observe that with the knowledge of what is happening in other processors, we can eliminate the above restrictions and optimize the fence executions. In this paper, we propose a hardware solution *address-aware fence* to reduce the overhead due to fence instructions without resorting to speculation. Unlike a traditional fence which is processor-centric, an address-aware fence collects necessary information to decide whether it should effect the stalling of following memory accesses. In particular, an address-aware fence instance has an associated *watchlist*, which contains memory addresses that should not be accessed by the memory accesses following the fence. The completion of a memory access following the fence is allowed if its memory address is not contained in the watchlist, appearing as if the fence does not take effect. Otherwise, the memory access is delayed to ensure correctness. By doing so, unnecessary fence instances behave as a *nop* (i.e., they do not stall the processors) and the other fence instances selectively stall memory operations (i.e., stalling is minimized). Therefore this approach effectively reduces overhead of fences whenever possible.

The key contributions of this work are:

1. We propose a new fence mechanism, called *address-aware fence*. This is a hardware mechanism that eliminates stalling at unnecessary fence executions and reduces performance overhead due to fence instructions without resorting to speculation. The technique is implemented in the microarchitecture without instruction set support and is transparent to programmers.

2. We describe the architectural design of address-aware fence. The hardware support only amounts to less than 1KB added to a conventional multiprocessor core.

3. We conduct experiments with a group of concurrent lock-free algorithms and SPLASH-2 benchmarks. The results show that address-aware fence eliminates nearly all the overhead associated with the traditional fence and yields an average reduction of 12.2% in execution time.

The rest of the paper is organized as follows. Section 2 presents the background on fence instructions. Section 3 proposes address-aware fence. Section 4 describes the hardware design. The experimental results are presented in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

In this section, we begin with the introduction to fence instructions provided by modern architectures and their applications, and then briefly present some existing approaches to reducing fence overhead.

2.1 Fence Instructions

Fence instructions, or *fences* for short, are provided in modern processors as a mechanism to selectively constrain the default relaxed memory access orders [1, 11]. Fence instructions of commercial architectures have various names and semantics, enforcing different memory orderings. For example, in Intel IA-32, there are three types of fence instructions: *mfence*, *lfence*, and *sfence*. While *mfence* enforces all memory orderings, *lfence* only enforces orders between memory load instructions, and *sfence* only enforces orders between memory store instructions. Similarly, in SPARC V9, MEMBAR instruction can be customized to enforce different memory orderings; in PowerPC, there are *sync* and *lwsync*; and in Alpha model, there are memory barrier (MB) and write memory barrier (WMB). In the following discussion, we assume a traditional fence is a full fence, which guarantees that all memory operations prior to it have completed before its following memory operations can be performed.

(Applications of fences) For programs running on architectures that only support relaxed consistency models, it is important to enforce memory orderings that are critical

to the correctness of the programs. This is because, under relaxed consistency models, the orders specified by the program are not guaranteed during execution. The correctness of the program may rely on some orders, which can be potentially reordered by the processor or memory systems without any protection. Hence, fence instructions have been used frequently to enforce such necessary memory orderings. In fact, many lock-free algorithms have to use fence instructions to ensure that the algorithms perform correctly under relaxed consistency models [4] (e.g., Dekker algorithm [9], lock-free dynamic memory allocation [26]). Fences are also used to implement synchronization operations [33]. In some memory models (e.g., TSO), fences are implied by atomic read-modify-write (RMW) instructions for acquire operations. In addition, sufficient fences can also guarantee sequential consistency (SC) for programs running in architectures only supporting relaxed consistency models [11, 19, 30, 22].

2.2 Reducing Fence Overhead

Although fence operations are cheaper than locks, excessive use of fences can still hurt performance. In commercial applications, frequent thread synchronizations result in significant ordering delays due to fence instructions [37]. However, it is possible to reduce the overhead, based on the observation that memory accesses across a fence can be reordered as long as they are not observed by other processors. Therefore, some techniques have been proposed to reduce the overhead of fence instructions.

Techniques	Applicability			
	Dekker-like	Lock-free algo.	Lock	Improving SC
CMO [36]	No	No	Yes	No
l-mfence [17]	Yes	No	No	No
C-Fence [20]	Yes*	Yes*	Yes*	Yes*
Address-aware fence	Yes	Yes	Yes	Yes*

Table 1: Comparison of existing approaches and our approach (*Compiler support required for inserting fences).

(Speculative Techniques) First, there is a group of techniques that can address the problem by speculation [3, 5, 14, 15, 37]. Memory accesses after a fence can be speculatively retired when there is any pending store before the fence. Before these speculatively retired memory accesses are committed, if a memory access in another processor conflicts with one of them, rollback is performed and instructions are then re-executed. However, the hardware complexity associated with aggressive speculation can hinder wide-spread adoption of these speculative techniques.

(Non-speculative Techniques) There are also three non-speculative techniques that are listed in Table 1. The table compares the applicability of these techniques in terms of the kind of applications they can handle. In [36], Praun *et al.* observed that memory ordering instructions used on acquire and release of a lock are often unnecessary. They proposed a combined HW/SW mechanism *conditional memory ordering* (CMO) to omit unnecessary memory ordering instructions. CMO only focuses on memory orderings associated with lock acquire and lock release, but cannot handle the other cases. Ladan-Mozes *et al.* [17] proposed *location-based memory fences* to reduce fence overhead. A new instruction (*l-mfence*) is introduced, but it is limited to Dekker-like algorithms. In our prior work [20], we proposed *conditional fence* (C-Fence) to enforce sequential consistency with the help of compiler inserting necessary fences. The compiler is also responsible for collecting the information of fence associates, which are conveyed to hardware to improve the performance of sequential consistency by only making associates be staggered enough. In addition, C-Fence is also able to handle other kinds of applications as long as fence associates can be provided by compiler. However, it is still not able to optimize the scenarios shown in Fig. 1.

Our non-speculative solution, address-aware fence*, presented in this paper is superior to the above non-speculative techniques in two respects. It is more effective as it is able to exploit all the optimization opportunities shown in Fig. 1. It is also broadly applicable as it can be applied in all situations in Table 1.*

3. ADDRESS-AWARE FENCE

In this section we propose the *address-aware fence* mechanism for reducing fence overhead. Our technique is implemented in the microarchitecture without introduction of any new instructions and is thus completely transparent to the programmer. Fence instructions are introduced, as usual, either by the compiler or by the programmer. Fence instructions in the executable are identified at runtime and processed by the hardware. Our technique handles all encountered fences without being aware of how they are used. Thus, our technique can naturally handles all cases listed in Table 1.

