Fence Scoping

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

Vijay Nagarajan
School of Informatics
University of Edinburgh, UK
vijay.nagarajan@ed.ac.uk

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

***Abstract*—We observe that fence instructions used by programmers are usually only intended to order memory accesses within a limited scope. Based on this observation, we propose the concept fence scope which defines the scope within which a fence enforces the order of memory accesses, called scoped fence (S-Fence). S-Fence is a customizable fence, which enables programmers to express ordering demands by specifying the scope of fences when they only want to order part of memory accesses. At runtime, hardware uses the scope information conveyed by programmers to execute fence instructions in a manner that imposes fewer memory ordering constraints than a traditional fence, and hence improves program performance. Our experimental results show that the benefit of S-Fence hinges on the characteristics of applications and hardware parameters. A group of lock-free algorithms achieve peak speedups ranging from 1.13x to 1.34x; while full applications achieve speedups ranging from 1.04x to 1.23x.**

***Keywords*—Memory models, Fence instructions, Scope**

I. Introduction

Modern multiprocessor systems usually adopt shared memory as the primary system level programming abstraction, as it provides all processors with a single view of data and it is relatively easy for programmers to use. However, the reordering of accesses to shared memory may result in unintuitive program behavior. Hence, various memory consistency models [2] have been proposed to specify contracts between programmers and systems (compiler and hardware). Each of these models involves a balance between the ease of programming and high program performance. To achieve high performance, many manufacturers typically implement relaxed memory consistency models by allowing parts of memory accesses to be reordered, e.g., total store order (TSO), relaxed memory order (RMO), release consistency (RC), etc [2]. On the other hand, to prevent reordering of memory accesses that can otherwise be reordered under the supported memory consistency model, systems also provide *fence instruction* (also known as memory barrier) to constrain the reordering. This is important for the correctness of multithreaded programs running under relaxed memory models and are used to ensure that the program execution is consistent with the programmer's intention.

For shared memory programming, although locks remain the *de facto* mechanism for concurrency control on shared-memory data structures, it is not easy to design scalable algorithms based on locks, as they introduce problems such as priority inversion, deadlock and convoying [39]. Therefore, non-blocking algorithms [6], [21] have been developed to avoid these problems by eschewing mutual exclusion, and improve scalability and robustness while still ensuring safety. For example, non-blocking work-stealing [10] is a popular approach for balancing load in parallel programs. Cilk [16] bases on work-stealing to support load balancing of fully strict computations. X10 [11] extends the the work-stealing approach in Cilk to support terminally strict computations. Other commonly used concurrent algorithms include non-blocking concurrent queue [33] which is implemented in Java class ConcurrentLinkedQueue, Lamport queue [28], Harris's set [20], etc.

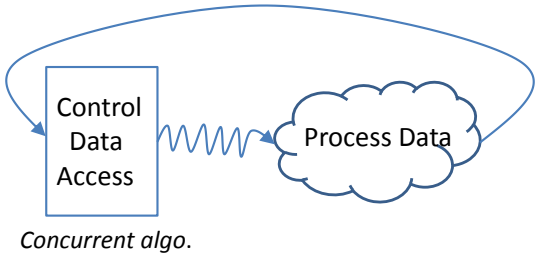

Fig. 1. A common data access pattern.

In these non-blocking algorithms, fence instructions and atomic instructions are required to ensure correctness when they are executed under relaxed memory models. However, traditional fence instructions order memory accesses without being aware of the programmer's intention. In practice, programmers typically use fence instructions to ensure ordering of specific memory operations, while others do not have to be ordered. In other words, the effect of fences is supposed to be limited in a certain scope. Programs using concurrent algorithms usually exhibit the pattern shown in Figure 1. Such programs repeatedly access shared data controlled by concurrent algorithms and process the accessed data. The fences in the concurrent algorithms are supposed to guarantee the correct concurrent accesses to shared data, without being aware of how the accessed data is processed later. However, due to their semantics, traditional fences also order memory accesses which belong to the part that processes the data. That is, if long latency memory accesses are encountered during processing of data, the fences in the concurrent algorithms have to wait for them to complete, incurring unnecessary stalling at fences. To prevent this, we need mechanisms to differentiate memory accesses that must be ordered by a fence from the rest of memory accesses.

Inspired by the above observation, we propose the concept *fence scope* which constrains the effect of fences in programs. We call such a fence as *scoped fence*, S-Fence for short. S-

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

```
1  void put(TASK task){
2    tail = TAIL;
3    wsq[tail] = task;
4    FENCE //storestore
5    TAIL = tail + 1;
6  }

7  TASK take(){
8    tail = TAIL −1;
9    TAIL = tail ;
10   FENCE //storeload
11   head = HEAD;
12   if ( tail < head){
13     TAIL = head;
14     return EMPTY;
15   }
16   task = wsq[tail ];
17   if ( tail > head)
18     return task;
19   TAIL = head + 1;
20   if (!CAS(&HEAD,
21            head,head+1))
22     return EMPTY;
23   TAIL = tail + 1;
24   return task;
25 }

26 TASK steal(){
27   head = HEAD;
28   tail = TAIL;
29   if (head ≥ tail)
30     return EMPTY;
31   task = wsq[head];
32   if (!CAS(&HEAD,
33            head,head+1))
34     return ABORT;
35   return task;
36 }
```

Fig. 2. Simplified Chase-Lev work-stealing queue [10].

Fence is a customizable fence which only orders memory accesses in its scope, without being aware of memory accesses beyond the scope. In practice, programmers can use such fences when they only intend to order part of memory accesses, but not all of them, e.g., the concurrent algorithms mentioned above. S-Fence enables programmers to specify the scope of fences using customizable fence instructions. The scope information is encoded into binaries and conveyed to hardware. At runtime, hardware utilizes the scope information to determine whether a fence needs to stall due to uncompleted memory accesses in the scope. The key contributions of this work are:

1) To the best of our knowledge, this is the first work that explores the scope in which fences impose ordering of memory accesses in hardware. We propose *fence scope* and a new customizable fence statement *scoped fence* (S-Fence). S-Fence is easy for programmers to use, and enables programmers to convey ordering demands more precisely to the hardware for performance improvement.
2) We propose a possible implementation of fence scoping with current object-oriented programming languages. The idea of fence scoping is consistent with the principle of encapsulation and modularity of object-oriented languages. This makes it easy to incorporate fence scoping to current popular object-oriented languages.
3) We describe the hardware design and compiler support for S-Fence. It only requires minor modification to the current hardware, and the compiler support is also straightforward. This makes scoped fence a practical solution for refining the traditional fence to improve performance.
4) We conduct experiments on a group of lock-free algorithms and the other group of full applications. The results show that, the benefit of S-Fence hinges on the characteristics of applications and hardware parameters. Lock-free algorithms achieve peak speedups ranging from 1.13x to 1.34x; while full applications achieve speedups ranging from 1.04x to 1.23x.

The rest of the paper is organized as follows. Section II presents the background on fence instructions and a motivating example. Section III proposes the concept *fence scope* and the design of *scoped fence* (S-Fence) statements. Section IV and Section V describe the two types of fence scopes and their compiler and hardware support. The experimental results are presented in Section VI. We discuss related work in Section VII and conclude in Section VIII.

