Fence-Free Synchronization with Dynamically Serialized Synchronization Variables

Yang Hong, Yang Zheng, Haibing Guan, Binyu Zang, and Haibo Chen, *Senior Member, IEEE*

Abstract—Memory fences are widely used to ensure the correctness for synchronization constructs on machines with relaxed consistency models. However, they are expensive and usually impose over-constrained ordering that causes unnecessary CPU stalls. In this paper, we observe that memory fences in TSO are merely intended to order synchronization variables. Based on this observation, we rethink the hardware-software interface of synchronization constructs on multicore processors and propose a new design called Sync-Order that differentiates synchronization variables (*sync-vars*) from normal ones. Sync-Order reduces hardware complexity such that the processor only needs to serialize the ordering among *sync-vars*. Its simplicity makes it easy to be integrated to the directory controller and it supports distributed directory, a missing feature in prior designs. We show that Sync-Order eliminates traditional fences on all sides of synchronization constructs (instead of only one side in prior work) and requires small effort for a programmer or compiler to annotate *sync-vars*. Our experimental results show that Sync-Order significantly reduces CPU stalls and boosts the performance of a set of synchronization constructs and concurrent data structures by 10 percent; meanwhile, the fence overhead of full applications from SPLASH-2 and PARSEC is reduced from 42 to 3 percent.

Index Terms—Fences, synchronization, memory consistency, sequential consistency, parallel programming, shared-memory multiprocessors

1 Introduction

Multicore applications and OS kernels intensively rely on synchronization constructs to handle concurrency. However, one key limiting factor of synchronization performance is the memory fence (or briefly fence), a set of instructions that prevent hardware from dynamically reordering memory accesses. Fences are essential to guarantee the correctness of synchronization constructs (e.g., Dekker's algorithm, bakery algorithm) which typically assume an underlying shared-memory system of sequential consistency [18]. While fences are designed as a low-overhead coordination compared to heavyweight synchronization mechanisms like locks, applications may still experience high latency due to fences. For example, the commodity hardware implementation of a fence instruction on Intel processors incurs 20-200 cycles overhead [9]. The reason for the delay is that memory accesses after a fence cannot complete until all accesses before the fence are globally visible. Some commodity processors like Intel Xeon even drain the entire store buffer upon a fence.

Because excessive fences degrade the performance of concurrent programs relying on fence-based synchronization mechanisms, there has been extensive research trying to reduce the fence overhead, which can be categorized into two main threads. The first thread is to reduce the fence overhead by allowing aggressive speculation [8], [9], [17], [19], [20]. Such designs dynamically detect possible races among multiple instruction streams and skip fences when reordering instructions does not hamper correctness. However, while such designs can effectively reduce the number of fences enforced, they usually require highly complex hardware and none of them so far supports distributed directory. The second thread is to avoid using fences when implementing synchronization primitives such as [6], [23], [25]. Such synchronization algorithms safely remove fences on fast paths based on the boundedness of the store buffer that a store buffer entry can be observed by other cores within bounded time. Yet, those approaches are usually algorithm-specific and only eliminate fences on one side of the synchronization parts.

We observe that the function of fences, which is ensuring sequential consistency for accesses to a set of variables (e.g., flags in Dekker's algorithm), can be achieved by enforcing specific inter-processor data dependencies instead of intra-processor serialization with fences. To this end, we rethink the hardware-software interface of synchronization design and propose a combined approach, *Sync-Order*, to enforce the correctness of parallel programs without fences. Our key idea is to distinguish those memory accesses with ordering requirements from the ordinary ones. We denote the variables that are key to the correct semantics of a program as *sync-var*s and the instructions accessing them as *sync-op*s.

Instead of inferring sync-vars and sync-ops from the context of a fence instruction, Sync-Order relies on programmers or the compiler to properly annotate variables that should be properly ordered. The processors can dynamically detect and enforce the annotated sync-ops to avoid violation situations without unnecessary stalls. It is

- *Y. Hong, Y. Zheng, and H. Guan are with the Shanghai Key Laboratory for Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {yang.hong, zhengy, hbguan}@sjtu.edu.cn.*
- *B. Zang is with the Software School, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: byzang@sjtu.edu.cn.*
- *H. Chen is with the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: haibochen@sjtu.edu.cn.*

Manuscript received 10 Apr. 2016; revised 14 Nov. 2016; accepted 16 Nov. 2016. Date of publication 29 Nov. 2016; date of current version 15 Nov. 2017. Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2633353

important to note that Sync-Order is not a complete replacement for fences, but an incremental optimization for existing synchronization algorithms with simple hardware modification. This is important as it provides an evolutionary path to evolve existing fence-based synchronization to such a fence-free design.

We design and implement a prototype of Sync-Order using a simulated eight-processor multicore architecture of TSO consistency model with a distributed directory-based coherence protocol. We choose TSO because it has been widely adopted in commercial processors [27]. We compare Sync-Order against conventional fence mechanism without post-fence speculation using both microbenchmarks and parallel programs from PARSEC and SPLASH-2. Our experimental results show that Sync-Order significantly reduces CPU stalls and boosts the performance of a set of synchronization constructs and concurrent data structures by 10 percent while the percentage of delayed cycles in full applications is reduced from 42.23 to 3.33 percent.

In summary, this paper makes the following contributions:

1) A new synchronization mechanism for the TSO memory consistency model called Sync-Order that only dynamically constrains *sync-vars* (Section 3).
2) An approach that leverages existing compiler analysis to convey to hardware the information of memory operations whose ordering must be enforced (Section 4).
3) A simple hardware extension that is more lightweight than prior work yet supports distributed directory (Section 5).
4) A simulation-based implementation as well as a set of evaluations that confirms the efficiency of Sync-Order (Section 7).

2 Background and Motivation

2.1 Memory Consistency and Delay Analysis

A memory consistency model specifies how the memory subsystem of a shared-memory multiprocessor behaves. Specifically, the model specifies the possible final states and impossible states of the shared memory with respect to the results of read and write operations executed by multiple processors.

Sequential consistency (SC), a model formally defined by Lamport [18], is the most intuitive model expected by most programmers. The two requirements of SC are:

1) Program order is respected: memory accesses from the same processor appear in the order specified by the program.
2) Single sequential order is maintained: all processors have the same view of the sequential order of all memory operations.

Despite several proposed SC architectures (such as [3], [5], [12]), commercial processors typically implement relaxed memory consistency models. Relaxed memory consistency models enable finer-grained ILP by reordering and overlapping memory accesses. Total Store Ordering (TSO), a model used in x86-TSO and SPARC TSO processors, relaxes the program order requirement of SC to facilitate store latency hiding and store-to-load forwarding.

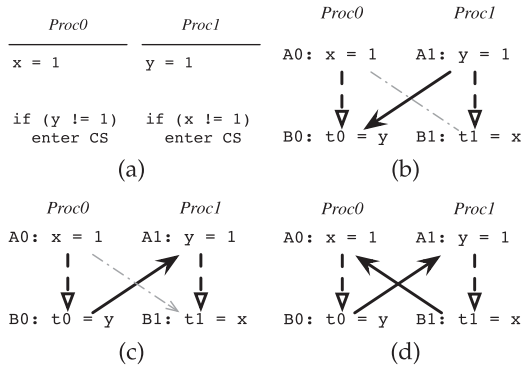


Fig. 1. Pseudo code for flag principle.

Specifically, TSO allows a load to be reordered with a previous store from the same processor targeting a different memory location. However, such optimizations are achieved at the cost of the good intuition of SC.

Consider the code in Fig. 1 with x and y initially being zero. This code pattern is designed to ensure exclusive access to the critical section under SC. Each solid arrow denotes an inter-processor *happened-before* relation between accesses to the same variable. If some store S happened before some load L to the same variable, the result of L should be the latest value written by S. Otherwise, L gets the value before S applies the modification. Each dashed arrow denotes the program order of two memory operations. In Fig. 1a, both *Proc0* and *Proc1* announce the intention to enter the critical section by writing 1 to x and y. Then each processor reads the other's flag to check whether someone else also wants to enter it. If *Proc0* receives the value of y as 1, as shown in Fig. 1b, it will not enter the critical section because of the *happened-before* relation $A1 \rightarrow B0$. Therefore, whether *Proc0* enters the critical section does not break the exclusiveness guarantee because at most one processors will enter the critical section.

In the second case (Fig. 1c), *Proc0* decides to enter the critical section as it observes y as 0 (indicated by the solid arrow $B0 \rightarrow A1$). Assuming the program order $A0 \rightarrow B0$ is respected, a programmer would infer $A0 \rightarrow B1$ with the transitive property. Therefore, *Proc1* shall not enter the critical section and the exclusiveness is guaranteed.

However, the result becomes counter-intuitive when TSO reorders loads before stores. This means the program order constraints $A0 \rightarrow B0$ and $A1 \rightarrow B1$ in Fig. 1c are not guaranteed by hardware. The case in Fig. 1d may happen if *Proc0* and *Proc1* both reorder the memory accesses. In that case, *t0* and *t1* both get old values of y and x, indicating happened-before relations $B1 \rightarrow A0$ and $B0 \rightarrow A1$. Inference based on the program order will lead to a cycle indicating that a total order of the memory operations is impossible to determine and the exclusiveness is violated.