A fence instruction forces a strict ordering between memory accesses that precede it and those that follow it. A traditional fence enforces this by stalling the processor's execution till all memory accesses encountered before the fence have completed. However, the above approach is more restrictive than what is really required. The purpose of a fence is to prevent memory accesses from being reordered and *observed by other processors*. In other words, if the reordering is not observed by other processors, the reordering is allowed and hence stalling at the fence is not necessary. Let us call the order enforced by a fence as *fence order*. We can relax the execution of a fence such that the execution is correct – a fence execution is said to be *correct if the execution appears to maintain fence orders*. Obviously, the traditional fence execution is correct. However, it is inefficient due to unnecessary stalls.

The *address-aware fence* mechanism exploits the above observation to improve the efficiency of the fence execution mechanism. This section first presents a *sufficient condition* for a correct fence execution and then our solution *address-aware fence* that utilizes the condition.

3.1 The Condition for Enforcing Fence Orders

Fig. 2(a) shows two fences in two processors respectively. $A1, A2, B1$, and $B2$ represent blocks of instructions separated by fences. Furthermore, let us assume $a1$, $a2$, $b1$, and $b2$ are memory accesses in $A1$, $A2$, $B1$, and $B2$ respectively. Due to the fence, any memory access in $A1$ is ordered before any memory access in $A2$, represented by the solid line with arrow; and the same is the case for $B1$ and $B2$. Thus, the fence orders $a1 \rightarrow a2$ and $b1 \rightarrow b2$ are enforced by the fences. A correct fence execution should make these fence

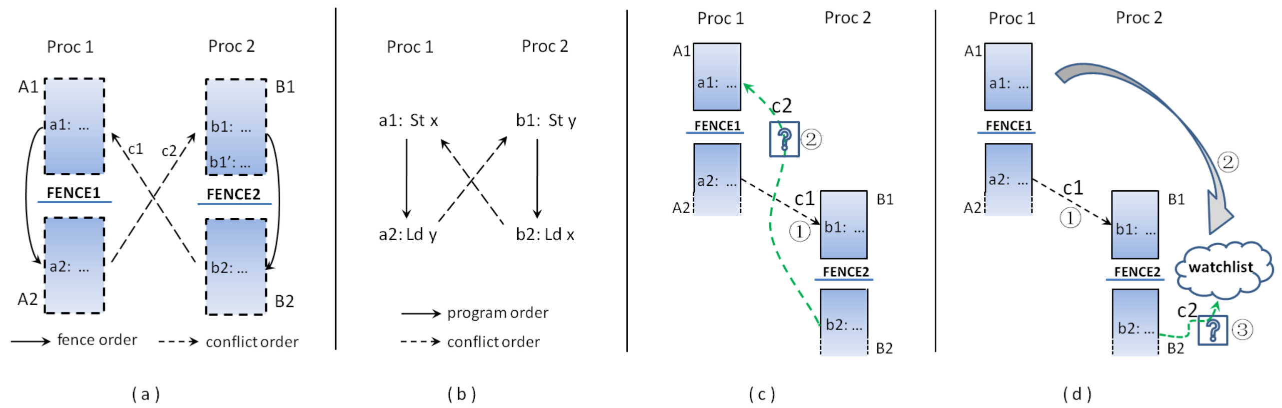

Figure 2: (a) Violation of fence order; (b) Violation of program order; (c) Cycle detection; (d) Our approach.

orders *appear* to be enforced. In [30], Shasha and Snir have shown the condition for enforcing program orders in context of sequential consistency (SC) enforcement. Extending that condition, we can have the condition for enforcing fence orders, as all fence orders is a subset of all program orders.

Recall that sequential consistency requires that all memory accesses *appear* to take place in the program order which is specified by the program, i.e., all program orders should be enforced. Shasha and Snir [30] have shown that an execution does not violate sequential consistency iff *program orders* ($\mathbf{P}$) and *conflict orders* ($\mathbf{E}$) do not form a cycle (i.e., no cycle in $\mathbf{P} \cup \mathbf{E}$). Here, a *conflict order* is an execution order of conflicting memory accesses [30]. Fig. 2(b) shows such a cycle $a1 \rightarrow a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, where $a1$ conflicts with $b2$ and $a2$ conflicts with $b1$. The execution sequence $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$ will violate sequential consistency, because no sequentially consistent execution can generate the same result, where both x and y read their old values before stores. In the case of fence orders in Fig. 2(a), we have to enforce all fence orders ($\mathbf{F}$), analogous to program orders for SC in Fig. 2(b). In particular, all fence orders is a subset of all program orders ($\mathbf{F} \subseteq \mathbf{P}$). Accordingly, we can have the condition for enforcing fence orders – *fence orders are enforced iff fence orders and conflict orders do not form a cycle* (i.e., no cycle in $\mathbf{F} \cup \mathbf{E}$). This is because, without cycles, $\mathbf{F} \cup \mathbf{E}$ can be extended to a total order [7], which indicates all fence orders $\mathbf{F}$ are enforced. Therefore, in Fig. 2(a), to enforce fence orders, we have to prevent such cycles as $a1 \rightarrow a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, assuming $(a1, b2)$, $(b1, a2)$ are conflict relations. Note that, even if there is another memory access $b1'$ after $b1$ in $B1$, and $(a1, b1')$ and $(a2, b1)$ are conflict relations, there is no violation of fence orders involving $a1, a2, b1$, and $b1'$. This is because there is no fence order between $b1$ and $b1'$ and hence there is no cycle formed by fence orders and conflict orders.

3.2 Our Approach

Address-aware fence only orders memory operations involving certain memory addresses that must be ordered to maintain fence orders. In other words, even if there are stores from before the fence that are pending, the following memory operations can still complete if their addresses are not forbidden by the fence. This is why we name the fence as address-aware fence. An address-aware fence can decide whether it must stall memory operations according to their memory addresses. We make use of the condition described earlier to detect at runtime whether it is possible to form a cycle with fence orders and conflict orders. If no cycles can be formed, fences do not stall the memory operations following them; otherwise, fences will function so as to maintain fence orders.