II. Background and Motivation

A. Fence instructions

Modern multiprocessors implement relaxed memory consistency models for achieving high performance. These systems provide *fence instructions* as a mechanism for selectively overriding their default relaxed memory access order [2], [14]. A fence instruction ensures that all memory operations prior to it have completed before the memory operations following it are performed. Commercial architectures provide various fence instructions, enforcing different memory orders, e.g., *lfence*, *sfence* and *mfence* in Intel IA-32, customizable MEMBAR instruction in SPARC V9, etc. In addition, atomic instructions usually imply the same effect as fence instructions.

Fence and atomic instructions are important for enforcing correctness of programs when they are executed on machines that only support relaxed memory models. For example, implementations of many lock-free concurrent algorithms have to use fence and atomic instructions. [3], [22] have proven that, under relaxed memory models, the use of fences or atomic instructions is inevitable for building concurrent implementations of sets, queues, stacks, mutual exclusion, etc. However, fence instructions are substantially slower than regular instructions. The use of fence instructions can incur high overhead. For example, Frigo *et al.* [16] observe that in an Intel multiprocessor, Cilk-5's THE protocol spends half of its time executing a memory fence. Hence, reducing fence overhead is beneficial for the program performance.

B. Work-stealing queue: A motivating example

Figure 2 shows a simplified C-like pseudo code of Chase-Lev work-stealing queue [10]. Work-stealing is a popular method for balancing load in parallel programs. Chase-Lev work-stealing queue implements a lock-free dequeue using a growable cyclic array, which has three operations: put, take and steal as shown in Figure 2. In the code, HEAD and TAIL are two global shared variables which record the head and tail indices of the valid tasks in the cyclic array wsq. The owner thread can put and take a task on the tail of the queue, while other thief threads can steal a task on the head of the queue. Under sequential consistency, the algorithm will execute correctly, complying with the semantics, i.e., each inserted task is eventually extracted exactly once, either by the owner thread or other thief threads. However, under relaxed memory models, to guarantee the correctness of the algorithm, fences have to be inserted to enforce the ordering of some

memory accesses [25], [32]. Under TSO, a store-load fence in Line 10 is required to guarantee that no task is fetched by two threads; while under PSO, one more store-store fence in Line 4 is required to guarantee steal does not return a phantom task [32]. In addition, there is need for two compare-and-swap instructions: at Line 20 and 32.

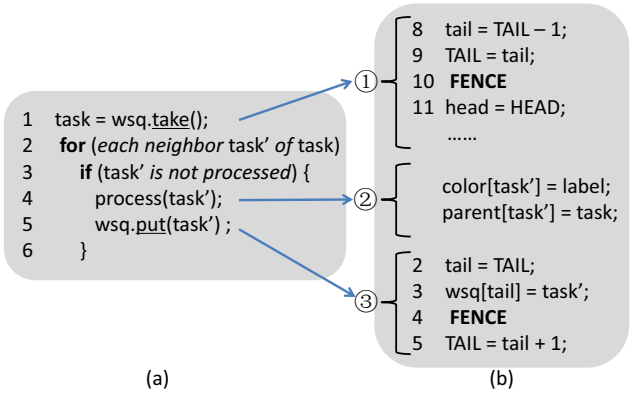

Fig. 3. Parallel spanning tree algorithm.

Let us consider an application of work-stealing queue – parallel spanning tree algorithm, an important building block for many graph algorithms [5], [34]. The parallel spanning tree algorithm is discussed in [5], which uses work-stealing queue to ensure load balancing because of the irregular nature of graph applications. Here, we focus on how the work-stealing queue is used. Figure 3(a) shows the core operations of the algorithm that include calls to work-stealing queue functions take and put. First a *task* is extracted from the work-stealing queue (Line 1), then each unprocessed neighbor *task'* of *task* is processed (Line 4), and *task'* is put into the work-stealing queue (Line 5). Let us further expand these operations as shown in Figure 3(b) (three blocks in (b) correspond to three operations in (a)). We can see that there are two fences that are executed. Consider the fence residing in put (Line 4). The traditional fence semantics will require all its preceding memory accesses to complete before the following accesses can execute, including those in the blocks ①, ② and ③. The problem here is that, since graph applications usually do not exhibit data locality, accessing neighbors of a node may incur long latency cache misses. Thus, stores to arrays *color* and *parent* in ② can be long latency memory accesses, which leads to a long stall time for the fence in ③, even though accesses in Line 2 and 3 can complete quickly. Moreover, these operations are inside a loop, which imposes a significant impact on the whole application performance. However, such stalls are not necessary – the application runs correctly even if the fence does not wait for memory accesses in ② to complete. This is because: (1) fences in the work-stealing queue algorithm are only supposed to order memory accesses inside the algorithm (e.g., in function put, the fence in Line 4 orders the stores in Lines 3 and 5) – the implementers guarantee the correctness of the algorithm without being aware of memory accesses beyond the algorithm; and (2) the parallel spanning tree algorithm does not rely on the fences inside the work-stealing queue algorithm – the users call the functions put and take, but they ensure the correctness of their applications on their own (e.g., Line 3 in (a) tests whether a task is processed). In fact, the correctness of parallel spanning tree algorithm is provable with some ordering requirements under relaxed consistency models [5], which we do not discuss here.

The above example shows how people program and ensure correctness of their programs. The semantics of traditional fence is too restrictive in that it orders all memory accesses without differentiating them; thus, causing unnecessary stalls. If a fence could differentiate the memory accesses it has to order from the other accesses, there will be opportunities to eliminate stalls while still enforcing program correctness. Consider the memory accesses in ③ in Figure 3(b). Since the queue is only occasionally accessed by thief threads, the array wsq and shared variables HEAD and TAIL often reside in the processor's cache, as long as they are not kicked out by conflicting cache lines. This indicates that memory accesses in Lines 3 can often complete quickly. If the fence in Line 4 only needs to order data accesses related to work-stealing queue, without waiting for accesses in ② to complete, the stall time due to the fence can be greatly reduced. The same also applies to the fence in ①. Thus, we introduce the concept of *scope* for fences.

III. Scoped Fence

In this section, we propose *scoped fence*, S-Fence for short, that constrains the effect of a fence to a limited scope. We first define the semantics of S-Fence, then introduce the scope of a fence and how programmers can specify the scope.

A. Semantics of S-Fence

S-Fence can be considered as a refinement of traditional fence as it imposes more accurate constraints on memory ordering. Recall that, the semantics of traditional fence requires that all memory accesses preceding the fence must complete before the memory accesses following the fence are issued. However, S-Fence further limits the scope of the fence. We adopt the following definition of S-Fence throughout the rest of this paper.

> **S-Fence** A S-Fence imposes ordering between memory accesses in such a way that when a S-Fence is executed by a processor, all previous memory accesses *in the scope* of the fence are guaranteed to have completed before any memory access that follows the S-Fence in the program is issued.

In other words, if a memory access prior to the fence is not in the scope of the fence, the fence does not need to wait for it to complete. Although S-Fence can also be considered as a finer form of traditional fence, it is different from other finer fences in current commercial architectures [2], such as *mfence*, *lfence*, and *sfence* in Intel IA-32 and customizable MEMBAR instruction in SPARC V9. The existing finer fences explore the ordering of previous load/store operations with respect to future load/store operations, while S-Fence explores the ordering of a subset of memory accesses that are in the scope of a fence.