Delay Analysis: Shasha and Snir [28] propose *delay analysis* to avoid such cases by delaying memory accesses until the preceding accesses have completed. In the theory of delay analysis, a partial *order* is an irreflexive, asymmetric, transitive relation. *Program order* (P) is defined as the order of instructions described by the program. *Conflict relation* (C) is an irreflexive, symmetric, transitive relation of a pair of memory accesses to the same memory location with at least one being a store operation. For example, Wx conflicts with

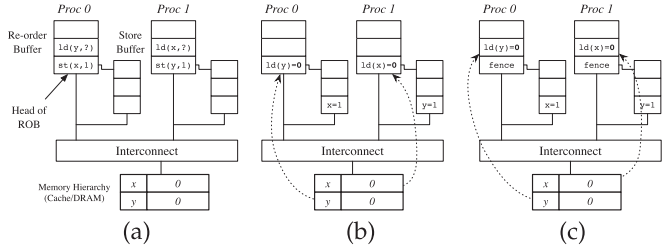


Fig. 2. SCV under TSO model.

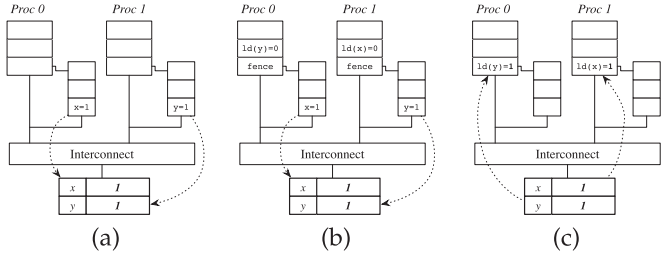


Fig. 3. Fences prevent an SCV.

Rx or *Wx* but *Rx* does not conflict with *Rx*. *Execution order* (E) is an *orientation* of C. In other words if memory operations u and v access the same address with at least one being store, they have conflict relation and uCv holds. Moreover, either uEv or vEu holds. When there is a cycle in $P \cup C$, this cycle may lead to an execution result that breaks the intuition of SC. This is called *sequential consistency violation* (SCV). From Fig. 1d, we can find that:

$$P = \{(A0, B0), (A1, B1)\}$$

$$C = \{(A0, B1), (B1, A0), (A1, B0), (B0, A1)\}$$

Therefore, the set of pairs that form a cycle is $\{(A0, B0), (B0, A1), (A1, B1), (B1, A0)\} \subseteq P \cup C$. The four pairs of relation cannot be satisfied simultaneously. TSO allows $(A0, B0)$ or $(A1, B1)$ to be removed and reach a total order where *Proc0* and *Proc1* both observe flags as 0, which is apparently impossible under SC.

The proposed approach to *breaking the cycle* is to find a subset of P called *delay set* (D) and to enforce the program order within the subset using hardware. Specifically, If uDv holds, the hardware prevents v from starting until u finishes. In the above example, $D = P = \{(A0, B0), (A1, B1)\}$. When D is enforced with hardware, the only way to break the cycle is to preclude $(B1, A0)$ or $(B0, A1)$ and the final states become legal under SC.

2.2 Modern Processors: Store Buffer and Fences

To see why relaxing the store to load order from the program order causes confusion, it is necessary to understand how store buffer changes the visibility of memory operations. In an out-of-order processor with TSO model, instructions are decoded into the *reorder buffer* (ROB) and dispatched in program order. Once all the required operands are ready, an instruction can be issued immediately out-of-order. A memory operation is issued as soon as its address operands are ready and committed when it is at the head of ROB. A memory instruction *completes* when its result becomes *globally visible* after being committed. A load is considered globally visible when the value loaded into the destination register is determined [13]. Hence, a load immediately completes when it is committed such that its result cannot be squashed. A store may have its value buffered in the *store buffer* to hide write latency after being committed. Thus, a store is globally visible when the value is flushed from the store buffer to the memory hierarchy (e.g., CPU cache). By forwarding values from the store buffer to later load operations reading the same variable, TSO efficiently achieves the illusion of SC for a uniprocessor.

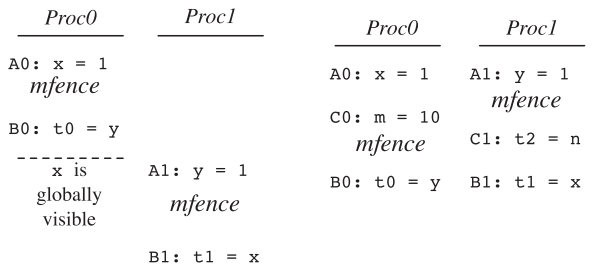
However, connecting multiple such uniprocessors does not guarantee SC to a multiprocessor. Because values in the store buffer are only accessible to the local processor, other processors may get stale values from the memory hierarchy. Fig. 2a shows the initial state of the four memory operations in the flag example with Fig. 1a. Both processors are about to commit two stores followed by two loads. After the store operations are committed, as shown in Fig. 2b, the two loads get issued and retrieve stale values from memory hierarchy before previous stores complete. Finally, stores are globally visible (Fig. 2c) after the completion of the two loads as if the two loads were reordered before stores. This also demonstrates the architectural cause for non-SC results in delay analysis.

To restore the intuition of SC, the TSO model provides fence instructions with the functionality of *delay*. A fence prevents post-fence memory instructions from being committed until all pre-fence memory accesses complete. In practice, a fence at the head of ROB does not complete until the store buffer is drained. Optimized implementations [11] allow post-fence loads to be issued and speculatively fetch data before being committed. Fig. 3a shows that two loads speculatively retrieve values as *0*s and wait for completion. Fig. 3b shows fences block later loads until stores are flushed. In Fig. 3c, updated cache lines of *x* and *y* send invalidations to processors to force the two loads to reload. Hence, fences can prevent unwanted reordering.

2.3 Issues with Fences

While the fence mechanism can prevent reordering, it has several inherent issues:

1) *Fences are Expensive.* Fence instructions virtually neutralize the effect of the store buffer that hides store latency and can result in high latency for synchronization constructs. According to Duan et al. [9], a single `mfence` instruction in Intel processors induces 20 to 200 cycles slowdown.
2) *Fences are Enforced Conservatively.* While the fence mechanism does not require any knowledge of memory operations in other processors, the serialization is sometimes unnecessary. In Fig. 4a, the fence delays are conservatively enforced for *Proc0* and *Proc1* despite that the memory operations do not overlap.
3) *Fences Cause Head-of-Line Blocking.* A fence blocks all memory operations after it. However, for specific algorithms, reordering irrelevant loads and stores targeting different addresses does not compromise the correctness. As shown in Fig. 4b, $B0$ need not wait for $C0$ to complete and $C1$ need not wait for $A1$. Blocking the victim memory accesses causes additional delay.



(a) No overlapping (b) Victim accesses

Fig. 4. Defects of fences.

2.4 Issues with Prior Approaches

A number of techniques [9], [17], [19], [20] have been proposed to reduce the cost of fences. Among them, WeeFence [9] and Address-aware Fences [20] are the most aggressive ones that dynamically detect SCVs and only enforce fence semantics when reordering can lead to a potential SCV. They take a similar approach that allows a load to complete before pre-fence stores complete *only* when there is no risk of an SCV. However, they usually require some relatively complex hardware mechanisms and still have some limitations, as show in Table 1.

First, they require that fences must come in pairs to form fence groups. Consider the case in Fig. 5, there is no need to add a fence between $(A1, B1)$ because TSO does not reorder store with store. Therefore when *Proc0* is detecting global pending memory operations, it cannot get the information from *Proc1*. The conflicting order of $B0$ and $A1$ cannot be detected and a cycle would occur by reordering $B0$ ahead of $A0$. To solve this problem, both techniques add additional complexity to the architectural design, namely *watch list* and *Bypass Set List* accordingly.

Second, they have consistency issues when retrieving global states from a *distributed directory*, which is a common way to scale cache coherence to large multiprocessors. Because a fence does not explicitly specify which memory operations may cause conflicts, both approaches collect the set of pre-fence stores on each processor and expose to other processors. It is challenging to maintain the consistency of such information pieces from multiple directory controllers. Therefore, their approaches have to frequently fallback to the traditional fence mechanism when using a distributed directory.

3 Approach Overview

Sync-Order is a new synchronization mechanism that prevents SCVs without inserting fences. It avoids SCVs while still allowing the most acceptable extent of store-load reordering. Our work focuses on the TSO model because TSO is currently widely used in commodity architectures such as x86-TSO.

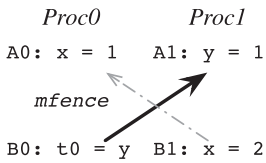


Fig. 5. Single fence problem.

Instead of using fences to serialize *local* execution of memory operations according to the program order, Sync-Order seeks to preclude *inter-processor* data dependencies that can potentially result in SCVs. It monitors data dependencies among memory accesses to specified variables and coordinates such accesses to avoid SCV hazards. A load can still complete ahead of a preceding store as long as inter-processor data dependencies concerning those accesses cannot result in an SCV. Sync-Order reduces unnecessary serialization cost of fences and hence accelerates synchronization constructs.

It is important to note that Sync-Order does not aim at a complete replacement for fences, but an incremental optimization with simple hardware modification for existing synchronization algorithms. When critical memory accesses for ensuring correctness are clearly identifiable, Sync-Order is superior to fences due to its lower overhead and more permitted reorderings.

In this section, we first describe the overlapping of memory accesses where non-SC results may arise. We then propose Sync-Order by showing how to prevent incorrect results by delaying specific load accesses.