In this section, we present the high-level algorithm of our approach; while in the next section we will describe the detailed hardware design. The key problem is how to detect possible cycles at runtime. During execution, fence orders are easily known, and hence we have to detect the conflict orders that can form a cycle along with the fence orders. Fig. 2(c) shows how cycles can be detected. In Proc 1, suppose all instructions in $A1$ have completed except some pending memory operations, and `FENCE1` does not stall the following instructions in $A2$ initially. Hence, some memory operations are reordered across `FENCE1` at runtime. At this point, in Proc 2, there is a memory operation in $B1$ which detects a conflicting memory operation in $A2$ (in the next section we will show how the detection is performed relying on cache coherence transactions). It forms a conflict order $c1$ from $A2$ to $B1$ as shown in the figure (①). This event triggers the following event: every memory operation after `FENCE2` has to detect whether there is any pending memory operation prior to `FENCE1` that conflicts with it, forming the conflict order $c2$ (②). Memory operations in $B2$ that do not conflict with pending memory operations in $A1$ can complete without being stalled by `FENCE2` even when there is any pending memory operation in $B1$. However, if there does exist a potential conflict order $c2$ from $B2$ to $A1$, `FENCE2` will delay the involved memory operation in $B2$ to break this potential conflict order. Moreover, if $c1$ does not exist, the detection of $c2$ is unnecessary, as no cycle will be formed without $c1$. Detecting $c2$ is only triggered when there is a possible $c1$.

The cycle detection discussed above does not consider how to relate $c1$ and $c2$. In particular, we should know where to check conflict, e.g., memory operations in $B2$ should check conflict with pending memory operations in $A1$. Furthermore, detecting $c2$ requires every memory operation in $B2$ to check conflict in $A1$ which is on a different processor and thus this is inefficient. Fig. 2(d) shows our approach to address this problem. If a conflict order $c1$ is detected (①), memory addresses of all pending memory operations before `FENCE1` are collected to form a *watchlist*, which is associated with `FENCE2` (②). Now, the memory operations after `FENCE2` only need to check the local watchlist to detect conflict, i.e., to detect $c2$ (③). If the address of a memory operation is

not contained in the watchlist, it indicates there is no conflict to form $c2$, which further indicates there is no cycle detected to violate fence orders. Hence, those memory operations whose addresses are not contained in the watchlist can still complete without being stalled by the fence even when there are pending memory operations from before the fence.

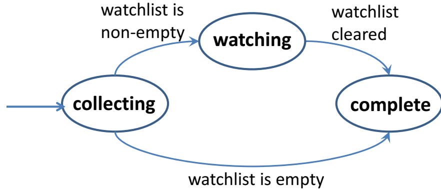

Figure 3: States of address-aware fence.

(Life cycle of a fence instance) The execution of a fence instruction proceeds according to the state transition graph in Fig. 3. There are three states: *collecting* state, *watching* state, and *complete* state. When the fence is issued it starts out being in *collecting* state where it waits for its watchlist. The memory operations following the fence cannot complete, as at this time the watchlist that is needed to detect cycles is not available. After the fence has collected all necessary addresses to form its watchlist, it switches to next state: the fence switches to *watching* state if the watchlist is non-empty; otherwise it switches directly to *complete* state. In *watching* state the memory operations following the fence must check the watchlist before completion. The watchlist is cleared when the corresponding pending memory operations have completed and the fence switches to *complete* state, where the fence has completed execution. If a fence directly switches from *collecting* to *complete* state, it causes no stalls; thus, it can be viewed as being dynamically eliminated.

4. HARDWARE DESIGN

In this section, we describe our hardware design for address-aware fence. The challenge is to detect and avoid cycles efficiently. In the following discussion, we present the detailed hardware design in context of a scalable CMP with distributed directory-based invalidation cache coherence protocol. Each processor has a local L1 cache, a bank of L2 cache, and a portion of directory. Each coherence transaction is kept in the directory until it receives the notification of the transaction completion. We assume each processor core to be a dynamically scheduled ILP processor with a reorder buffer (ROB). All instructions retire from ROB in program order. At the head of ROB, loads can retire when they complete, while stores can retire as soon as the value and destination address are available through store buffering technique [12], which allows stores to retire from the head of ROB even before they complete. We will see later that we use an augmented buffer which incorporates the function of store buffer, in which stores are allowed to complete out of order. The store buffering relaxes Store-Load and Store-Store orders, but the traditional fence requires the store buffer to be drained before executing following memory accesses.

Moreover, the processors also support in-window speculation [13], which guarantees Load-Load order by speculative load execution (a speculative load in ROB is squashed if its loaded data is invalidated or replaced before it retires). Besides, Load-Store order is naturally guaranteed by ROB. Therefore, memory operations cannot complete past a pending load, and hence the pending memory operations that can be bypassed can only be *pending stores*.

4.1 Operations on Address-aware Fence

Each processor functions as usual when there is no fence executing. However, when a fence is issued, the processor initiates the process of handling the fence using address-aware fence mechanism. The key operations are *collecting* and *clearing* watchlist for a fence, which are introduced in this section. After the watchlist has been collected, the fence can retire when it reaches the ROB head. If any load/store following the fence tries to *retire* while there are still pending stores from before the fence, the processor will check whether the address of the load/store is contained in the watchlist.

4.1.1 Collecting watchlist

When a fence is issued, the processor starts to collect watchlist for the fence, which is now in *collecting* state as shown in Fig. 3. As described in Section 3.2, a watchlist consists of the memory addresses of a set of pending memory operations that are all pending stores. It is important to obtain the set of pending stores quickly, as memory operations following the fence cannot retire until the watchlist is obtained. To do this, we utilize the directory, where we are able to find all pending stores being serviced by the directory. In the following discussion for simplicity we assume a centralized directory; however, later we describe the modifications needed for a distributed directory. When a fence is issued, the processor sends a request to the directory to fetch the addresses (block addresses) of all pending stores in other processors. The directory compresses the addresses into a watchlist using a bloom filter and sends it back to the requesting processor. Then the requesting processor records the replied watchlist locally for checking conflict. In this way, we conservatively assume there is a conflict order $c1$ (Fig. 2(d)①), and obtain the watchlist by fetching all pending stores (Fig. 2(d)②) from the directory.