B. Scope of a fence

The *scope* of a fence defines the context in which memory accesses should be ordered by the fence. We are all aware of various scoping rules for variables. For example, function

scope is a commonly-used scope, where a locally defined variable is only valid within the function; block scope is a finer-grained scope, where a variable is made local to a block of statements. Programming languages also offer various constructs for controlling scope. Object-oriented languages, e.g., C++ and Java, use `class` to group data and functions that manipulate the data.

We will make use of `class` in object-oriented programming languages to illustrate the concept of fence scoping. However, in this work, we do not target any specific language, but focus on exploring benefits of fence scoping. We would like to provide means for fence scoping that capture its main characteristics and are easy for programmers to understand and use. Without loss of generality, we offer two types of fence scoping, i.e., *class scope* and *set scope*. We also provide programming support that allows programmers to specify the fence scope they want to use, as shown in Figure 4. There are three fence statements customized with parameters, which define the scopes. The specified scope information will be utilized by the compiler and conveyed to the hardware. The first statement has no parameter. It simply represents a traditional fence, which has a *global scope*. In the following sections, we focus on *class scope* and *set scope*, as well as corresponding programming, compiler, and hardware support. In particular, *class scope* makes use of programming language constructs to specify the fence scope; while *set scope* provides a way for programmers to specify fence scope more accurately.

1. **S-FENCE** [global scope]
2. **S-FENCE**[class] [class scope]
3. **S-FENCE**[set, {*var1, var2, ...*}] [set scope]

Fig. 4. Customized fence statements.

Memory consistency models. Note that, the concept of S-Fence does not assume a specific memory model. Fences are still put into programs according to the underlying memory models. The difference of S-Fence is that it further allows to specify the scope of each fence, and such information is conveyed to hardware to order memory operations more accurately. Therefore, fence scoping is orthogonal to memory models, although in our evaluation we consider RMO memory model.

IV. Class Scope

The fence statement **S-FENCE**[class] is used to specify that the fence has a *class scope*. The intuition of class scope is that, since function members of the class operate on data members of the class, fences in function members only have to order memory accesses to the data inside the class – they do not have to order those outside the class. In other words, class scope contains all memory accesses to the data members of the class; if the class has a data member of another class (say A), then class scope also contains memory accesses to the data members of class A and so on recursively.

More formally, Figure 5 shows the semantics of fences with class scope. The semantics only focuses on the memory operations and fence operations. We denote the set of all memory operations by *MemOp*, and the set of all method members by *F*. For each $f \in F$, *C(f)* denotes the class which defines this method f. Moreover, $Seq(F)$ denotes the set of all finite sequences over F, $s \cdot t$ denotes the concatenation of two sequences s and t, and $[\![s]\!]$ denotes the set of all distinct elements in the sequence s. The semantics is presented in an operational style with a set of inference rules. We use the following semantic domains.

- $FSeq \in Seq(F)$, which is used for recording nested method invocation.
- $Scope \in Class \mapsto \mathbb{P}(MemOp)$. Each class forms a scope for the fences used in the class, and the class is associated with a set of memory operations that have to be ordered by these fences.
- $pc \in PC$, which is the program counter. We use $next(pc)$ to denote the instruction following pc.

The formulation in Figure 5 focuses on the effects in processors, but omits the effects in memory subsystem, which depends on the underlying memory models. These inference rules are applied to a single process. They define the state transition from $\langle FSeq \times Scope \times pc \rangle$ to $\langle FSeq' \times Scope' \times pc' \rangle$. Components not updated in the rules are assumed to be unchanged.

$$\textsc{ScopeEnt}\ \frac{stmt(pc) = enter\_md\ f \quad FSeq = s}{FSeq' = s \cdot f \quad pc' = next(pc)}$$

$$\textsc{ScopeEx}\ \frac{stmt(pc) = exit\_md\ f \quad FSeq = s \cdot f}{FSeq' = s \quad pc' = next(pc)}$$

$$\textsc{MemOp}\ \frac{stmt(pc) = mop \quad FSeq = s}{\forall f \in [\![s]\!], Scope(C(f)) = Scope(C(f)) \cup \{mop\} \quad pc' = next(pc)}$$

$$\textsc{Fence}\ \frac{stmt(pc) = fence \quad FSeq = s \cdot f \quad Scope(C(f)) = \emptyset}{pc' = next(pc)}$$

Fig. 5. Semantics of class scope.

The first two rules [ScopeEnt] and [ScopeEx] show the operations at the entrance and exit of a method containing fences. The rule [MemOp] shows that, when a memory operation *mop* is encountered, it is added to its corresponding scopes, which may include multiple nested scopes. We omit the rules for removing memory operations from scopes when they are completed, as this is done by the memory subsystem, which can implement different memory models. The rule [Fence] shows that a fence can complete only when all memory operations *in the corresponding scope* have completed, indicated by *Scope(C(f))*=$\emptyset$.

Figure 6 shows an example of class scope. Suppose fences at Lines 6 and 16 have class scope. Consider the memory accesses to m1 and m2 in the class A, n1 and n2 in the class B. The fence at Line 16 will order the accesses to n1 and n2, as they are in class B; while the fence at Line 6 will order all four memory accesses, as accesses to m1 and m2 are in the class A and n1 and n2 are data members of class B accessed by b.funcB() (Line 5).

Recall the algorithm of Chase-Lev work-stealing queue in Figure 2. Assume those operations are implemented in a class. To only order data accesses related to work-stealing queue, we can apply class scope to the fences by specifying them as **S-FENCE**[class], which forces the fences to only order memory

```
1  class A{
2    B b;
3    int m1, m2;
4    void funcA1(){
5      b.funcB();
6      FENCE
7      m1 = val0;
8    }
9    void funcA2()
10   {m2 = val1;}
11 }

12 class B {
13   int n1, n2;
14   void funcB(){
15     n1 = val2;
16     FENCE
17     n2 = val3;
18   }
19 }
```

Fig. 6. An example of class scope.

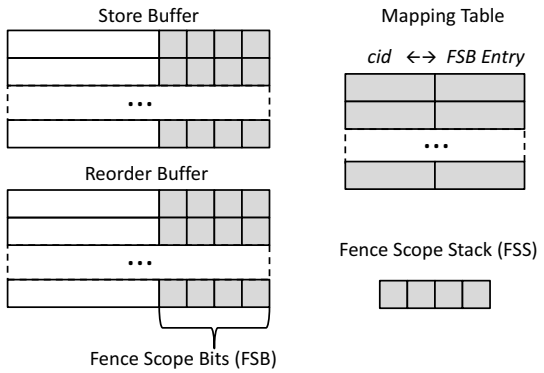

Fig. 7. Hardware support for class scope.

accesses inside the class. Hence, for the parallel spanning tree algorithm in Figure 3, since the memory accesses in ② are out of scope of the fence in Line 4, the fence does not have to wait for accesses in ② to complete.

A. Implementation Design

We say a hardware implementation for S-Fence is consistent with the semantics of S-Fence if it guarantees that any execution in the hardware does not violate its semantics. Obviously, the naive implementation is to consider S-Fence as full fence, stalling the pipeline if there is any memory access not complete prior to the fence. However, to take advantage of S-Fence, the hardware should be able to flag whether a memory access is in the scope of a given fence. Hence, the main hardware support for S-Fence is in form of additional bits, called *fence scope bits*, that are associated with each entry of the reorder buffer (ROB) and store buffer.

1) Compiler support: To convey the scope information to hardware, compiler has to incorporate it into binaries. *We assume that the compiler does not reorder memory accesses across any fence.* For class scope, we only need the extension of Instruction Set Architecture (ISA) shown in Table I.