3.1 Sync-Order Semantics

The first step to detecting inter-processor data dependencies is having a global order of beginnings and ends of memory operations on all processors. Fig. 6a shows the timeline of an interleaving of memory accesses with an SCV risk. In each processor, the store operation is issued before the load but the load completes before the store. However, `P0:Ry` completes before `P1:Wy` and `P1:Rx` completes before `P0:Wx`. The resulting data dependencies are shown as $\texttt{P0 : Ry} \rightarrow \texttt{P1 : Wy}$ and $\texttt{P1 : Rx} \rightarrow \texttt{P0 : Wx}$. This is a sign of an SCV because both loads seem to be executed before stores, and we cannot decide a total order of the four accesses which is consistent under SC.

Sync-Order coordinates memory accesses based on one basic principle: *for two conflicting operations W and R from two processors, if R is issued after W and before W completes, R must not complete until W completes.*

TABLE 1
A Comparison of Efforts in Reducing Fence Overhead

	lmfence	C-Fence	WeeFence	AAF	Conflict Ordering	End-to-End SC	Sync-Order	TBTSO	prwlock
Hardware solution	✓	✓	✓	✓	✓	✓	✓		
Remove fences							✓	✓	✓
Allow skipping fences	✓	✓	✓	✓	✓		✓		
Compiler assistance		✓				✓	✓		
Global state		✓	✓	✓			✓		

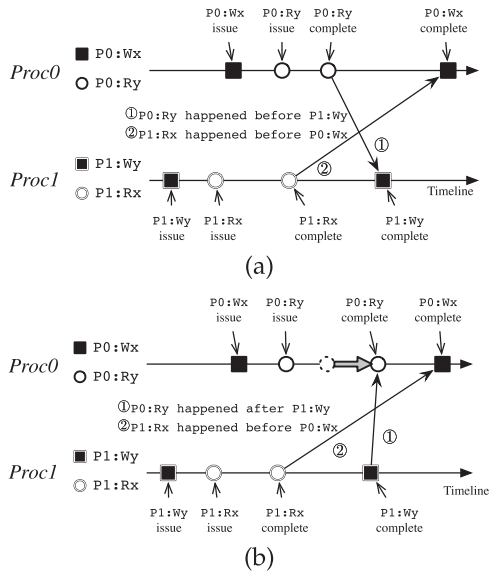


Fig. 6. Examples of Sync-Order.

In Fig. 6b, when the load P0:Ry is issued, a conflicting store P1:Wy from *Proc1* has already been issued and is still in progress. *Proc0* becomes aware of the order of the issue events of P1:Wy and decides P0:Ry must be correctly ordered. So the load is kept from reading the value until P1:Wy updates the value globally. On the other hand, P1:Rx is issued before P0:Wx and cannot observe any ongoing conflict access. Hence, *Proc1* can commit P1:Rx without stalling. Note that the program order is still violated after coordination as P0:Ry is globally visible before its preceding store P0:Wx and P1:Rx before P1:Wy. Nevertheless, the violation result is effectively avoided. This indicates that the program order need not be always enforced. Compared with fence-based SC enforcements, Sync-Order shortens the delay required.

3.2 Correctness of Sync-Order

We now present an informal proof to show that Sync-Order guarantees mutual exclusion when being used in flag-based synchronization algorithms. The proof discusses several interleavings of issue events and completions, and explains why Sync-Order can guarantee the correctness.

We say that a load retrieves the effective value at completion. Also, a store makes sure the new value is updated in the memory hierarchy at completion. The period between issue stage and completion represents the time that a memory access lasts, and is marked as the range between two black cubes in Fig. 7.

Theorem 1. *Given the code pattern in Fig. 1a, Sync-Order ensures that at most one processor will enter the critical section.*

Proof. If the execution traces of two store operations do not overlap, the processor that issues the store later does not enter the critical section. This case is shown in Fig. 7. The store operation from *Proc1* is issued at *t1* and completes at *t2* while the store from *Proc0* is issued at *t3* and completes at *t4*. There is no overlapping of the two ranges from the two processors. In this condition, reordering loads and stores within each processor does not yield non-SC results because loads in *Proc0* shall *always* observe the new value.

Considering the cases when two operations overlap. The condition where the time ranges of the two stores exactly overlap is a special case of Fig. 7c thus will not be discussed. Without loss of generality, we assume *P1* issues the store first. The store P0:Wx is issued at time *t2* and completes at *t4*. Similarly the store P1:Wy is issued at time *t1* and completes at time *t3*. We divide the time range into 3 phases separated by pipeline events at *t1*, *t2*, *t3* and *t4*, namely α, β and γ.

1) *Proc1 Issues P1:Rx in Phase α.* On one hand, because at the issue time of P1:Rx, P0:Wx has not been issued. Sync-Order does not enforce the delay of the load. Hence, P1:Rx may get the old value and *P1* may enter the critical section. On the other hand, If P0:Ry is issued in phase γ, P1:Wy has already completed thus P0:Ry must get the new value. As a result, *P0* will not enter the critical section. If P0:Ry is issued in phase β, Sync-Order will detect this since the load is issued between the issue and completion stages of a conflicting store. Sync-Order will delay the completion of P0:Ry to phase γ as shown in Fig. 7c. Finally, *P0* gets the new value of y and does not enter the critical section.
2) *Proc1 Issues P1:Rx in Phase γ.* Sync-Order will discover that P1:Rx is issued between the issue and completion stages of P0:Wx and enforce the delay action. After getting the new value of x, *P1* also *backs off* from the critical section. For *P0*, the discussion is the same for case 1. Finally, neither processors will enter the critical section and mutual exclusion is also guaranteed. □

4 Synchronization Variables

Though Sync-Order can enforce SC without fences, efficient identification of sync-vars is essential since applying

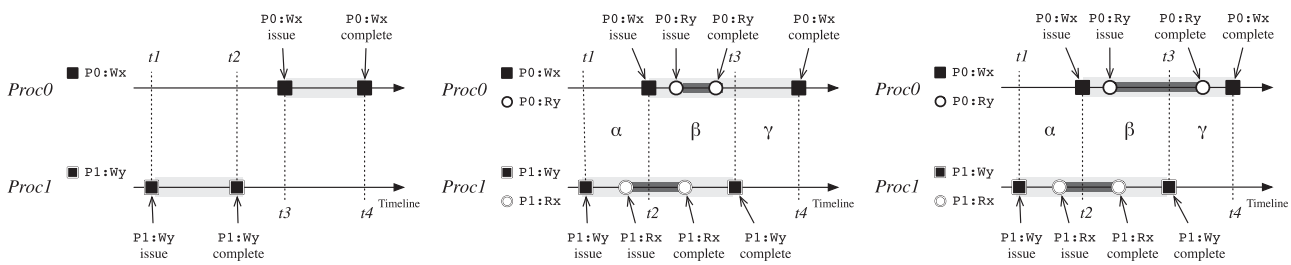


Fig. 7. Overlapping of store windows.

dynamic detection to every instruction is inefficient and unrealistic. This section describes the compiler analysis that assists processors to reduce the detection overhead.

4.1 *Sync-Vars*: Reducing Detection Overhead

We try to reduce the detection overhead based on the observation that *only a small portion of memory operations are of interest to enforce SC*. For TSO architectures, the occurrence of an SCV is the result of several conditions. First, there must be data races on multiple shared variables. Second, the racy memory operations must overlap in time. Third, the processors must reorder loads and stores of the racy accesses to form dependency cycles. The main implication here is that *we only need to enforce SC of those racy accesses to make the whole program correct*.

So we aim at identifying critical memory accesses and only applying detection to those operations. We refer to such operations as *synchronization operations* (**sync-ops**). The memory locations accessed by sync-ops are named *synchronization variables* (**sync-vars**). For example, the flag-based mutual exclusion algorithms (e.g., Dekker's algorithm and Lamport's bakery algorithm) rely on a set of flags to guarantee the exclusive access to a critical section; an SCV that breaks the exclusiveness of such algorithms is only related to the flag variables. Hence, we treat the flag accesses as sync-ops and eliminates SCVs related to such sync-ops.

4.2 Identifying Sync-Vars

Though manually identifying and annotating sync-ops and sync-vars in a parallel program is possible, it requires much experience from programmers. We propose to utilize compiler analysis to automate the process to reduce the burden of programmers.

Although Sync-Order differs from the fence mechanism in that Sync-Order focuses on *inter-processor data dependency* while the fence mechanism focuses on *program order enforcement*, there are some useful techniques with the fence mechanism which are also applicable to Sync-Order. These two different approaches have one common thing that *they both change the ordering of memory operations that could form cycles*. Therefore, the algorithms and analysis techniques to reduce the number of fence insertions can also help reduce the number of sync-ops.

Fortunately, there have been a lot of work on automatic insertion of fences to enforce SC of parallel programs. Some are based on the conservative algorithm using thread escape analysis ([7], [10]) to find delay sets. Liu et al. [22] uses dynamic synthesis to expose violations of memory model specification and insert fences to prevent violations. Some [2], [15], [16] focus on reachability of error states in TSO, PSO and RMO.

In practice, we reuse Memorax [2] to identify the memory operations that can result in violations. The result of this tool is several *synchronization sets* instead of fence locations. For the flag example, the synchronization sets are $\{\text{“}P0 : Wx\text{”}, \text{“}P1 : Wy\text{”}\}$. The set is the same set required by Sync-Order. We encode such information in program binaries. The processors can dynamically distinguish and enforce sync-ops upon detecting them.