The processor starts to collect the watchlist as soon as a fence is issued, because we would like to obtain the watchlist as early as possible, so that the fence does not stall the pipeline. However, the collected watchlist may become stale before it is used for checking conflict. Let us take Fig. 4(a) as an example for the following discussion. Suppose $a1$ is a store miss, and $a2$ completes past $a1$; then $b1$ completes after $a2$. If we do not take care of $b2$, the execution order $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$ will form a cycle and violate the fence order. In Fig. 4(b), let us assume `FENCE2` is issued at time $t1$, when no pending store is present in the directory (i.e., cache miss $a1$ has not reached the directory). So the watchlist obtained for `FENCE2` will be empty. If we use this empty watchlist, we cannot avoid the execution order $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, as the empty watchlist will allow $b2$ to complete before $a1$. This is because the watchlist becomes stale and it does not contain x accessed by the pending store $a1$.

To address this problem, we observe that *the watchlist only needs to be updated when there is a cache miss before the fence*. To see why, let us recall cycle detection in Fig. 2(c). A cache miss in $B1$ would indicate that there is a potential conflict order like $c1$, so we need to avoid conflict order $c2$. The stale watchlist may not contain all pending stores in $A1$, as new pending stores may be generated after

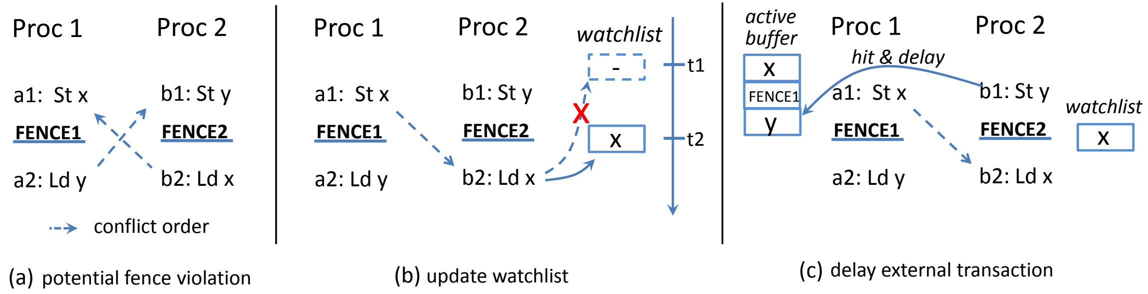

Figure 4: Collecting and clearing watchlists.

the stale watchlist was obtained. Hence, we have to update the watchlist. On the other hand, a cache hit does not create a conflict order like $c1$, so there is no need to update the watchlist. Therefore, if there is any memory operation before the fence that is a cache miss (load/store miss) in the local cache, the cache miss transaction sent to the directory will also require the directory to reply with an updated watchlist. Note that, the fence can only retire after it has received the above reply from the directory. For the case in Fig. 4(b), if $a2$ completes past $a1$, $a1$ will be present in the directory if it has not completed. Thus, at time $t2$, $b1$ is found to be a miss, so the transaction sent to the directory will obtain an updated watchlist which includes x. Now the updated watchlist containing x will stall $b2$, enforcing the order $a1 \rightarrow b2$ and avoiding fence order violation.

(Unnecessary update) Although a cache miss may create the conflict order $c1$ as shown in Fig. 2(c), $c1$ does not really exist if the directory indicates the target block is not cached in other processors. In this case, the watchlist does not need to be updated. In particular, cache misses to local variables will fall into this case, which reduces much unnecessary traffic.

4.1.2 Clearing watchlist

A fence will be in *watching* state when it has retired, and memory operations after the fence have to check the watchlist to bypass it. But the fence cannot simply complete even when all pending stores before it have completed, because memory operations after the fence may still have to check the watchlist to avoid cycles. Let us consider the example in Fig. 4(b) again, where $a2$ completes past $a1$ and `FENCE2` has obtained the watchlist containing x. But this watchlist cannot be cleared even when $b1$ has completed, as $b2$ still has to check the watchlist to avoid the cycle by ordering $a1 \rightarrow b2$. The watchlist can be cleared only when $a1$ has completed. That is, the watchlist can be cleared and hence the fence can complete *only when the pending stores whose addresses are contained in the watchlist have all completed*. If we make each complete store notify the processors which may contain its address, there will be complicated communication of tracking information between processors, complicating the hardware design.

To address this problem, we will delay $b1$ until $a1$ completes. Hence, we introduce a buffer for each processor, called *active buffer*. We consider a memory operation or a fence as *active* if (1) it has retired; and (2) the memory operation is a pending store or there is a pending store before the memory operation/fence. In other words, after retirement, a memory operation or a fence becomes inactive when there is no pending store before it. The *active buffer* records the addresses of all active memory operations, as well as active fences, in the order they are retired. Now, each external coherence transaction will also check the active buffer. If there is a conflict in the active buffer (i.e., the target memory address is found in the active buffer) and there is a fence prior to the conflicting entry, then this coherence transaction is delayed until the target address has been removed from the buffer (i.e., the corresponding memory operation is no longer active). Let us recall Fig. 2(d). By using active buffer, if there is a conflict order $c1$ from $A2$ to $B1$, the completion of $B1$ will indicate the completion of $A1$; otherwise, $B1$ cannot complete as coherence transactions sent to `Proc 1` will be delayed. Thus, the watchlist can be safely cleared as soon as $B1$ has completed, because it indicates all pending stores in $A1$ have completed. This simplifies the implementation for clearing watchlist, as the processor can decide when to clear watchlist based on the local information. Note that, to guarantee that any potential conflict is detected, the cache block whose address is in the active buffer is not allowed to be evicted from local cache.