New fence inst.	class-fence
Supporting inst.	fs_start, fs_end

TABLE I. THE EXTENSION OF ISA FOR CLASS SCOPE.

First, we use a new instruction *class-fence* to represent a fence with class scope. Second, for class scope, we have to convey to hardware: (1) whether a memory access is in a class scope; and (2) which scope a memory access belongs to. To do this, we assign a unique ID to a class if it contains class-scope fences in any of its function members, called *cid*. In the generated binary, *cid* is incorporated into function members of the class. In particular, we introduce two instructions fs_start (start of a fence scope) and fs_end (end of a fence scope) with *cid* as their operand to embrace each function. For each *public* function, we insert fs_start at the entry of the function, and insert fs_end for each exit. Note that, there might be multiple exits for a function. At runtime, they behave as a *nop* operation other than informing ROB to set bits properly. For the remainder of the program, no extra work is done by the compiler, e.g., they are compiled as using traditional compilers.

2) Hardware support: Figure 7 shows the hardware support for class scope in an out-of-order processor core with a ROB and a store buffer. All instructions are retired from the head of ROB in program order. At the head of ROB, loads are retired when they complete, while stores are retired to the store buffer as soon as the value and destination address are available. To support class scope, each ROB and store buffer entry is extended with the *fence scope bits* (FSB) as shown in Figure 7, to flag whether a memory operation is in the scope of some fence. Besides, we use an auxiliary *mapping table* to maintain the mapping from *cid* to FSB entry, and a *fence scope stack* (FSS) to handle nested scopes properly. The key step at runtime is to properly set the bits in FSB and check if a fence has to stall the processor when it is encountered.

3) Setting fence scope bits: Each entry in FSB represents a distinct scope. Memory accesses that belong to the same scope of a fence set the same entry of FSB. Each set bit is cleared when the corresponding memory access has completed. Note that, at a given point in execution, the number of active fence scopes can be quite large; thus possibly exceeding the limited number of entries allowed by FSB. We must deal with this situation in the hardware implementation.

```
1  fs_start cid:
2  //operations in mapping table
3  if (cid recorded in mapping table)
4    current_entry = map[cid];
5  else
6    current_entry = a new entry available
7      in FSB;
8    map[cid] = current_entry;
9  //operations in FSS
10 FSS.push(current_entry);
11
12 fs_end cid:
13 //operations in FSS
14 FSS.pop();
```

Fig. 8. Micro-operations on fs_start and fs_end.

The compiler-inserted instructions fs_start and fs_end are utilized to flag memory accesses in the class scope of a fence. Figure 8 shows the micro-operations on fs_start and fs_end. (1) When the processor issues a fs_start, it indicates the start of a scope, and the operand is the *cid* of the scope. The mapping table is first looked up to see if an FSB entry has been assigned to this scope. If not, a new available entry is used to flag this scope, and the mapping information is added to the mapping table. Moreover, FSS is updated by pushing the current FSB entry. Note that, FSS records the nested active

scopes, where the outermost scope is at the bottom of the stack while the innermost scope is on the top of the stack. Hence, the current scope in which instructions are being decoded is on the top of the stack. The entries recorded in FSS determine which FSB entries have to be flagged. When FSS is not empty, a newly issued memory operation sets all FSB entries that are contained in FSS. By doing this, when an inner scope is flagged for an instruction, all of its outer scopes are also flagged. This is for the ease of next step for fence issue (recall that a class-fence also has to order memory accesses in its inner scope), as well as removing mapping information when an entry is no longer used for a scope. FSS does not change as long as the processor does not encounter a fs_start or fs_end, and hence the processor continues flagging memory accesses in the same FSB entries. (2) When the processor issues a fs_end, it indicates the end of current decoded scope, and the top of FSS is popped. (3) As for the *mapping table*, the mapping information is maintained as long as the corresponding scope is active. When bits in the same entry for all FSBs have been cleared, the processor looks up the mapping table and invalidates the mapping information with such FSB entry.

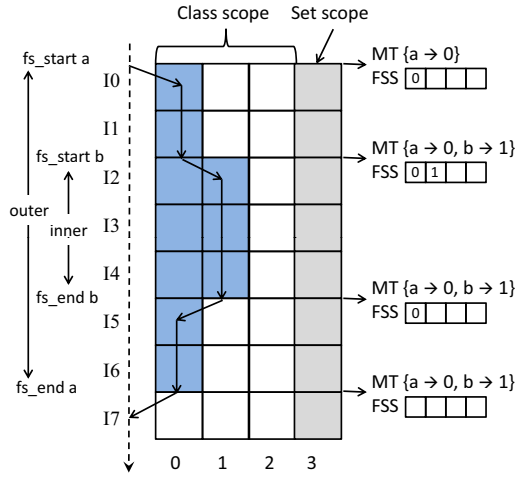

Fig. 9. Setting fence scope bits.

Figure 9 illustrates how we flag memory accesses for class scope. Here, we only show FSBs for ROB – each row corresponds to the FSB of a ROB entry. For simplicity, we only show memory operations, which are decoded in program order and allocated in the entries of ROB. Suppose there is a fs_start before Instructions 0 and 2, and a fs_end before Instructions 5 and 7. Recall that fs_start and fs_end should appear in pairs at runtime, and each pair embraces the instructions in its scope. Hence, we have two scopes here and they are nested, the inner one and the outer one. In the figure, the right side shows how the states of mapping table (MT) and FSS change as instructions are decoded. The line with arrow crosses the entries for current scope, and the highlighted entries in Columns 0-1 will be set to flag the memory accesses that are in the class scope of a fence. Initially, no memory access is flagged, and both the mapping table and FSS are empty. Since a fs_start with *cid* a is encountered before Instruction 0, the processor starts to use Entry 0 to flag the following memory accesses. The mapping $a \rightarrow 0$ is added to the mapping table, and Entry 0 is pushed to FSS. Before Instruction 2, another fs_start with *cid* b is encountered. The processor uses a new entry (Entry 1) for the inner scope, and the mapping table and FSS are also updated accordingly. Now, FSS contains two entries (0 and 1). For the following memory accesses, both Entry 0 and 1 are set, as Entry 1 represents the current scope and Entry 0 represents the outer scope. Since there is a fs_end before Instruction 5, the top of FSS is popped, with only Entry 0 remaining in FSS. However, the mapping table remains the same, as a mapping is only removed when all memory accesses in the corresponding entry have completed. Likewise, fs_end before Instruction 7 indicates the end of the outer scope. FSS is emptied, and hence no memory access is flagged afterwards.

Handling excessive scopes. There can be multiple simultaneously active fence scopes. Moreover, at a given point in execution, the number of active fence scopes may be too large for FSB to assign a different entry to each scope, i.e., FSB does not have enough entries. If the number of active scopes does exceed the number of FSB entries, for each newly encountered scope, we simply choose one specific FSB entry to flag memory accesses. The mapping table and FSS are updated in the same way. The difference is, in the mapping table, multiple fence scopes can be mapped to the same entry now. Such implementation is still consistent with the semantics of S-Fence, as it only places stricter constraints on memory ordering due to fences. However, it is unlikely that a program will involve too many simultaneously active fence scopes. Thus, we only need to maintain a small number of FSB entries in the hardware, and it almost does not affect program performance.