Due to the complexity of existing fence insertion algorithms, it is hard to verify the whole program accurately. Here, we pick the core synchronization components which are key to correctness (e.g., Dekker's lock, bakery lock) for verification and annotation. For other parts of the program, we allow false positives in identifying sync-ops and sync-vars, which means the compiler tool may treat some victim memory accesses as sync-ops. This does not affect correctness as the falsely annotated variables do not contribute to the SCV cycles and Sync-Order will not enforce ordering on them. Meanwhile, the rest of the program not analyzed and verified can still leverage the fence mechanism to ensure correct ordering.

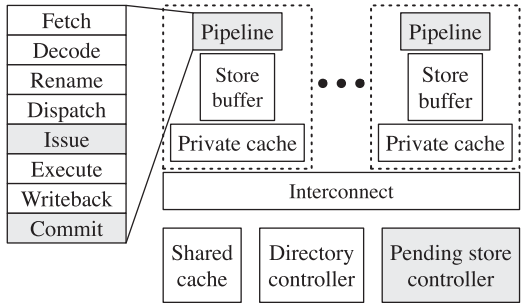


Fig. 8. Architecture of Sync-Order.

4.3 Data-Race Free Programs

Some programming languages [4], [24] specify the memory models implemented by the language runtimes. Those languages also provide high level abstraction of memory objects and language-specific concurrency support. For example, Java memory model promises that a Java program without data races behaves as if it was under SC consistency even though Java runtime allows many kinds of reordering. A DRF (data-race free) program ensures that all conflicting memory accesses are properly annotated by programmer or with the help of the compiler. While such annotations indeed simplify the sync-var identification in high level languages, the compiler analysis is still required to handle low-level languages without rich semantics. We will extend Sync-Order to utilize the rich semantics from language support to ease the process of sync-var identification in the future.

5 Architectural Support

This section discusses in detail the hardware extension to the existing TSO processor architecture with the sync-var information collected in the compiler analysis step. Fig. 8 shows the overall architecture of Sync-Order with our modifications in gray. We then describe the components of our extension and how to exploit the sync-var information to enforce order.

Sync-Order mainly adds two extensions to the existing TSO processors. First, the processor pipeline must identify sync-ops dynamically. We divide sync-ops into two types: *sync-store* and *sync-load*. The processor sends out messages for different states of *sync-store*s and *sync-load*s and check whether an SCV is possible for some sync-ops. If so, some specific *sync-load* must be delayed at commit stage. Second, a distributed controller is added to allow CPUs to share pending store information. The controller collects messages from processors and responds to queries about the state of specific sync-vars. There are four types of messages that the processor pipeline sends to the controller and one type that controller sends to the pipeline.

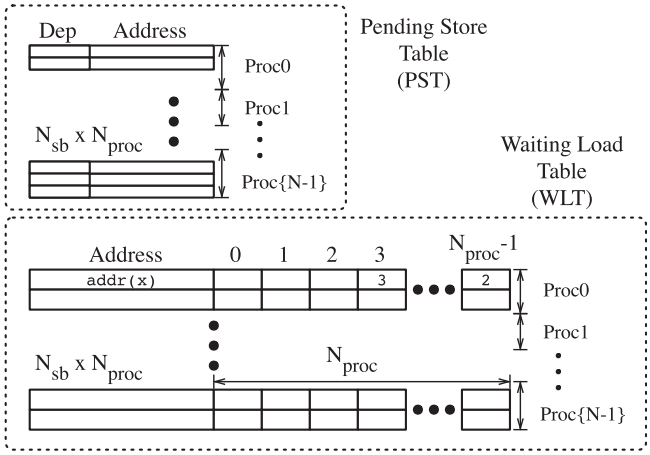


Fig. 9. Tables managed by *PSC*.

5.1 Pipeline Extension

First, we extend the processor pipeline with the ability to identify sync-ops from the instruction streams. One solution is adding new instructions "sync-store" and "sync-load" which are memory operations that trigger SCV detection. Theoretically, the identification of sync-ops can be done at decode stage. However, we choose another solution to differentiate sync-ops by allocating a range of linear addresses for sync-vars and marking accesses to this range as sync-ops. So the ISA remains unchanged and the program binaries require no modification. But sync-op identification is pushed off until the issue stage when the address calculation is done. A flag representing whether the operation is sync-op is raised in the store buffer entry to notify future pipeline stages. The problem with this approach is that the compiler needs to instrument the program variables and the memory allocator must allocate dynamic sync-vars in the reserved area.

At the issue stage, when the addresses of *sync-load* and *sync-store* are calculated, the processor must send messages to the distributed controller to update or query states of sync-vars. Messages must be sent out at the issue stage because address operands are not ready until then. The processor must revoke and resend messages of a sync-op if the dependency instruction is invalidated.

5.2 Processing Sync-Ops

A sync-store is a store that updates a sync-var. Before a store is issued, the processor calculates the target address from the operands. If the store turns out to be a sync-store, the processor sends a *set* message to the controller to notify the start of a sync-store. At the same time, the sync-store continues to prepare data and puts the result in the store buffer. When the processor gets a message from the cache that the value of the sync-store has been updated in the cache, the sync-store completes. The processor sends an *UNSET* message to the controller to notify the completion event.

A sync-load operation is a load that reads from a sync-var. The target address is also calculated before the issue stage. Upon issuing a sync-load, the processor first sends a *get* message to the controller to query the state of the target sync-var. If the sync-var is being modified by some sync-store indicated by a flag set by a previous *set* message, the processor sets a flag in the re-order buffer (ROB) entry of the sync-load. If the sync-load does not need to wait for a pending sync-store, it directly reads the value of sync-var from the cache. At the final commit stage where the entry at the head of the ROB is committed and removed from ROB, the sync-load instruction with a flag set will be prohibited from committing. This causes a cycle of committing nothing for the pipeline. In the following cycles, the sync-load cannot be committed until a *WAKEUP* message from the controller signals the completion of a pending sync-store. If a message comes notifying the cache line of the sync-var is modified before the sync-load is unblocked, the stale result is also squashed and the sync-load has to be redone.

5.3 Pending Store Controller

As shown in Fig. 9, the *PSC*. is a controller in the memory hierarchy consisting of two tables. The *pending store table* (*PST*) keeps records of current pending stores from all the processors. It uses 64-bit addresses of sync-vars as the index and another 3-bit to record the dependency processor ID. The dependency field will be discussed in Section 5.5 In the worst case, all the processors can issue consecutive *sync-store*s as many as the length of the store buffer of one processor. To buffer all the pending requests, a maximum number of $N_{sb} \times N_{proc}$ entries are required, where N_{sb} is the number of entries of the store buffer and N_{proc} is the total number of processors. The other table, *waiting load table* (*WLT*), keeps all blocked *sync-load*s which are waiting for pending *sync-store*s to complete. Besides using 64-bit address as the index, the table uses another N_{proc} fields to count the pending stores on each processor for which this sync-load has to wait. This is the theoretical maximum size of the two tables. Later we will show how to practically determine the required size.

5.4 Message Handling

PSC. receives four kinds of messages as shown in Table 2: *SET*, *UNSET*, *GET* and *ANNUL*. Each message only contains the target address, the processor ID and the message type. *SET* messages are sent at the issue stage of sync-stores.

TABLE 2
Message Types

Message	From	To	Function
SET	sync-store	PSC	Set the pending bit in the controller
UNSET	sync-store	PSC	Cancel the pending bit to denote completion of store
GET	sync-load	PSC	Query whether there are pending stores to the same address
ANNUL	sync-load	PSC	Cancel a previous get request
WAKEUP	PSC	sync-load	Allow waiting sync-load to continue

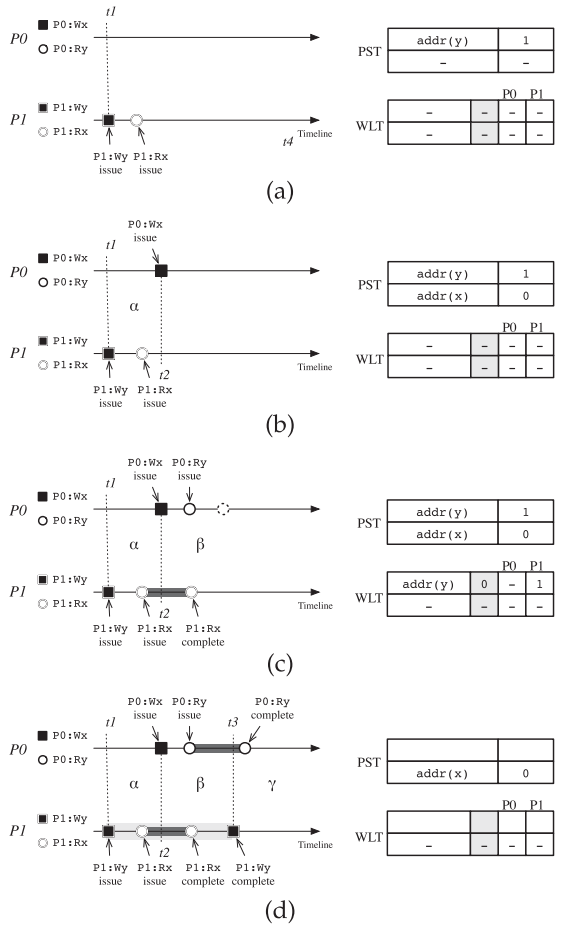


Fig. 10. Process of messages.

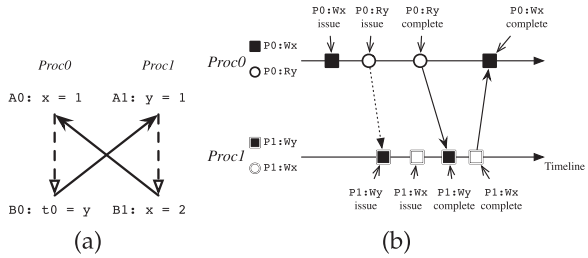


Fig. 11. Write-write conflict case.