We illustrate the algorithm in Fig. 4(c). Suppose $a1$ is a cache miss and $a2$ has completed past $a1$ (so $a1$ will be present in the directory if it has not completed). Now both $a1$ and $a2$ are active, so their addresses are recorded in the active buffer; and `FENCE1` is recorded as well. In `Proc 2`, $b1$ is then executed, which will be a miss. It obtains the watchlist containing x from the directory and also sends a coherence transaction for y to `Proc 1`. Since y is in the active buffer and there is a fence prior to y, the transaction is delayed, and hence the watchlist is not cleared. If $b2$ tries to retire, it has to check the watchlist and hence it stalls. When $a1$ has completed, the active buffer will be empty, and the delayed coherence transaction for y from `Proc 2` can be satisfied. Thus, the completion of $b1$ indicates memory operations prior to `FENCE1` have completed; otherwise, $a2$ is active and hence $b1$ cannot complete. Now the watchlist containing x can be safely cleared as soon as memory operations prior to `FENCE2` have completed, without the risk of forming cycles.

(Deadlock freedom) In the above example, it seems possible that all four instructions $a1, a2, b1$, and $b2$ are active, and $b1$ is delayed by $a2$ and $a1$ is delayed by $b2$, which forms a deadlock. However, we show that it is not possible that all four instructions are active at the same time. Since $a1$ and $b1$ are misses (otherwise $a2 \rightarrow b1$ and $b2 \rightarrow a1$ do not exist), they are sent to the directory. Suppose $a1$ first reaches the directory. So the watchlist for `FENCE2` will contain x accessed by $a1$. This watchlist stalls $b2$, which will not be able to retire and become active. Thus, the above scenario of deadlock is not possible.

4.1.3 Distributed directory

To make address-aware fence scalable, we now consider the implementation with a distributed directory. With a centralized directory, a cache miss transaction, that needs to update the watchlist, is sent to the directory and obtains all pending stores in other processors. However, when the directory is distributed, a cache miss transaction is only sent to its *home directory*, and only obtains the pending stores in that directory. Hence, the memory operations after the fence can use the watchlist for checking only if they are mapped to the same home directory where the watchlist is collected.

To accommodate distributed directory, we maintain a buffer called *watchlist buffer*, which has several entries recording watchlists collected from different home directories. Each entry in the buffer is also tagged with the ID of the tile where the home directory resides. When a cache miss brings back a watchlist, it is recorded in an entry of the buffer, tagged with the corresponding tile ID. Meanwhile, other entries are invalidated as they might contain stale watchlists. A memory operation trying to retire past the fence first checks whether there is a valid entry with the tile ID to which the memory access is mapped. If yes, the memory operation checks against the watchlist. Otherwise, it is forced to fetch the watchlist from its own home directory before it can check conflict for retirement. The fetched watchlist is also recorded, so that future memory operations mapped to the same home directory can check conflict locally. Thanks to spatial locality, most of the nearby memory accesses tend to be mapped to the same home directory, which allows most memory operations to check watchlists quickly and retire past fences. We use the distributed directory in our evaluation.

(Handling multiple fences) In the above discussion, we only consider one executing fence. However, it is also possible that multiple fences are executing in the pipeline. Since a watchlist has to be removed when the corresponding fence can complete, we have to know which watchlist is associated with which fence. To do this, each issued fence in the pipeline is associated with a unique tag, and the watchlist collected for the fence will also be associated with the same tag. When a fence completes, only the watchlists with the same tag as the fence are removed. Hence, the watchlists for uncompleted fences continue to remain recorded in the buffer. In particular, the watchlist for a fence can be replaced by the watchlist for a second fence if they are collected from the same home directory, because the latter is the updated watchlist from that directory.

4.2 Hardware Summary

Fig. 5 shows the overview of hardware support for address-aware fence. For simplicity, only main logic blocks related to our technique are presented. Each processor core is augmented with two new logic blocks to implement our technique: *active buffer* and *watchlist buffer* as discussed in the previous section. If a retired memory access or fence is active, it is added to the active buffer (①). Each external cache transaction has to check the active buffer, and is delayed if there is a hit (②). Each watchlist is recorded in the watchlist buffer when it is received from the directory (③). Each memory access has to check the watchlist buffer to see if it can retire (④). Although we leverage cache coherence to transfer metadata, the coherence protocol and its transactions are left unchanged. Next, we summarize the operations in the active buffer and watchlist buffer.

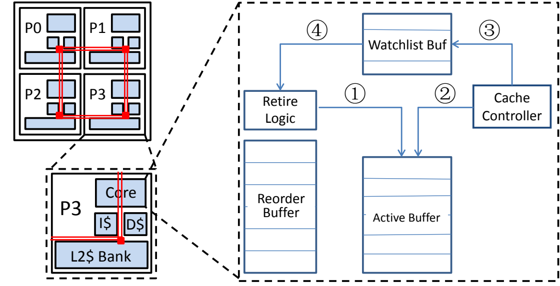

Figure 5: Overview of the architecture.

(Active buffer) The active buffer records the addresses of all active memory accesses and active fences. It also incorporates the function of store buffer. Each entry of the buffer consists of the following fields: *valid*, *type*, *done*, and *data*. Descriptions of fields are shown in Table 2. The *data* field depends on the type. For a store, it includes the destination memory address and a pointer to its data; for a load, it only includes the memory address; and for a fence, it includes a unique tag used to mark its watchlist.

Fields	Description
valid	whether it is a valid entry
type	load, store or fence
done	whether the operation has completed loads and fences are always complete
data	depending on the type

Table 2: Fields in active buffer.

The following are the operations for the active buffer. (1) *Add*. Active memory accesses and active fences are added to the active buffer in the order they retired. However, two consecutive loads or two consecutive stores can be merged if their destination addresses are mapped to the same cache block, reducing the size of active buffer. This is because the conflict detection is based on the block address. If the buffer is full, the retiring memory access or fence is delayed until there is an entry available. (2) *Delete*. If a memory access or fence is no longer active, it is removed. That is, at the head of active buffer, if a pending store has completed, entries until next pending store are invalidated. (3) *Conflict detection*. Each external coherence transaction has to detect conflict in the active buffer. If there is a conflict in the active buffer and there is an active fence prior to this conflicting entry, the transaction is delayed. It is important to make conflict detection efficient. We use a counting bloom filter to hash memory addresses in the active buffer. Conflict detection first checks the counting bloom filter before checking entries in the active buffer, which greatly reduces useless checks as conflicts are rare.