In some rare cases, the mapping table or FSS can be full. That is, when we encounter a fs_start instruction and there is no space for mapping table or FSS to add a new entry. To handle this, we maintain a counter to indicate how many additional scopes are encountered after mapping table or FSS is full. The counter increases by 1 with fs_start, and decreases by 1 with fs_end. During the period when the counter is not zero, each encountered fence will behave as a traditional fence, which orders all memory operations. After the counter becomes zero, the processor switches back to its normal state.

Handling branch prediction. Branch prediction is widely employed in today's pipelined microprocessors for improving performance. A branch misprediction requires ROB to discard all instructions following the branch instruction and those instructions are fetched and executed again. This process may affect the information recorded in FSS. For example, there is a branch between a pair of fs_start and fs_end. The issue of fs_start will push an entry to FSS. Then, the predicted branch leads to the issue of fs_end, and hence the entry in FSS is popped. Later on, the branch prediction is found to be incorrect, and the following instructions are fetched and executed again. In this case, the processor will issue another fs_end which is also paired with the previous fs_start. However, the entry in FSS has been popped because of the previous fs_end, which results in a problem in FSS. To solve this, we maintain a *shadow* copy of FSS, namely FSS′. FSS′ has the similar operations as FSS. The difference is that fs_start and fs_end only trigger operations on FSS′ if there is no unconfirmed branch prediction prior to them. Hence, FSS′ maintains the information that is not affected by branch prediction. When there is a branch misprediction, we copy the content in FSS′ back to FSS and start execution as usual.

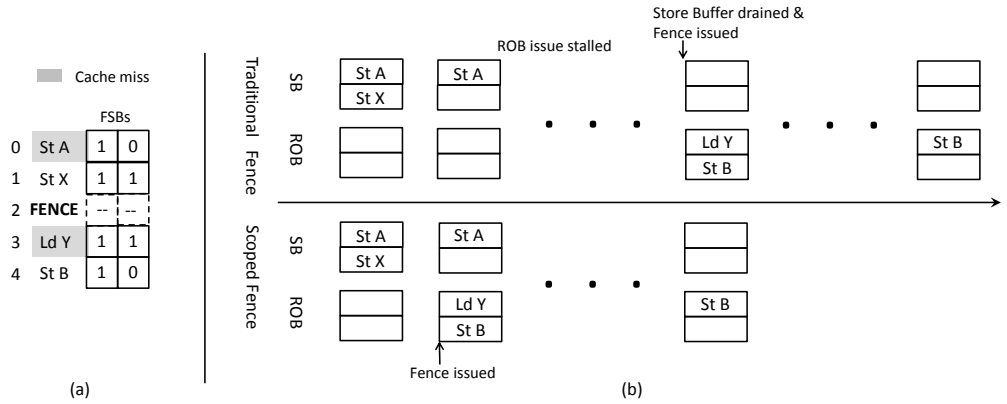

Fig. 10. Comparison between traditional fence and S-Fence.

4) Issuing Fence: After we have set FSB bits properly, it becomes straightforward to determine whether a fence can be issued. When a *class-fence* is encountered, the top of FSS indicates which entry of FSB is flagging the current scope. The processor checks this entry of all FSBs to determine if it is allowed to issue. If no entry is set, the fence is allowed to issue and so are the following instructions; otherwise, the fence is stalled until all entries are cleared.

B. An example

Figure 10 depicts an example to illustrate the performance advantages of S-Fence. Figure 10(a) shows the instructions the processor decodes, where only memory accesses are displayed. Instructions 1 and 3 are in the inner scope as indicated in FSBs, and the fence is a class-fence in the inner scope which only orders Instructions 1 and 3. Moreover, Instructions 0 and 3 are long latency cache misses. Figure 10(b) shows a timeline for executing instructions, in terms of the states of ROB and store buffer. The upper half is for traditional fence, while the lower half is for S-Fence. Initially, St A and St X are retired to the store buffer. St X is a cache hit, so it completes earlier than St A. The subsequent instruction is a fence. With traditional fence, the fence cannot be issued as St A has not completed. Once the store buffer is drained, the fence is issued and so are the following instructions Ld Y and St B. Then Ld Y takes some cycles to load data from memory as it is a cache miss. On the other hand, with S-Fence, since the fence is a scoped fence, it can be issued as soon as St X completes. In this case, Ld Y can be issued and it starts to load data earlier, without waiting for the store buffer to be drained. The total execution time is therefore reduced.

V. Set Scope

Class scope constrains a fence to limit its scope to the object class where it is used. Furthermore, a fence may only intend to order some specific memory accesses. Hence, we also provide a way to specify the fence scope more accurately. The fence statement **S-FENCE**[set, {*var1, var2, ...*}] is used to specify that the fence has a *set scope*, and it only needs to order memory accesses to a certain set of variables {*var1, var2, ...*}.

For example, Figure 11 shows Dekker's algorithm [12], which is designed to allow only one processor to enter the critical section at a time. The purpose of fences (Lines 4 and 10) is to order the accesses to flag0 and flag1. However, traditional fences will also order other memory accesses. In particular, in P0, if there is a long latency memory access to m0 (Line 1) before the store to flag0 (Line 3), the fence will stall its following memory accesses until the store to m0 completes, even when the store to flag0 completes quickly as a cache hit. However, the reordering of the access to m0 across the fence does not violate the programmer's intention, i.e., the exclusive access to the critical section. Hence, we can apply set scope to the fences by specifying them as **S-FENCE**[set, {*flag0, flag1*}], which forces the fences to only order the memory accesses to `flag0` and `flag1`. In this case, even if the store to `m0` (Line 1) is a long latency memory access, the fence (Line 4) does not have to wait for the store to complete. As long as the store to `flag0` (Line 3) has been completed, the fence will allow the following memory accesses to proceed.

	P0		P1
1	m0 = ...	7	m1 = ...
2		8	
3	flag0 = 1	9	flag1 = 1
4	**FENCE**	10	**FENCE**
5	**if** (flag1 == 0)	11	**if** (flag0 == 0)
6	critical_section	12	critical_section

Fig. 11. Simplified Dekker algorithm.

A. Implementation Design

Set scope requires to identify the memory accesses that have to be ordered at runtime. Similar to class scope, this can be easily implemented with compiler and hardware support.

New fence inst.	set-fence
Supporting inst.	*inst. flagging memory operations*

TABLE II. The extension of ISA for set scope.

1) Compiler support: For set scope, we only need the ISA extension shown in Table II. We use a new instruction *set-scope* to represent a fence with set scope. Besides, the ISA is extended to allow a compiler to flag memory accesses to the variables in the set scope. At runtime, when a processor core decodes a memory instruction which is flagged, it will set a scope bit of the allocated ROB entry.

2) Hardware support: Since memory accesses in the set scope of fences have been flagged using the extended ISA, it is straightforward to set FSB bits for these memory accesses when they are decoded and issued. For simplicity, in our design, we do not differentiate memory accesses in set scopes of different fences. Hence, we use a specific FSB entry (e.g., the last entry as shown in Figure 9) to flag if the memory access is in the set scope. By doing this, when the processor encounters a *set-fence*, it checks the last entry of all FSBs to determine whether it can be issued.