On receiving a *SET* message, the *PSC.* allocates a new entry in the *PST* with the target address and the ID of the sender processor. The entry indicates that a modification to some sync-var is pending and any sync-load that observes this event should wait for the sync-store to complete. An *UNSET* message is sent after a sync-store completes. *PSC.* reacts to an *UNSET* message by removing the corresponding *PST* record entry to represent the completion of some sync-store.

The *GET* messages are sent at the beginning of the issue stage of sync-loads. Upon receiving a *GET* message, the *PSC.* first searches *PST* for all entries with the same linear address as the sync-load and then counts the number of matches on each processor. If found, the numbers of matches for all processors together with the address are recorded in a newly allocated *WLT* entry. If no such pending sync-stores are found, *PSC.* will not record it but immediately sends back a *WAKEUP* message to the processor.

When an entry in *PST* is cleared, the *PSC.* searches *WLT* for sync-loads waiting with that address. If found, those entries are updated by decreasing the counter for the processor which just sends the *UNSET* message. Once all counters of some *WLT* entry are zero, it is removed from *WLT* and a *WAKEUP* message is sent from *PSC.* to the waiting processor to allow that sync-load to be committed.

When multiple processors issue multiple *SET* messages targeting the same address to the *PSC.*, multiple identical entries are allocated in the *PST* with the same address. When an *UNSET* from one processor comes, those entries are cleared randomly. Because TSO guarantees that stores are flushed from each processor in the program order and does not reorder store with store, *PSC.* can be sure that *UNSET* messages come in the same order as *SET* from the same processor to the same sync-var. So it is sufficient for stalled sync-loads to care only about the number of finished stores to the target address by each core.

Example: In Fig. 10, two processors are executing the code of the flag pattern. In Fig. 10a, *P1* first issues `P1:Wy` and `P1:Rx`. The *PSC.* updates the state of sync-var `y` as pending by *P1*. *P1* sends a *GET* message to *PSC.* and the response indicates there is no conflicting pending sync-store. So `P1:Rx` does not wait. In Fig. 10b, *P0* issues `P0:Wx` and updates *PSC.*. In Fig. 10c, *P0* issues `P0:Ry`. Response of *GET* message for `P0:Ry` shows there is one pending sync-store modifying `y`. So it cannot be committed until it receives a *wakeup* message. However `P1:Rx` is not stalled and committed successfully. Meanwhile the *PSC.* allocates a record in *WLT* to show that *"a sync-load from P0 should be stalled until 1 sync-store from P1 completes"*. In Fig. 10d, `P1:Wy` completes and `P0:Ry` is released. So first the entry of sync-var `y` is removed from *PST* and then the entry in *WLT* is removed.

5.5 Write-Write Conflict

However, the Sync-Order design so far cannot detect SC violation caused by two conflicting writes. The case is shown in Fig. 11a. *P1* first writes to `y` then writes to `x`. Therefore, `P0:Wx` and `P1:Wx` become conflicting operations. Because Sync-Order does not delay stores, an SC violation may arise as shown in Fig. 11b. The root cause for this violation is that currently Sync-Order does not track the ordering of stores.

In this case, one truth is that `P0:Wx` is issued earlier than `P1:Wx`. This truth is provable because that `P0:Ry` fails to get the new value of `y` means that `P0:Ry`'s *GET* reaches the *PST* before `P1:Wy`'s *SET*. Given that Sync-Order ensures sync-ops are issued in program order, we can infer that `P0:Wx` is issued before `P1:Wx`.

Based on the observation, we propose a dependency tracking mechanism to solve the problem. Each entry in *PST* has a field for tracking its dependency. When a new entry is added to *PST*, it scans the whole table to find the most recent pending store to the same address. Then it records the previous stores's processor ID in its own field. When the cache controller receives a sync-store request, the cache controller first check the *PST* entry's dependency. Only when the dependent processor has completed the sync-store may the current sync-store continue. Otherwise, the cache controller bounces the request and the requester processor must retry.

This mechanism effectively enforces that the completion order of sync-writes conforms to the order of *SET* messages. In the scenario of Fig. 11b, the SC violation can be avoided because `P0:Wx` always sends *SET* message earlier than `P1:Wx`.

5.6 Additional Considerations

5.6.1 Read-Modify-Write Instructions

An RMW instruction usually combines several memory operations into one instruction. To execute an RMW atomically, a lock prefix must be added to opcode encoding. Standard spinlocks use atomic RMWs to atomically check and set flags. We assume that each atomic RMW implies a following conventional fence and flushes the store buffer. This is true on Intel machines [14]. Therefore, Sync-Order need not handle reordering across RMWs because the fence semantic naturally prevents pre-RMW accesses from being reordered with post-RMW accesses. As for reordering that happens on either side of an RMW, Sync-Order can handle it correctly.

5.6.2 Issuing Sync-Ops in Program Order

One important assumption in our approach is that sync-ops are issued in the program order. Sync-Order relies on the order of PST messages to reflect the program order on all processors. Given that the current implementation sends out PST messages in the issue stage after calculating the address, in-order issue is necessary for the correctness. Otherwise, Sync-Order may fail to detect possible SC violations.

However, we expect the influence to be small because only a small portion of instructions are sync-ops. Instead of enforcing the program order of sync-ops at the dispatch stage, we adopt the abort-redo approach. When a sync-op *A* is issued, the processor checks forward in the ROB. If any subsequent sync-op *B* in program order has been issued before the current sync-op, the processor must first redo *B* by sending the combination of *ANNUL* message and *SET* message to the *PSC.*. Because ROB entries are committed in the program order, the processor can always abort and redo previous execution of *B*. Hence, it is guaranteed that sync-ops are issued in program order.

5.6.3 Combine PSC with Directory Controller

We implemented a standalone controller as *PSC.*. Combining *PSC.* with the directory controller is possible. But existing messages for directory-based coherence transactions are insufficient for Sync-Order to enforce ordering. The main reason is that messages for store operations are sent too late to be used for ordering detection. When the store is being committed and the processor emits a writeback message, a load that follows the store may have already been issued or even committed.

To integrate Sync-Order with existing directory-based interconnect, there are some modifications required. A *SET* message must be sent at the issue stage of a sync-store to allow early detection of program order. This message is independent of existing cache coherence messages because typically store-related messages are sent when executing coherence transactions. An *UNSET* can be omitted because the *PSC.* can directly get the status of a particular cache line from the directory. When *PSC.* observes a cache line is successfully updated by some processor, it removes from *PST* the corresponding entry and sends back an *UNSET*. A *GET* message is attached to a READ message when a load is issued. A *WAKEUP* message is attached to the message that returns the value for some sync-load. If the sync-load should block, the value is not retrieved from cache until *PSC.* allows the sync-load to continue. An *ANNUL* message is sent when a sync-op is discarded and replayed due to some events (e.g., cache invalidation, branch misprediction, sync-op replay).

5.6.4 Duplicated Messages

One design decision of Sync-Order is that we allow a processor to issue multiple *SET* messages for multiple sync-stores to the same variable. This design complicates the structure of *PSC.* because the two tables have to treat duplicated *SET* messages as distinct instances. Nevertheless, the message mechanism is simplified. Otherwise, if a later sync-store accesses the same variable as the previous pending sync-store, the processor must invalidate the previous *SET* message using an *ANNUL* message and resend a new *SET* message. We think the simplicity of message mechanism outweighs the simplicity of *PSC.* because our detection method integrated with processor pipeline should be as efficient as possible.

5.6.5 Distributed Controller

Previous fence optimization techniques [9], [20] have consistency problems when retrieving information from distributed controllers. For Sync-Order, with the increasing number of processors, the size of the two tables maintained by *PSC.* also grows. To reduce the cost of address matching on a large number of entries, we propose a distributed design of *PSC.*, which partitions *PST* and *WLT* and distributes them in multiple controllers. A message is dispatched to one of the controllers according to the hash value of the address. With a distributed controller, all messages targeting the same sync-var go to the same controller. In reaction to a message the *PSC.* only updates entries having the address contained in the message in *PST* and *WLT*. Hence, all operations and events regarding one sync-var are serialized within a single *PSC.* partition. For Sync-Order, there is no need to update or query information of multiple sync-vars in a single message. Hence, the consistency problem caused by fetching memory operation information from distributed modules does not exist. A distributed directory may shorten Sync-Order message roundtrip time while the correctness is not affected.

In our simulation prototype, we have implemented a centralized *PSC.* as well as a distributed version. Since the performance of Sync-Order is not affected by distribution of *PSC.*, we use the distributed version by default in our evaluation.

5.6.6 Support More Processors

So far we use SCVs caused by data races on two processors for exposition simplicity. In fact, Sync-Order can be applied to detect SCVs involving any number of cycles. Take the program in Fig. 12b as an example, if an SCV happens, it is possible that `P0:Ry`, `P1:Rz` and `P2:Rx` will complete before stores and get old values. However in Sync-Order it is not possible.