(Watchlist buffer) The watchlist buffer records all collected watchlists. Each watchlist is a bit vector which contains memory addresses compressed using bloom filter. This is to minimize the network bandwidth, because watchlists are exchanged along with cache transactions. Each entry consists of the following fields: *valid*, *fence tag*, *tile ID*, and *watchlist*. Descriptions of each field are shown in Table 3.

The following are the operations for the watchlist buffer. (1) *Add*. Each watchlist is recorded when it is replied from

Fields	Description
valid	whether it is a valid entry
fence tag	which fence is the watchlist for
tile ID	which directory the watchlist is collected
watchlist	bit vector of pending stores

Table 3: Fields in watchlist buffer.

the directory, with the corresponding fence tag and tile ID. (2) *Delete*. When all memory operations before a fence have completed, the entries with the same fence tag is invalidated. Moreover, when a cache miss before the fence brings back an updated watchlist, other entries are invalidated as they might contain stale watchlists. (3) *Retirement check*. Each retiring memory operation checks against the watchlist with the tile ID to which its memory address is mapped. If it does not find such watchlist, it is first forced to fetch the watchlist from its home directory. If the block address of the memory operation is not contained in the watchlist, it can retire even if there is a pending store prior to the fence. Otherwise, it indicates the retirement of the memory access may result in a violation of fence order, and hence it is delayed. It is worth noting that watchlist buffer will eventually become empty if the processor is stalled, which indicates that the forward progress is guaranteed.

5. EXPERIMENTAL EVALUATION

The goals of our evaluation are: (1) to understand why address-aware fence performs better; (2) to assess the performance of address-aware fence compared to traditional fence; and (3) to assess the space and traffic overhead.

(Simulation) We have developed a hardware simulation infrastructure using the Pin tool [24] that simulates a directory-based shared memory multiprocessor system. Each processor has a private 4-way 32KB L1 cache and all processors share a L2 cache (16-way 1MB/core). All L1 caches are kept coherent using a directory-based MESI protocol. All cores are connected via a mesh network, which has a link latency of 2 cycles and router latency of 3 cycles. Each instruction takes 1 cycle to execute, and it takes 2, 10, and 300 cycles to access the L1 cache, L2 cache, and main memory, respectively.

Benchmarks	Description
dekker	Dekker algorithm [9]
lamport	Lamport Queue [18]
msn	Non-blocking Queue [27]
wsq	Chase-Lev's Work Stealing Queue [6]
bst	Binary search tree
SPLASH-2	8 programs from SPLASH-2 [38]

Table 4: Benchmark description.

(Benchmarks) We evaluate our technique using benchmarks shown in Table 4. There are two groups, concurrent lock-free algorithms and SPLASH-2 benchmark programs. In the first group, concurrent lock-free algorithms are implemented using fences and atomic compare-and-swap (CAS) instructions. Dekker algorithm (*dekker*) [9] is a classic solution to mutual exclusion problems using only shared variables for communication. Lamport Queue (*lamport*) [18] is a single-producer and single-consumer queue. Non-blocking concurrent queue (*msn*) is a multiple-producer and multiple-consumer queue. Chase-Lev work-stealing queue (*wsq*) [6] is a lock-free work-stealing deque implemented with a growable cyclic array. *bst* is a concurrent search structure implemented using atomic CAS instructions. Since these lock-free data structures are not closed programs, we constructed harnesses to use them to assess the performance of address-aware fences. The second group of benchmarks are from SPLASH-2 [38]. In these benchmarks, fences are inserted to enforce sequential consistency. We identified the fence insertion points based on Shasha and Snir's delay set analysis, where we employed dynamic analysis to find conflicting accesses as in [10, 20]. We also use these benchmarks to compare address-aware fence and C-Fence [20].

5.1 Performance

In this section, we would like to understand how address-aware fences improve fence performance and evaluate the performance overhead induced by address-aware fences.

5.1.1 *Effectiveness of address-aware fence*

Benchmarks	#Fences	Address-aware fences		C-Fence
		#Check	#in W.L.	#Conf.
barnes	83M	47M	206	21M
fmm	5M	2M	223	1M
ocean	9M	15M	383	1M
radiosity	30M	5M	90	4M
raytrace	44M	34M	239	5M
volrend	39M	49M	436	3M
water-ns	9M	1M	79	656K
water-sp	7M	2M	88	395K
AVG.	28.8M	19.4M	218	4.9M

Table 5: Effectiveness of address-aware fence.

In this evaluation, we use SPLASH-2 benchmark programs where fences have been inserted for enforcing sequential consistency. All programs are run with 8 threads. Table 5 characterizes address-aware fences in terms of how often fences have to take effect. Column 2 in the table shows the number of dynamic fence instructions executed in each program. Although they only account for a small part of all instructions (∼1%), they induce relatively larger execution time overhead, as shown later in Fig. 6. With address-aware fence, when there is any active fence, memory accesses have to check with watchlists before retirement. Column 3 shows the number of such memory accesses. Column 4 shows the number of memory accesses whose block addresses are found to be present in the watchlists. We can see that, compared with the number in Column 3, very few memory accesses are stalled by fences. In fact, the number of fences that need to stall (Column 3) is negligible compared with the total number of fences (Column 2). For example, *volrend* has the largest number of detected conflicts, but this is still very small compared the total number of fences – 436 *vs.* 39 Million. On average, only 218 memory accesses are found in watchlists; thus, avoiding nearly all fences from taking effect.

(Comparison with C-Fence) We also implemented and studied C-Fence mechanism [20], which is able to dynamically eliminate a fence as long as none of its associate fence (provided by compiler) is executing concurrently in other processors. We measured the number of fences which detect executing associates and have to stall. Numbers in Column 5 show that the number of fences that have to stall when C-Fence is used. On an average, around 15% of C-Fences have to stall. In contrast, address-aware fences require very few

stalls (Column 4 *vs.* Column 5). This is because address-aware fence is able to exploit more optimization opportunities dynamically, e.g., scenarios in Fig. 1, which C-Fence is not able to optimize.