B. Class scope vs. Set scope

With set scope, a fence can have a narrower scope compared with class scope. For example, in the work stealing queue (Figure 2), we can either use class scope, or set scope with parameters of shared variables (e.g., HEAD, TAIL, etc). There are trade-offs between using class scope and using set scope. (1) *Compiler*. With class scope, compiler only needs to insert fs_start and fs_end to embrace the functions; with set scope, compiler has to analyze the program to identify the memory accesses to the specified variables, which will involve alias analysis. (2) *Hardware*. Class scope has higher hardware complexity than set scope. Class scope has to set fence scope bits according to the inserted fs_start and fs_end, and handle nested scope properly. However, set scope can set fence scope bits easily according to the flagged memory operations. (3) *Performance*. Since set scope is more accurate on what memory operations to order, it may have better performance than class scope. We will compare the performance in the evaluation.

VI. Experimental Evaluation

The goals of our experimental evaluation are: (1) to assess the performance of S-Fence compared to traditional fences; (2) to understand how applications can benefit from S-Fence, and what characteristics can affect the performance of S-Fence; (3) to study the effect of varying the values of the parameters in the hardware implementation.

Processor	8 core CMP, out-of-order
ROB size	128
L1 Cache	private 32 KB, 4 way, 2-cycle latency
L2 Cache	shared 1 MB, 8 way, 10-cycle latency
Memory	300-cycle latency
# of FSB entries	4
# of FSS entries	4

TABLE III. Architectural Parameters.

Simulation. We implemented S-Fence in the simulator SESC [37] targeting the MIPS architecture. The simulator is a cycle-accurate, execution-driven multi-core simulator with detailed models for the processor and memory systems. We implemented S-Fence by adding FSB, FSS, FSS′ and the associated control logic to the simulator. Currently, scopes for fences in each benchmark program are manually identified, and the scope information is fed to the simulator for runtime usage. Table III shows the default architectural parameters used in all experiments unless explicitly stated otherwise.

Benchmarks. We evaluate our technique using benchmarks in Table IV. In these benchmark programs, fences and atomic compare-and-swap (CAS) instructions are utilized to implement lock-free algorithms. There are two groups of benchmark programs. The first group consists of several lock-free algorithms, i.e., *dekker, wsq, msn* and *harris*. We use these applications to study how program characteristics can affect the performance of S-Fence. Dekker algorithm (*dekker*) [12] is a classic solution to mutual exclusion problems using only shared variables for communication. Chase-Lev work-stealing queue (*wsq*) [10] is a lock-free work-stealing deque implemented with a growable cyclic array. Non-blocking concurrent queue (*msn*) proposed by Michael and Scott [33] is a multiple-producer and multiple-consumer queue. Harris's set (*harris*) [20] is a concurrent set implementation using sorted linked list to represent the set. Since these lock-free data structures are not closed programs, we constructed harnesses to use them to assess the performance of S-Fence. The second group consists of several full applications. We use them to evaluate how they can benefit from S-Fence, and how architecture parameters affect the performance. *pst* and *ptc* are parallel spanning tree algorithm [5] and parallel transitive closure algorithm [15] using work-stealing queue [10]. *barnes* and *radiosity* are from SPLASH-2 [43], and they are inserted with fences to enforce sequential consistency [38].

Benchmarks	Type	Description
dekker	set	Dekker algorithm [12]
wsq	class	Work-stealing queue [10]
msn	class	Non-blocking Queue [33]
harris	class	Harris's set [20]
barnes	set	Barnes-Hut n-body [43]
radiosity	set	Diffuse radiosity method [43]
pst	class	Parallel spanning tree [5]
ptc	class	Parallel transitive closure [15]

TABLE IV. Benchmark Description.

A. Lock-free algorithms

In the lock-free algorithms, fences and atomic instructions are used to ensure correctness when they are executed under relaxed memory models. Fences are inserted as suggested in [8], [25], [32]. We use these applications to have a preliminary understanding on the performance of S-Fence. Moreover, the workload between fences may affect the benefit of S-Fence. Hence, we developed the harness programs that can control the workload. Different workloads have different amounts of computation. In this experiment, the harness program repeatedly 1) accesses shared variables using lock-free algorithms; and 2) performs arithmetic computations on private variables, whose accesses do not need to be ordered by fences. We vary the amount of computations by filling up different amounts of arithmetic computations in the loop, to evaluate the performance of S-Fence. We only measured the execution time of the parallel sections in the programs. Figure 12 shows the speedups of S-Fence over traditional fence, where the x axis represents different amounts of computations, from low to high.

As we can see, S-Fence achieves improvement for all applications, with peak speedups ranging from 1.13x to 1.34x. Moreover, for each application, its speedup varies with different workload. The trend is first increasing before reaching the peak speedup and decreasing afterwards. This is because, with low workload, the fence costs relatively more time to order

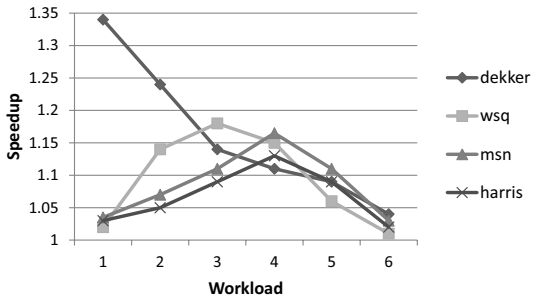

Fig. 12. Impact of workload.

the memory accesses in the scope, in which case S-Fence does not completely manifest its advantage over traditional fence. As the workload increases, it will reach a point at which S-Fence can manifest its advantage the most, and hence the speedup reaches the peak value. When the workload increases further, the time cost by the workload will gradually dominate the overall running time of the program, and the stalls due to fences gradually become insignificant. Hence, the speedup becomes smaller. From the figure, we can also observe that, different benchmarks reach peak speedups with different workload. One reason of this result is that they have different amount of computation in the scope, and hence they need different amount of workload to reach the peak value. In particular, *dekker* reaches the peak value with low workload. Therefore, the speedup of S-Fence over traditional fence depends on the relative cost of the workload. However, S-Fence always performs better than traditional fence.

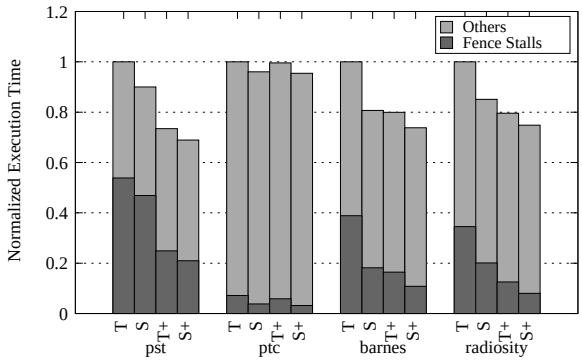

Fig. 13. Normalized execution time (T – traditional fence; S – S-Fence; $T+$ – traditional fence with in-window speculation; $S+$ – S-Fence with in-window speculation) .

B. Performance on full applications

We now evaluate the performance of S-Fence on several full applications. Figure 13 shows the normalized execution time of applications with traditional fence and S-Fence, with and without in-window speculation [18]. Each bar consists of two parts, the stalling time due to fences *Fence stalls* and the rest of the execution time *Others*. All execution time is normalized to the total execution time with traditional fence (the lower, the better). Let us first see their execution time without in-window speculation.

(***pst*** and ***ptc***) We use *pst* (parallel spanning tree [5]) and *ptc* (parallel transitive closure [15]) to evaluate S-Fence with class scope. These two applications are both graph applications and use work-stealing queue to achieve load balancing because of the irregular nature of graph applications. We use S-Fence with class scope in the work-stealing queue implementation, and evaluate their performance. Note that, using S-Fence in these applications does not violate the applications' correctness.