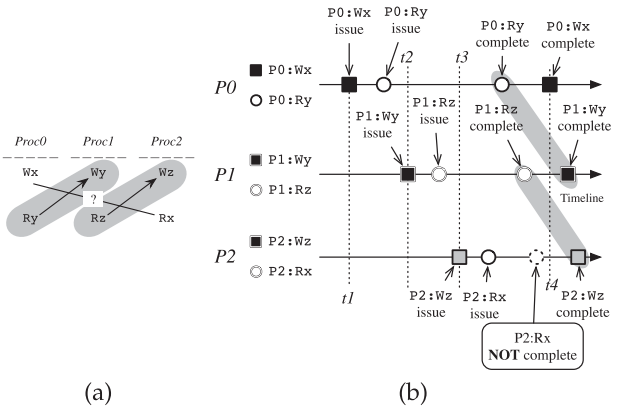


Fig. 12. Apply Sync-Order with 3 processors.

First, let us consider sync-ops {P0:Ry, P1:Wy}. If P0:Ry is issued after P1:Wy, Sync-Order guarantees P0:Ry will be stalled until P1:Wy completes and a violation is prevented. Otherwise, the violation is still possible. Similarly, P1:Rz must be issued before P2:Wz if a violation were to happen. The status of all operations is shown in Fig. 12b. From the previous assumptions, we have P0:Ry issued before P1:Wy and P1:Rz issued before P2:Wz. Combined with issue order requirement, we finally come to that P0:Wx is issued before P2:Rx. According to the semantics of Sync-Order, P2:Rx is guaranteed to observe the result of P0:Wx because it cannot be committed until the sync-store completes. So a cycle in Fig. 12a is impossible because the edge P2:Rx→P0:Wx is prevented.

5.6.7 Reduce Storage Cost

According to the structure in Section 5.3, the additional storage required by Sync-Order for an 8-core processor with 64-bit memory address width and 48-entry store buffer is about 76 KB. To reduce the storage overhead for tables, we can limit the number of messages sent to the *PSC.* from each processor. We regard a message as *consumed* when it no longer changes the behavior of a sync-op. A *SET* message is considered consumed when an *UNSET* is sent after completion of cache transaction. A *GET* message is considered consumed when either the sync-load does not wait or the sync-load is released from blocking. We can set a limit to the number of messages that have not been consumed. When a processor is going to send a 9th *SET* message while the limit of 8 is reached, the message will be held back until a message is consumed, for example when a sync-store completes. This optimization reduces the storage overhead of maintaining dependency of sync-ops at the cost of possible delay for Sync-Order messages. In evaluation, we will show that the maximum average number of entries per processor in the *PSC.* is relatively small.

TABLE 3
Simulated Architecture Specification

Architecture	8-core multicore CMP
Core	Out-of-Order, four-issue wide
ROB & Store Buffer	128-entry ROB & 48-entry store buffer & 48-entry load buffer
L1 cache	Private 128K, eight-way, two-cycle RT, 64B lines
L2 cache	Private 2M, eight-way, five-cycle RT, 64B lines
L3 cache	Shared 8M WB, eight-way, eight-cycle RT, 64B lines
Cache coherence	MOESI under TSO
On-chip network	Bus, one-cycle latency
Sync-Store	one-cycle message delay
Sync-Load	one-cycle message delay
Distributed *PSC.*	Each module contains a PST with 8×48 entries and a WLT with 8×48 entries, two-cycle access delay
DRAM latency	50 ns

TABLE 4
The *kernels* Microbenchmarks

bakery	Mutual exclusion algorithm for arbitrary # of threads
dekker	Mutual exclusion algorithm for two threads
peterson	Mutual exclusion algorithm for arbitrary # of threads
lazylist	Concurrent list algorithm using bakery lock
ms2	Concurrent queue algorithm using bakery lock

6 Evaluation Setup

6.1 Simulation

We implemented Sync-Order in the MARSS [26] simulator targeting x86 architecture with the TSO model. MARSS is a cycle accurate full-system multicore simulator which can run multi-threaded applications on top of Linux. We model a multicore architecture connected with crossbar switch using MOESI coherence protocol using a distributed directory controller. As the simulation time significantly increases for a 16- and 32-processor setting, we only model up to 8 processors as prior work [9]. Each processor has its private L1 and L2 cache and shares the L3 cache. Table 3 shows the detailed specifications. The message delays of sync-loads and sync-stores are in one way direction, which means that the round trip time should double. Since the latency of fences are affected by many factors in reality, we do not accurately simulate the fence latency but use the default mechanism in the simulator.

We evaluate our design by comparing two multicore architectures: the baseline architecture using conventional fences and Sync-Order with a distributed *PSC.*. The baseline architecture does not implement post-fence load speculation and MARSS simulates the effect of store buffer via memory request queueing mechanism at the different controllers (e.g., CPU controller, cache controller).

A *PST* entry is 67-bit and a *WLT* entry has 136 bits for our simulated 8-processor architecture. Assuming 48 store buffer entries for each processor, each *PSC.* module requires about 9.5 KB storage for the two tables. As a result, the total storage for our distributed controller is 76 KB with eight modules.

6.2 Methodology

We measured two types of benchmarks: five microbenchmarks of synchronization constructs from WeeFence [9] (shown in Table 4) and several multi-threaded application benchmarks from PARSEC and SPLASH-2. Microbenchmarks frequently use mutual exclusion primitives for synchronization while application benchmarks typically

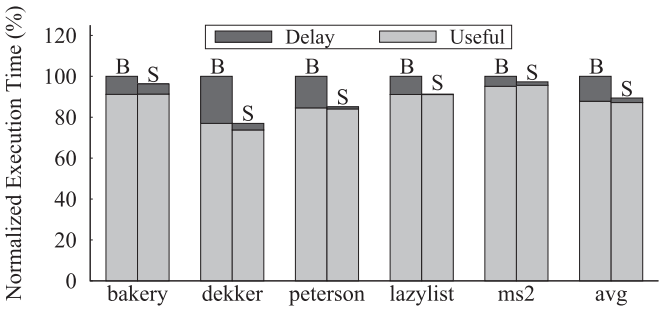


Fig. 13. Performance impact on kernels. B and S stand for *baseline* and *Sync-Order*.

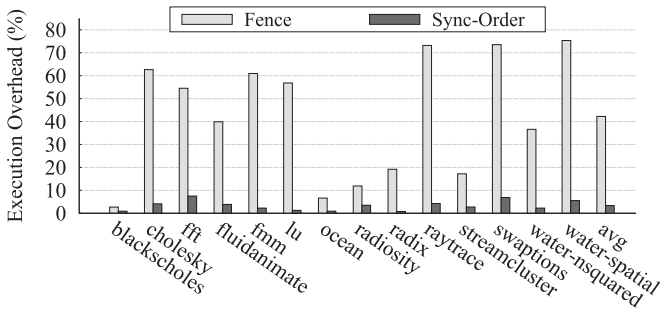


Fig. 15. Execution overhead of over-annotation with fences or sync-vars.

manifest different synchronization patterns and react differently to fence optimization. We selected some benchmarks from PARSEC because others require unacceptably long time to finish. We use `simdev` input for PARSEC programs.

To rule out the influence of fence semantics from atomic RMWs, we replace the spin locks used in those applications with bakery locks which do not apply atomic RMWs. For the baseline version, the bakery lock uses `mfence` instructions to prevent reorderings; the Sync-Order version enforces sequential consistency among *sync-var*s in bakery locks.

We use Memorax [2] to analyze the specification of the bakery lock which has been translated from C code to RMM modeling language. The result is sets of variables which may cause SCVs. We treat all the variables in the sets as *sync-var*s and place them in a reserved linear address range.

Because sync-var identification is time-consuming, compilers may perform over-annotation and trade the minimality of sync-var sets for fast analysis. To simulate the overhead of over-annotation, we add more fences or annotate more sync-vars, as similarly done in prior work [9], [19], [20], we annotate all possibly shared variables as sync-vars. For baseline, we also add a fence after each store operation to such possibly shared variables. The number of fences for baseline applications is almost the same as the number of sync-store and sync-load in Sync-Order.

7 Evaluation

7.1 Performance with Microbenchmarks

Fig. 13 shows the execution time of microbenchmarks with baseline and Sync-Order. For each microbenchmark, the results are normalized to that of baseline and broken down into *delay* and *useful* parts. The delay part is either caused by fences or stalled sync-loads. The rest of execution time is useful cycles not stalled due to ordering enforcement. The figure shows that fences in microbenchmarks induce an average of 12.20 percent slowdown while Sync-Order reduces the overhead to 2.21 percent. The overhead of Sync-Order depends on the actual number of sync-loads that are stalled and sync-stores almost incur no overhead because they are never stalled by sync-loads. The delay for Sync-Order is almost negligible because in most cases shared accesses to sync-vars from multiple processors do not cause cycles and sync-loads do not block. Note that reduced time to acquire a lock can in turn shorten the blocking time for a thread. For example, the execution time for *dekker* is reduced.

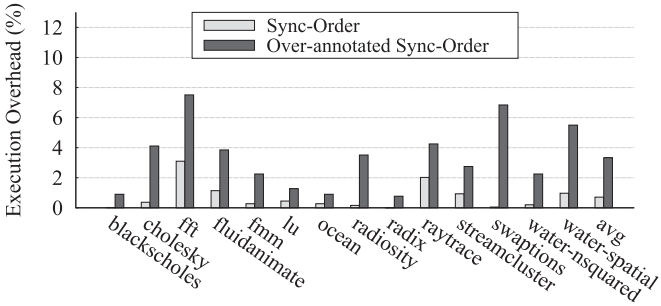


Fig. 14. Execution overhead introduced by over-annotation on Sync-Order.