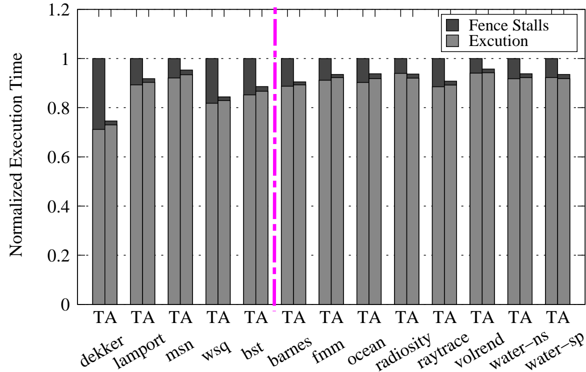

Figure 6: Execution time – T represents traditional fence and A represents address-aware fence.

5.1.2 *Execution time*

We measured the execution time of programs with traditional fences and address-aware fences, respectively. Fig. 6 shows the results, which are normalized to the execution time achieved using traditional fences. The execution time is broken down into two parts: the stall time due to fences (*Fence stalls*) and the rest of the execution time (*Execution*). For the first group (concurrent lock-free algorithms), with traditional fences, we can see that fence stalls account for 8%-29% of the total execution time. Actually, the fence overhead can be larger depending on the programs using them. Concurrent algorithms have to guarantee correct data accesses by multiple threads, and fences are used to guarantee this goal under relaxed consistency models. However, if data is not frequently accessed by multiple threads concurrently, address-aware fences are able to utilize this opportunity to reduce the fence overhead. In the second group (SPLASH-2 programs), fences are inserted to ensure sequential consistency. Similarly, they experience significant overhead due to fences (about 10% on average). However, since fences are inserted conservatively, many dynamic fence instances are unnecessary, some of which were illustrated in Fig. 1. With address-aware fence, only very few fences have to take effect and delay the memory accesses that follow them. On an average, address-aware fence improves the performance of all benchmark programs by 12.2%. More importantly, we can see that address-aware fence only induces negligible execution time overhead – a fence can retire as long as it has obtained its watchlist, which can be fetched from the directory efficiently. On the other hand, traditional fence has to stall the pipeline until all outstanding stores have completed, which takes a much longer time to access the memory or invalidate shared copies.

(Scalability) We vary the number of processors with 4, 8 and 16 processors and measure the execution time of benchmark programs from SPLASH-2. The results are shown in Fig. 7, where data are presented in the same way as in Fig. 6. Each benchmark program has execution time of traditional fence and address-aware fence with 4, 8 and 16 processors, respectively. We can see that, with different numbers of processors, the execution time overhead induced by

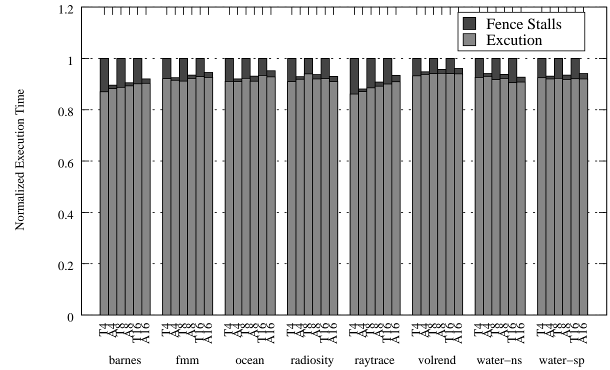

Figure 7: Scalability – Tn represents traditional fence with n processors and An represents address-aware fence with n processors.

address-aware fence is not affected significantly. Therefore, the implementation appears scalable. In fact, our implementation leverages directory-based cache coherence protocol, piggybacking required information on coherence transactions. Without introducing centralized structures, the implementation maintains the level of scalability delivered by the directory-based cache coherence.

5.2 Space and Traffic

Benchmarks	Active buffer %Emp.	Active buffer %<64	#Pend. stores	#Delay /1K inst.	Traffic %Inc.
barnes	77.9	98.7	2.2	0.0	13.0
fmm	85.5	99.4	2.8	0.0	18.6
ocean	59.6	84.1	8.7	0.0	9.7
radiosity	78.1	99.5	3.6	0.0	17.4
raytrace	82.5	98.9	2.3	0.0	22.1
volrend	75.2	98.7	4.5	0.0	13.9
water-ns	86.4	99.8	4.1	0.0	8.9
water-sp	89.0	99.9	5.2	0.0	12.5
AVG.	79.2	97.5	4.2	0.0	14.7

Table 6: Characterization of space and traffic.

To support address-aware fence, we introduced active buffer and watchlist buffer to record information and leveraged cache coherence to avoid fence order violation. Table 6 characterizes space and traffic induced by address-aware fence. We conducted the experiments using SPLASH-2 benchmark programs with 8 threads.

Recall that, active buffer stores addresses of active memory accesses and fences, and two consecutive loads or two consecutive stores are merged when they have the same block address. Column 2 shows that, on average, the active buffer is frequently empty – at the rate of 79.2%. When it is empty, memory accesses at the top of ROB can retire immediately, as there is no active fence prior to them. To size the active buffer, we tracked the number of active memory accesses during execution, and found that this number is less than 64 most of the time (97.5% on average as shown in Column 3). *ocean* has more than 64 active accesses most frequently (15.9%) among all programs, but the other benchmark programs almost never exceed 64 active accesses. Thus, we used 64 entries in our implementation. Besides, for watchlist buffer, we used the number of entries equal to the number of processors, with each entry recording the watchlist obtained from the corresponding processor.

A watchlist is obtained from the directory, consisting of the addresses of pending stores being serviced in the direc-

tory. Column 4 shows the average number of block addresses compressed in the directory when a processor requests for a watchlist. As we can see, the average number is 4.2, which is small. In our implementation, we compressed the addresses into a watchlist of 160 bits, which are enough to obtain very low false positive. Column 5 shows the number of external cache coherence transactions that are delayed due to its hit in the active buffer. We can see that all of them are 0.0 per 1K instructions. So it has little effect on the performance. Column 6 shows the traffic increase for each program. Address-aware fence induces additional traffic to transfer watchlists between processors and the directory, and this is the main source of the additional traffic. On an average, the traffic increases by 14.7%, which is modest.

(Hardware Cost Summary) In the highest performing configuration, address-aware fence only adds active buffer and watchlist buffer to each core. The active buffer has 64 entries, each of which has about 8 bytes; the watchlist buffer has 8 entries (for 8 processors), each of which has about 20 bytes. The above two buffers amount to a total of 672 bytes. We expect the extra power consumption of our technique to be small compared to the state-of-the-art aggressive speculative techniques, as the area overhead of our technique is small and it does not have to maintain speculative states or require rollback associated with misspeculations.