As we can see from Figure 13, in the case of *pst*, traditional fences used in the work-stealing queue incurs stalls accounting for more than 50% of the overall execution time. Using S-Fence reduces 12.9% fence stalls and achieves 1.11x speedup in the overall execution time. We can see that S-Fence does not reduce as many stalls as that in *barnes* and *radiosity*. This is because, in addition to the fences used in the work-stealing queue implementation, another fence is required between the stores to arrays *color* and *parent* (segment ② in Figure 3) under relaxed consistency models. Since S-Fence does not optimize this fence, it is a full fence outside the work-stealing queue implementation. The existence of this full fence limits the optimization space for S-Fence. In the case of *ptc*, we can see that fence stalls only occupy a small percentage of overall execution time, as workload between fences is relatively large. However, S-Fence is still able to reduce around half of fence stalls in *ptc*, and achieves 4.3% improvement in the overalll execution time.

(***barnes*** and ***radiosity***) We use *barnes* and *radiosity* to evaluate S-Fence with set scope. Programs running on machines only supporting relaxed consistency models can be inserted with fences to enforce sequential consistency [38]. This can be done by compilers to identify memory pairs which have to be ordered based on delay set analysis [38]. Hence, the inserted fences are used to order some specific memory accesses, but not all of them. So S-Fence with set scope can be utilized here during compilation, flagging memory operations that have to be ordered with delay set analysis.

As we can see from Figure 13, in the case of *barnes* and *radiosity* with traditional fence, fence stalls account for a significant portion of the total execution time (38.8% and 34.5% respectively). However, S-Fence is able to eliminate 40%-50% fence stalls, and hence reduce the overall execution time by 19.5% and 15.8%. This is because, memory accesses to private or read-only data account for a significant portion of all memory accesses [40], and such memory accesses will not be flagged by S-Fence, as they are not involved in any conflicting accesses in the delay set analysis [38]. Hence, S-Fence only flags a part of memory accesses, and orders them at runtime. In particular, those long latency private memory accesses will not be flagged, and hence they are not ordered by S-Fence. This will help hide long latency memory accesses.

In-window speculation. In-window speculation [18], where speculation on reordering is employed in instruction window, can be used to reduce some of fence stalls. To incorporate in-window speculation into S-Fence, a fence now can be issued speculatively, but before it can be retired from ROB, it has to check the FSBs of store buffer. Figure 13 also shows the performance when in-window speculation is employed. As we can see, with in-window speculation, fence stalls are reduced significantly for both traditional fence and S-Fence. However, S-Fence still achieves performance improvement over traditional fence.

C. Class scope vs. Set scope

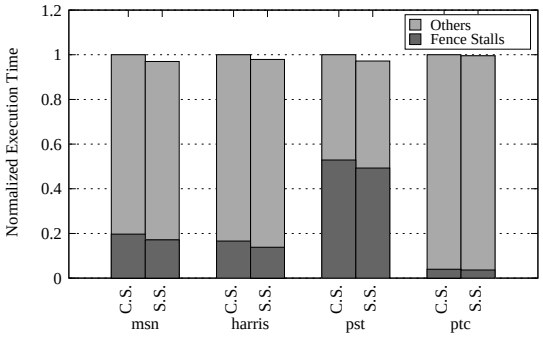

Fig. 14. Performance comparison between class scope and set scope ($C.S.$ – class scope; $S.S.$ – set scope)

We now compare the performance of class scope and set scope. *msn*, *harris*, *pst*, and *ptc* are used for this experiment. They use class scope in previous evaluation, but it is also possible to use set scope by only flagging shared variables that have to be ordered. Figure 14 shows the results. For all benchmarks, performance with set scope is slightly better than that with class scope, as set scope orders fewer memory accesses. However, the difference between them is not significant. This is because fence stalls are not reduced significantly by set scope. Since class scope is easier to use, programmers would be able to choose to use it, instead of set scope, without significant performance loss.

D. Sensitivity Study

In this section, we study how the architecture parameters affect the performance of S-Fence, including memory access latency and reorder buffer size.

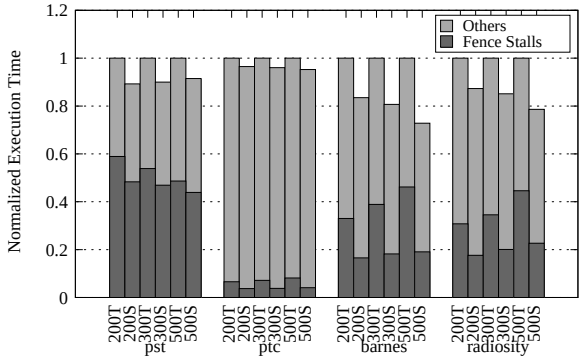

Fig. 15. Varying memory access latency.

Memory access latency. A fence stalls because some memory accesses prior to it has not completed. Long latency memory accesses impose long stalling for fences. In particular, a cache miss takes as much time as the round trip latency to the memory, incurring long stalling to its following fence. To study the impact of memory access latency on S-Fence, we varied the memory access latency with values of 200, 300, and 500 cycles. Figure 15 shows the execution time normalized to the total execution time with traditional fence (the lower, the better). Each cluster shows the results for each benchmark, including traditional fence and S-Fence with different memory access latencies (*xT* and *xS* represent the execution time with *x* cycles latency for traditional fence and S-Fence respectively). As we can see, for *barnes* and *radiosity*, the improvement of S-Fence increases as the latency increases. In particular, larger latency results in larger portion of fence stalls, and S-Fence is able to reduce 40%-50% fence stalls. However, we see a different trend for *pst*. As the latency increases, we do not see the increase in improvement, and the fence stalls account for less portion of the overall execution time. One reason for this is that, the full fence in *pst* outside the work-stealing queue incurs more stalls as the latency increases, and the benefit of S-Fence is offset by such stalls.

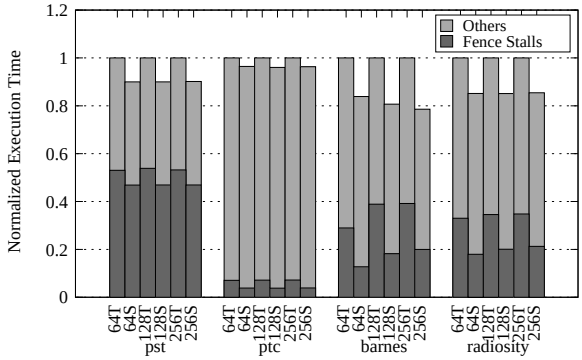

Fig. 16. Varying ROB size.

Reorder buffer size. The reorder buffer (ROB) enables out-of-order instruction execution. S-Fence only stops issuing new instructions into ROB when any memory access in the scope prior to the fence has not completed. When a S-Fence does not have to stall, larger ROB size would allow more instructions following the fence to be issued into ROB. This would increase the improvement of using S-Fence. In Figure 16, we show the impact of ROB size on the performance of S-Fence, where we varied the ROB size with values of 64, 128 and 256 (*xT* and *xS* represent the execution time with *x* entries of ROB for traditional fence and S-Fence respectively). As we can see, for *barnes*, S-Fence achieves better performance when the ROB size increases from 64 to 256. This is because S-Fence used in *barnes* benefits from larger ROB size by allowing more instructions to be issued to ROB when a S-Fence does not have to stall. On the other hand, in the case of *radiosity*, *pst* and *ptc*, the performance of S-Fence remains stable with different ROB sizes. This is because a smaller ROB size already exposes the critical path in these applications, and hence larger ROB size does not result in more overlap of instruction execution. In fact, with 256 entries of ROB, the average number of used ROB entries is less than 80 for *radiosity*, *pst* and *ptc*, which indicates they do not benefit from a larger ROB.