7.2 Performance with Over-Annotation

Fig. 14 shows the overhead of over-annotation with Sync-Order. The applications without over-annotation have an average of 0.71 percent overhead while the over-annotated versions have 3.33 percent overhead. Over-annotation slightly adds to the overhead of Sync-Order because the unnecessarily annotated sync-ops still incur message overhead. Another source of additional overhead is that a sync-load always has to wait for the response from the controller before it continues to load from cache or gets blocked.

Fig. 15 shows the execution overhead of fence and Sync-Order with over-annotation. The baseline introduces 42.23 percent overhead on average. Some applications have higher fence overhead because they have more shared variables after which fences are inserted conservatively. The average overhead for Sync-Order is 3.33 percent. The overhead is lower than microbenchmarks because of longer critical sections and more application logic. And cycles are less likely to happen in real applications than microbenchmarks.

7.3 Impact of PSC Latency

We evaluate the influence of varying latencies of the *PSC*. with several applications and compare the result with baseline. Fig. 16 shows the overhead with different *PSC*. latencies. The fence mechanism has constant overhead for each program and appears as a horizontal line since we are only modifying the latency of *PSC*.. Sync-Order has increased overhead with increasing latency for *PSC*.. Note that even if accessing the *PSC*. incurs much higher overhead than accessing L3 cache (eight cycles), the overhead of Sync-Order is still much lower than fences. But lower latency indeed makes Sync-Order more efficient.

7.4 Impact of L3 Cache Latency

We evaluate the performance of Sync-Order with different L3 cache latencies using microbenchmarks and the result in

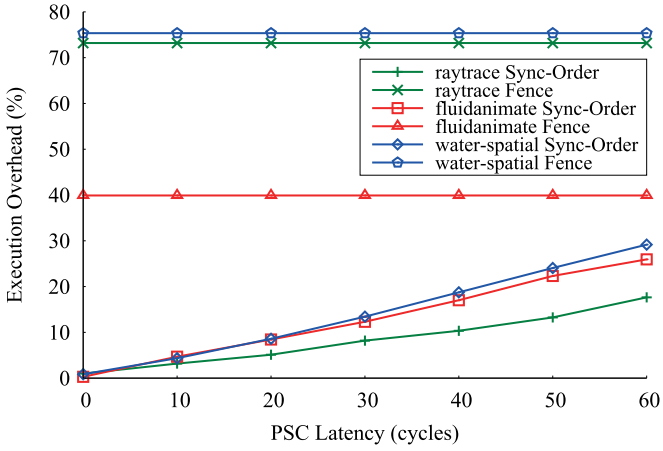


Fig. 16. Execution overhead impact of PSC latency.

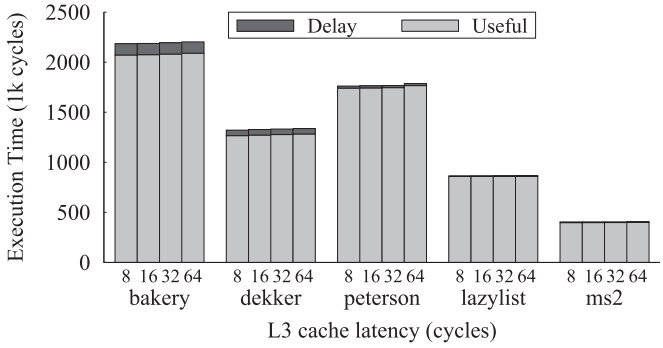


Fig. 17. Performance impact of L3 cache latency.

Fig. 17 shows that the execution time difference is negligible. Because Sync-Order mainly handles cache transactions between L2 caches, L3 latency has little influence.

7.5 Characterizing Sync-Order

Table 5 gives the detailed characteristics of Sync-Order by CPU statistics and *PSC*. statistics.

CPU statistics Column 3 to 5 show the results of the number of loads in every 1K instructions. Column 3 is the number of all loads including *sync-load*s and normal loads. Column 4 is the number of *sync-load*s and column 5 shows loads actually delayed due to conflicts. Generally, microbenchmarks have more frequent *sync-load*s and delayed *sync-load*s than multi-threaded applications because their critical sections are shorter and contention is higher. Over-annotated benchmarks have more manually added *sync-load*s than their original versions but the delayed cycles do not increase a lot. Column 4 and 5 show that only a small fraction of *sync-load*s are actually stalled for conflicting stores. This supports our observation that conflicts rarely happen in execution.

Column 6 and 7 show the delayed cycles of all *sync-load*s in column 4. Column 6 is the average cycles delayed with every *sync-load* and column 7 gives the maximum. The results show that the amortized cost of *sync-load* delay is less than 5 cycles, but the maximum can be high. This is caused by occasional overlapping of conflicting accesses that results in multiple pending requests in the *PSC*.. A *sync-load* is delayed for more cycles when there are multiple conflicting *sync-store*s pending on remote processors, as it must wait for all to finish.

Column 8 and 9 present the number of store instructions of every 1K instructions. Column 8 shows the number of all stores including *sync-store* and normal stores. Column 9 gives the number of *sync-store*. In application groups, *sync-load*s are more than *sync-store*s but only a few are delayed. This is expected as most reorderings can be allowed.

PSC statistics Column 10 to 13 characterize *PSC* by the number of *sync-load* and and *sync-store* records in the tables

TABLE 5
Characterization of Sync-Order

Groups	Benchmarks	CPU statistics/1K inst							PS controller statistics				Traffic inc (%)
		#ld	sync #ld	stalled #ld	stalled cycles		#st	sync #st	pending st		waiting ld		
					Avg	Max			Avg	Max	Avg	Max	
Synchronization Constructs	bakery	366.70	86.36	14.88	4.47	184	70.40	12.14	0.37	5	0.14	28	21
	dekker	329.98	64.00	19.04	4.67	110	91.06	22.87	0.60	10	0.26	31	19
	peterson	283.80	17.10	2.97	4.47	208	134.56	17.28	0.28	4	0.01	10	6
	lazylist	330.46	13.39	0.14	4.12	165	128.16	6.66	0.16	8	0.01	16	9
	ms2	293.91	35.39	1.19	4.35	181	103.78	6.89	0.09	4	0.03	17	4
Overannotated Apps	*fluidanimate*	184.08	36.10	0.00	4.21	141	46.10	4.95	0.35	15	0.00	23	17
	water-nsquared	174.20	55.46	0.00	4.06	258	59.73	4.36	2.99	64	0.01	69	25
	ocean	203.71	4.40	0.00	0.51	7	112.99	0.10	0.00	3	0.00	0	1
	radiosity	267.73	24.20	0.04	2.02	151	101.69	0.25	0.03	6	0.01	60	7
	blackscholes	165.33	1.70	0.00	4.02	5	87.82	0.00	0.00	0	0.00	0	1
	cholesky	258.41	59.20	0.00	4.05	156	38.51	3.92	1.66	67	0.00	34	21
	fft	180.28	20.79	0.00	4.20	112	83.75	9.40	1.20	68	0.00	9	10
	fmm	130.12	20.99	0.01	4.08	219	25.56	0.11	0.08	17	0.05	63	14
	lu	164.55	5.53	0.00	4.01	162	76.91	0.00	0.00	5	0.00	9	2
	radix	121.54	37.28	0.00	4.05	7	37.64	0.43	0.04	61	0.00	0	24
	raytrace	264.26	24.34	0.00	2.74	73	43.81	0.10	0.01	2	0.00	12	8
	streamcluster	248.43	1.32	0.00	4.88	429	79.67	0.01	0.00	15	0.40	13	0
	swaptions	177.78	15.22	0.00	4.12	7	99.09	1.76	1.70	25	0.00	0	6
	water-spatial	177.43	50.32	0.01	4.03	326	64.36	1.41	0.37	53	0.00	50	21

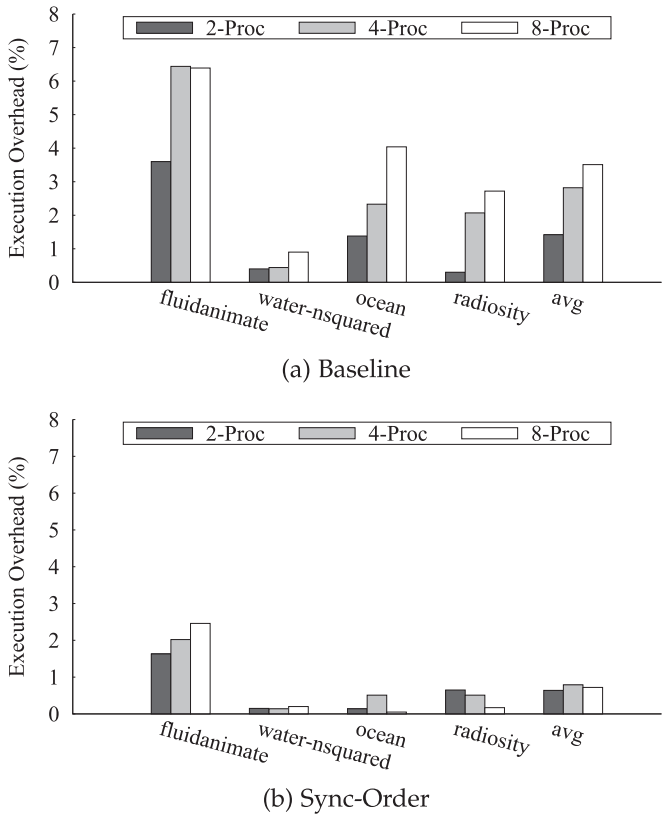


Fig. 18. Scalability of Sync-Order.

of *PSC*. Column 10 and 11 show the averge and maximum number of pending *sync-store*s at any time and the following two give the number of waiting *sync-load*s. The number of records really depends on the frequency of synchronization of applications. But even for microbenchmarks and barrier-intensive applications (e.g., *fluidanimate* and *streamcluster*), the maximum is within 90. This reveals an opportunity to reduce the size of *PST* and *WLT*.