6. RELATED WORK

Our work is closely related to the techniques listed in Table 1, which are also aimed at reducing overhead of fence operations non-speculatively. In [36], Praun *et al.* proposed *conditional memory ordering* (CMO) based on the observation that memory ordering instructions used on acquire and release of a lock are often unnecessary. The goal of CMO is to optimize and reduce the cost of the acquire-release memory synchronization protocol using a purely runtime technique. However, it does not address the memory ordering situations using fences as shown in Fig. 1. Ladan-Mozes *et al.* [17] proposed *location-based memory fences* (`l-mfence`) to reduce fence overhead. It is a lightweight solution, but it is limited to the Dekker-like algorithms. Our prior work [20] proposed *conditional fence* (C-Fence) to enforce sequential consistency. Although it can greatly reduce dynamic fence instructions, it relies on compiler to provide fence associates information, and stalls due to fence associate conflict are still conservative as shown in our experiments. On the other hand, our work aims at reducing all kinds of memory ordering overhead induced by fences, and by looking at the addresses of memory operations across fences at runtime, we require much less fence stalls.

There has been work on optimizing lock implementations. Thin locks [2] and other refinements [16, 28] strive to reduce the overhead of locking in the context of Java. The basic idea is to allow a particular thread to reserve a lock, and hence acquisitions of the lock by the reserving thread can be performed efficiently. [35] proposed a fast biased lock, which simplifies and generalizes prior implementations of biased lock. The above work focuses on reducing the frequency of atomic RMW operations for implementing locks, while our work reduces the memory ordering overhead induced by fence instructions. Speculative lock elision [29] and lock elision with transactional memory [8, 34] both use speculative technique to dynamically eliminate lock operations. However, our technique does not simply focus on lock operations but also fences which are also used for lock implementation, and our technique does not require speculation.

Extensive research has been conducted on implementing sequential consistency (SC) efficiently. [11, 19, 30] rely on static analysis to minimize the number of fences which are required for enforcing SC, and [3, 5, 14, 15, 37, 21, 32] are various runtime solutions for implementing SC. Besides, in [23, 25, 31], researchers also argued that SC violation should be treated as exceptions. However, these techniques are designed to improve SC performance and provide programmers with a strengthened shared memory system, while our work is to reduce the cost of fence instructions. Although some of the above techniques can be adjusted to address the fence overhead [3, 5, 15, 37], they have to employ speculation. Memory accesses after a fence can be speculatively retired when there is any pending memory operation before the fence. However, the hardware complexity associated with aggressive speculation can hinder wide-spread adoption. [21, 32] are non-speculative SC implementations. End-to-End SC [32] proposes to identify thread-local and shared read-only data, and enforces SC by only ordering the accesses to remaining data. This approach is also useful for reducing the memory ordering overhead due to fences. But address-aware fence is more aggressive, as it further allows accesses to shared data to be reordered. In our prior work [21], we propose *conflict ordering* to enforce SC efficiently without speculation. However, address-aware fence only focuses on memory orderings enforced by fences which are present in the programs. The processor only initiates the process of handling fences when fences are encountered.

7. CONCLUSION

In this work, we propose a hardware solution *address-aware fence* to reduce the overhead due to fence instructions without speculation. Our technique is implemented in the microarchitecture without instruction set support and is transparent to programmers. Address-aware fences only enforce memory orderings that are necessary to maintain the effect that the traditional fences strive to enforce, while other fences are dynamically eliminated. Our experiments conducted on a group of concurrent lock-free algorithms and SPLASH-2 benchmarks show that address-aware fences eliminate nearly all the overhead associated with traditional fences and achieve an average performance improvement of 12.2% on all benchmark programs.

ACKNOWLEDGMENTS

We would like to thank all reviewers for their helpful comments and advice for improving this paper. This research work is supported by the National Science Foundation grant CCF-0963996 to the University of California, Riverside, and by the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council to the University of Edinburgh.

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. PLDI '98, pages 258–268.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. ISCA '09, pages 233–244.

[4] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. PLDI '07, pages 12–21.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. ISCA '07, pages 278–289.

[6] D. Chase and Y. Lev. Dynamic circular work-stealing deque. SPAA '05, pages 21–28.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2001.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. ASPLOS '09, pages 157–168.

[9] E. W. Dijkstra. Cooperating sequential processes. *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138, 2002.

[10] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew. Detecting and eliminating potential violations of sequential consistency for concurrent C/C++ programs. CGO '09, pages 25–34.

[11] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. ICS '03, pages 285–294.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. ASPLOS '91, pages 245–257.

[13] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. ISCA '91, pages 355–364.

[14] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. PACT '02, pages 179–188.

[15] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? ISCA '99, pages 162–171.

[16] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. OOPSLA '02, pages 130–141.

[17] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. SPAA '11, pages 75–84.

[18] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, Apr. 1983.

[19] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.*, 50(8):824–833, 2001.

[20] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. PACT '10, pages 295–306.

[21] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient sequential consistency via conflict ordering. ASPLOS '12, pages 273–286.

[22] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. PLDI '12, pages 429–440.

[23] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. ISCA '10, pages 210–221.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05, pages 190–200.

[25] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: a simple and efficient memory model for concurrent programming languages. PLDI '10, pages 351–362.

[26] M. M. Michael. Scalable lock-free dynamic memory allocation. PLDI '04, pages 35–46.

[27] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. PODC '96, pages 267–275.

[28] T. Ogasawara, H. Komatsu, and T. Nakatani. To-lock: Removing lock overhead using the owners' temporal locality. PACT '04, pages 255–266.

[29] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. MICRO '01, pages 294–305.

[30] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[31] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient processor support for DRFx, a memory model with exceptions. ASPLOS '11, pages 53–66.

[32] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. ISCA '12, pages 524–535.

[33] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[34] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. PACT '09, pages 3–14.

[35] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. PACT '10, pages 65–74.

[36] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu. Conditional memory ordering. ISCA '06, pages 41–52.

[37] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. ISCA '07, pages 266–277.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. ISCA '95, pages 24–36.