E. Hardware Cost

S-Fence is implemented with low hardware cost. The main hardware change is only extending each entry of ROB and Store Buffer with a few (e.g., 4 bits) fence scope bits (FSB); and the auxiliary structures, mapping table and fence scope stack (FSS), also only cost very small amount of hardware. In the case of 128-entry ROB and 8-entry Store Buffer, the hardware overhead would be less than 80 bytes for each core. More importantly, all these changes are made locally in each processor core, without adding inter-processor communication for multiprocessors, e.g., affecting cache coherence.

VII. Related Work

The purpose of fence scoping is to constrain memory ordering effect of fences to certain scopes and hence improve program performance, while the correctness of programs is still enforced. In concurrent work to ours, Heterogeneous-Race-Free (HRF) [23] memory models are proposed very recently to formalize synchronization behavior with the notion of scopes in heterogeneous systems. Scope in HRF refers to a group of tasks or threads (e.g., sub-group, work-group, device, and system in OpenCL [35]). In contrast, scope in our work refers to the part of the program where the fence is valid. We believe both notions can co-exist for heterogeneous systems.

Many concurrent algorithms are usually first developed assuming Sequential Consistency (SC) memory model. However, to guarantee correctness under relaxed memory models, [3], [22] have proven that fences or atomic instructions are inevitable to build concurrent implementations of sets, queues, stacks, mutual exclusion, etc. Hence, it is essential to put fences, preferably as few as possible, in these algorithms. There have been works [14], [29], [41] on fence inference based on the concept of delay set analysis [38] to enforce SC, where they rely on static analysis to minimize the number of fences. Another group of research [8], [24]–[26], [32] uses model checking techniques to insert fences to ensure SC. In addition, [34] takes a different approach to reducing number of fences. It exploits the relaxed semantics of work-stealing algorithm – tasks are allowed to be executed multiple times for some applications – and avoid some fences in the algorithm. *While our work does not target reducing number of fences, it is complementary to the above techniques to improve program performance*.

There has been work on reducing fence overhead. Techniques in [13], [27], [30], [31], [42], based on the observation that most fences are not necessary dynamically, reduce most memory ordering overhead due to fences non-speculatively. Techniques in [7], [9], [19], developed to enforce SC through speculation, can also be utilized to reduce memory ordering overhead of fences. More recently, [40] 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. Besides, there is work on optimizing lock implementations, which is related to fence optimizations. For example, [4] allows a particular thread to reserve a lock to make the acquisitions of the lock by the reserving thread more efficient; [36] proposes speculative lock elision to dynamically eliminate lock operations. *Our work is different in that we reduce fence overhead by constraining memory ordering effect of fences in certain scopes. In particular,* S-Fence *only makes changes locally in each processor core, without adding inter-processor communication for multiprocessors, and hence scalability is not a problem for* S-Fence.

There are also finer fences in commercial architectures [2]. In Intel IA-32, there are three types of fence instructions, i.e., *mfence*, *lfence* and *sfence*. While *mfence* enforces all memory orders, *lfence* only enforces orders between memory load instructions and *sfence* only enforces orders between memory store instructions. Moreover, in SPARC V9, MEMBAR instruction can be customized to enforce different memory orders; in PowerPC, there are lightweight fence *lwsync* and heavyweight fence *sync*, where *sync* is a full fence, while *lwsync* guarantees all other orders except RAW; in Alpha model, there are memory barrier (MB) and write memory barrier (WMB). Although our work also provides fences for enforcing ordering of a subset of memory operations, it is orthogonal to the above works – they refine the semantics of full fence in different directions. While the existing works explore the ordering of a combination of previous load and store operations with respect to future load and store operations, *our work explores the ordering of a certain set of memory accesses that are in the scope of fences. However, they are also complementary* – the idea of S-Fence can be combined with the above various finer fences to further improve program performance.

Besides, there is also some work using specified variables to allow compilers to do optimizations across fence instructions. For example, OpenMP [1] provides *flush* instruction with a list of variables. This allows compiler to reorder accesses to variables that are not included in the list across the *flush* instruction. In the same spirit, Cedar [17] provides *advance* and *await* routines with specified variables for synchronization. Cedar targets Fortran, but performing the same optimization analysis at compile time in C/C++ code would be very hard. On the contrary, S-Fence *focuses on hardware optimization, reordering memory operations across fences at runtime*. The above compiler work would be complementary to S-Fence.

VIII. Conclusion

We propose the concept *fence scope*, and a new fence instruction *scoped fence* (S-Fence), which is constrained in its scope. S-Fence expresses programmers' intention in their programs, and conveys such information to the hardware to reduce memory ordering requirements. S-Fence is easy to be incorporated in current popular object-oriented programming languages, and the hardware support is lightweight. The experiments show that S-Fence achieves peak speedups ranging from 1.13x to 1.34x for lock-free algorithms, and obtains speedups from 1.04x to 1.23x for full applications.

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 grants CCF-1318103 and CCF-0963996 to the University of California, Riverside, and by an Intel early career faculty award and EPSRC grants EP/G036136/1 and EP/L000725/1 to the University of Edinburgh.

References

[1] OpenMP application program interface. http://www.openmp.org/mp-documents/spec30.pdf.

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

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, 2011.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 258–268, 1998.

[5] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel Distrib. Comput.*, 65(9):994–1006, Sept. 2005.

[6] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, 1993.

[7] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of ISCA-36*, pages 233–244, 2009.

[8] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 12–21, 2007.

[9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of ISCA-34*, pages 278–289, 2007.

[10] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, 2005.

[11] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545, 2008.

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

[13] Y. Duan, A. Muzahid, and J. Torrellas. Weefence: toward making fences free in TSO. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 213–224, 2013.

[14] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 285–294, 2003.

[15] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, 1998.

[17] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar: A large scale multiprocessor. *SIGARCH Comput. Archit. News*, 11(1):7–11, Mar. 1983.

[18] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, ISCA '91, pages 355–364, 1991.

[19] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of ISCA-26*, pages 162–171, 1999.

[20] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, 2001.

[21] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.

[22] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[23] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 427–440, 2014.

[24] T. Q. Huynh and A. Roychoudhury. Memory model sensitive bytecode verification. *Form. Methods Syst. Des.*, 31(3):281–305, Dec. 2007.

[25] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, 2010.

[26] M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 187–198, 2011.

[27] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 75–84, 2011.

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

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

[30] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 295–306, 2010.

[31] C. Lin, V. Nagarajan, and R. Gupta. Address-aware fences. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 313–324, 2013.

[32] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 429–440, 2012.

[33] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[34] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 45–54, 2009.

[35] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[36] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, 2001.

[37] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[38] 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.

[39] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[40] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 524–535, 2012.

[41] 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 *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–13, 2005.

[42] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu. Conditional memory ordering. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 41–52, 2006.

[43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of ISCA-22*, pages 24–36, 1995.