7.6 Scalability

We replace mutexes in parallel application benchmarks with bakery locks with Sync-Order version and `mfence` version respectively and run applications with the number of threads equal to the number of processors. Fig. 18a shows the fence overhead with 2, 4, and 8 processors. The execution overhead of most benchmarks increases a lot with the increasing number of processors. Fig. 18b shows the scalability of Sync-Order which is better.

8 Related Work

End-to-End SC. End-to-End SC proposes a memory access type driven SC hardware design. It distinguishes between private and shared memory accesses. For addresses that are read-only or private, it does not try to enforce SC to optimize but prohibit reordering of shared accesses. The limitation is it does not allow possible safe reordering to shared accesses.

Location-Based Fence. Ladan-Mozes et al. [17] take a different point of view of the fence problem. The paper proposes to guarantee correctness by associating a fence with an address of store. A fence does not stall unless the address is accessed by another fence. However, the implementation is based on the cache state and a conflict causes the executing processors to serialize the execution which is inefficient.

Conflict Ordering. [21] is designed to ensure SC by ordering general data access conflicts under non-SC memory models. This may incur additional overhead for TSO because TSO only allows one kind of ordering relaxation. Conflict Ordering may not achieve optimal performance due to unnecessary enforcement for TSO.

Conditional Fence. [19] observes that only 8 percent of executed fences are really necessary to ensure SC. They propose C-Fence that augments processors to monitor fences on other processors. A C-Fence can allow reordering post-fence loads with pre-fence stores if no other processor is executing C-Fence. Otherwise, all C-Fences fall back to conventional fences. C-Fence is a coarse-grained fence elimination approach and it still induces unnecessary delay.

Conditional Memory Ordering. [30] observes that there are several sources of memory ordering redundancy in fences in the current lock implementation. They propose a new programming model that combines processor ID and synchronization semantics to dynamically reduce the overhead of memory ordering. This approach is relatively specific to the mutual exclusion algorithms and hard to be applied to other scenarios.

WeeFence, AAF and Asymmetric Fences. WeeFence [9] and AAF [20] share some similarities. They track the pending stores, and enforce fences when an SCV is possible. WeeFence extends the interconnect by adding a GRT which manages pending sets consisting of pre-fence pending stores. AAF allows stores to stores to complete out of order implying the directory has the knowledge of all stores that are active and not yet completed. WeeFence and AAF achieves a similar result of collecting pending set of ongoing stores with different approaches.

Asymmetric Memory Fences [8] points out their common limitations. First, collecting pending stores from a distributed directory is subject to consistency issues. Second, both approaches involves great hardware complexity to collect information. Third, handling single fences cases on TSO increases hardware complexity. Asymmetric Memory Fence [8] does not solve the aforementioned problems directly but turns to a hybrid solution instead.

Prwlock. Liu et al. [23] implemented a scalable reader-writer lock with low reader-side latency by removing the fences on the reader fast path. This approach is effective in reducing reader side latency but no current hardware claims to guarantee such bound and thus IPIs are still needed to infer straggler readers.

TBTSO. Morrison et al. [25] proposed an hardware approach called TBTSO to optimizing synchronization primitives based on the flag principle. TBTSO also leverages the observation that the time to drain a store buffer entry is bounded to order the accesses to flags. This approach makes too much assumption about the bound of different components within a processor and thus requires significant design and implementation complexity.

Automatic Fence Insertion. [1], [2], [7], [10], [15], [16], [22], [29] use extensive compiler techniques to automatically insert fences in multi-threaded programs and enforce sequential consistency with low overhead. However, without hardware extension, inserted fences still cannot

dynamically avoid unnecessary delay. Also finding the minimal set of fences within reasonable time remains a challenge. Some approaches either conservatively insert more fences or reduce state space in model checking. But even for conservative approaches, Sync-Order can detect unnecessary fences and skip them.

9 Conclusion

This paper introduced Sync-Order, a mechanism combining simple hardware extension with compiler assistance to enforce the ordering of specific memory operations. Our approach is different from conventional fence mechanism provided by some relaxed memory models in that it allows succeeding memory loads to bypass preceding memory stores to some extent. With the knowledge from compiler analysis, Sync-Order has low overhead when an SCV is possible and does not affect victim instructions. sys can handle multiple processors in a centralized or distributed manner. We implemented Sync-Order in a multiprocessor simulator with TSO model. Our evaluation with synchronization structs shows that Sync-Order can notably reduce the cost of enforcing ordering.

Acknowledgments

This work is supported in part by National Key Research and Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314), National Top-notch Youth Talents Program of China, Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and Singapore NRF (CREATE E2S2). Haibo Chen is the corresponding author.

References

[1] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine, "Automatic fence insertion in integer programs via predicate abstraction," in *Proc. 19th Int. Conf. Static Anal.*, 2012, pp. 164–180.

[2] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine, "Memorax, a precise and sound tool for automatic fence insertion under TSO," in *Proc. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 530–536.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: Performance-transparent memory ordering in conventional multiprocessors," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 233–244.

[4] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," in *Proc. 29th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2008, pp. 68–78.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 278–289.

[6] H. Chen, H. Zhang, R. Liu, B. Zang, and H. Guan, "Fast consensus using bounded staleness for scalable read-mostly synchronization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3485–3500, Dec. 2016.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java," in *Proc. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 1999, pp. 1–19.

[8] Y. Duan, N. Honarmand, and J. Torrellas, "Asymmetric memory fences: Optimizing both performance and implementability," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 531–543.

[9] Y. Duan, A. Muzahid, and J. Torrellas, "WeeFence: Toward making fences free in TSO," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 213–224.

[10] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *Proc. 17th Annu. Int. Conf. Supercomputing*, 2003, pp. 285–294.

[11] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proc. Int. Conf. Parallel Process.*, 1991, pp. 355–364.

[12] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is sc + ilp = rc?" in *Proc. Annu. Int. Symp. Comput. Archit.*, 1999, pp. 162–171.

[13] Intel, "Intel® 64 and ia-32 architectures software developers manual," *Volumn 2: Instruction Set Reference, A-Z*, Intel, Santa Clara, CA, USA, 2015.

[14] Intel, "Intel® 64 and ia-32 architectures software developers manual," *Volumn 3: System Programming Guide*, Intel, Santa Clara, CA, USA, 2015.

[15] M. Kuperstein, M. Vechev, and E. Yahav, "Automatic inference of memory fences," in *Proc. Conf. Formal Methods Comput.-Aided Des.*, 2010, pp. 111–120.

[16] M. Kuperstein, M. Vechev, and E. Yahav, "Partial-coherence abstractions for relaxed memory models," in *Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2011, pp. 187–198.

[17] E. Ladan-Mozes, I. Lee, D. Vyukov, et al., "Location-based memory fences," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 75–84.

[18] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[19] C. Lin, V. Nagarajan, and R. Gupta, "Efficient sequential consistency using conditional fences," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 295–306.

[20] C. Lin, V. Nagarajan, and R. Gupta, "Address-aware fences," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 313–324.

[21] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient sequential consistency via conflict ordering," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 273–286.

[22] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav, "Dynamic synthesis for relaxed memory models," in *Proc. 33rd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2012, pp. 429–440.

[23] R. Liu, H. Zhang, and H. Chen, "Scalable read-mostly synchronization using passive reader-writer locks," in *Proc. USENIX Conf. USENIX Annu. Techn. Conf.*, 2014, pp. 219–230.

[24] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proc. Symp. Principles Program. Languages*, 2005, pp. 378–391.

[25] A. Morrison and Y. Afek, "Temporally bounding tso for fence-free asymmetric synchronization," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 45–58.

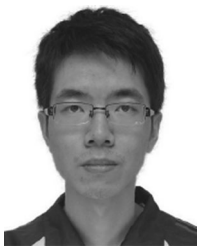
[26] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *Proc. 48th Des. Autom. Conf.*, 2011, pp. 1050–1055.

[27] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[28] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Languages Syst.*, vol. 10, no. 2, pp. 282–312, 1988.

[29] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua, "Compiler techniques for high performance sequentially consistent java programs," in *Proc. 10th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2005, pp. 2–13.

[30] C. Von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu, "Conditional memory ordering," in *Proc. 33rd Annu. Int. Symp. Comput. Archit.*, 2006, pp. 41–52.



Yang Hong received the BS degree in software engineering from Shanghai Jiao Tong University, China, in 2013. He is now working toward the PhD degree at the School of Software, Shanghai Jiao Tong University. His research interests include parallel systems and networked systems.

Yang Zheng is working toward the undergraduate degree at Shanghai Jiao Tong University. He is currently a research assistant in the School of Software, Shanghai Jiao Tong University.

Haibing Guan received the PhD degree from Tongji University, in 1999. He is a professor in the School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University. His research interests include distributed computing, network security, network storage, green IT, and cloud computing.

Binyu Zang received the PhD degree in computer science from Fudan University, in 2000. He is currently a professor in the School of Software, Shanghai Jiao Tong University. His research interests include systems software, compiler design, and implementation.

Haibo Chen received the PhD degree in computer science from Fudan University, in 2009. He is currently a professor in the School of Software, Shanghai Jiao Tong University. His research interests include operating systems and parallel and distributed systems. He is a senior member of the IEEE.