UNIVERSITY OF CALIFORNIA
RIVERSIDE

Imposing Minimal Memory Ordering on Multiprocessors

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Changhui Lin

August 2013

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Gianfranco Ciardo
Dr. Iulian Neamtiu
Dr. Philip Brisk

The Dissertation of Changhui Lin is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduation experience has been one that I will cherish forever.

Foremost, I would like to express my sincere gratitude to my advisor, Dr. Rajiv Gupta, for leading me to today's achievements. His perpetual enthusiasm in research, meticulous scholarship, extensive knowledge and insightful vision have influenced me immensely in the course of performing research. More importantly, he deeply cares about his students. I really appreciate his belief in my potential, which allows me to explore problems extensively. From him, I have learned innumerable lessons and insights on the working of academic research in general, which have helped me come a long way and will always guide me in future. *Thanks, Dr. Gupta!*

I would like to thank my other dissertation committee members, Dr. Gianfranco Ciardo, Dr. Iulian Neamtiu and Dr. Philip Brisk for taking their time to help me improve this dissertation.

I would like to express my gratitude to all the members of my research group including Vijay Nagarajan, Dennis Jeffrey, Chen Tian, Min Feng, Kishore Kumar Pusukuri, Yan Wang, and Sai Charan Koduru for helping me in many ways during these years. I would also like to extend a general thank you to all teachers I have had throughout my life. I am where I am today because of them.

Finally, I would like to thank my family, particularly my father Dianyi Lin, my mother Youfang Zhang, and my wife Ying He, for their unconditional support throughout my life and career.

To my wife and parents, for their endless love and support.

ABSTRACT OF THE DISSERTATION

Imposing Minimal Memory Ordering on Multiprocessors

by

Changhui Lin

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2013
Dr. Rajiv Gupta, Chairperson

Shared memory has been widely adopted as the primary system level programming abstraction on modern multiprocessor systems due to its ease of programming. To assist programmers in understanding program behaviors with respect to read and write operations originating from multiple processors, many *memory consistency models* have been proposed. *Sequential consistency* (SC) memory model is the simplest and most intuitive model, but its strict memory ordering requirements can restrict many hardware and compiler optimizations that are possible in uniprocessors. For higher performance, many manufacturers typically choose to support *relaxed consistency models*. In these models, *memory fence* instructions are also provided to permit selective overriding of default relaxed memory access ordering, where strict ordering must be enforced for program correctness. However, memory fences are costly because they cause a processor to stall.

Although the underlying memory models or memory fence instructions require a set of memory orderings to be enforced, they are not always necessary at runtime. If reordering of two memory operations executed on one processor is not observed by any other processor, then the reordering is safe as the program will behave the same as if they

were not reordered. Thus we observe that it is possible to relax some of memory orderings that are *in general required* but are *unnecessary in the current execution*. The goal of this dissertation is to dynamically identify the necessary memory orderings and thus relax the rest of the required memory orderings. The challenge of achieving this goal is to *efficiently* identify the necessary memory orderings.

This dissertation explores programming, compiler, and hardware support for eliminating unnecessary memory orderings to improve program performance. First, *conflict ordering* is proposed for implementing SC efficiently, where SC is enforced by explicitly ordering only conflicting memory operations in the global memory order, instead of all memory operations. The approach achieves SC performance comparable to RMO (relaxed memory order), without aggressive post-retirement speculation. Next, three approaches (i.e., *scoped fence, conditional fence,* and *address-aware fence*) are proposed to relax memory orderings imposed by memory fences, representing different levels of support from programmer, compiler and hardware. These approaches make memory fences lightweight to use.

The programming directed approach, *scoped fence* (S-Fence), introduces the concept of *fence scope* which constrains the effect of fences in programs. S-Fence enables programmers to specify the scope of fences using a customizable fence statement. The scope information is then 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. S-Fence bridges the gap between programmers' intention and hardware execution with respect to the memory ordering enforced by fences.

The compiler directed approach, *conditional fence* (C-Fence), utilizes compiler information to dynamically decide if there is a need to stall at each fence. The compiler

helps to identify *fence associates* and incorporate this information in a C-Fence instruction. At runtime, to decide whether a fence can be dynamically eliminated, the hardware uses the fence associate information to check if there is any associate that is executing. It significantly reduces fence overhead, while only requiring lightweight hardware support.

The hardware directed approach, *address-aware fence*, utilizes hardware support to collect memory access information from other processors to form a *watchlist* for each fence instance. The completion of a memory access following the fence is allowed if its memory address is not contained in the watchlist, appearing as if the fence does not take effect; otherwise, the memory access is delayed to ensure correctness. Address-aware fence is implemented in the microarchitecture without instruction set support and is transparent to programmers. It has the highest precision, eliminating nearly all possible unnecessary memory orderings due to fences.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multiprocessors are becoming ubiquitous in all computing domains, from mobile devices to datacenter servers as they are able to deliver high performance via parallelism. With this development, the importance of parallel programming continues to grow as the onus of writing parallel programs that deliver increased performance is on the programmers. It is a well recognized fact that writing and debugging parallel programs is not an easy task. Consequently, there has been significant research on developing programming models, memory models, and tools for making programmers' job easier. To simplify programming for multiprocessors, shared memory is widely adopted as the primary system level programming abstraction. In uniprocessors, as long as dependences are respected, even when accesses to different memory locations are reordered, the program behaves the same as if no reordering was performed. However, in multiprocessors, when accesses to different memory locations in shared memory are effectively reordered, the program can exhibit behaviors that are different from programmers' expectation.

P0		P1	
1	flag0 = 1	4	flag1 = 1
2	**if** (flag1 == 0)	5	**if** (flag0 == 0)
3	critical_section	6	critical_section

Figure 1.1: Effect of reordering in the Dekker's algorithm.

Consider the code segment in Fig. 1.1, which is a simplified version of the Dekker's algorithm [34, 4]. This algorithm is designed to allow only one processor to enter the critical section at a time, using two shared memory variables flag0 and flag1. The two flag variables indicate an intention on the part of each processor to enter the critical section. Initially, flag0 and flag1 are initialized to 0. When $P0$ attempts to enter the critical section, it first updates flag0 to 1 and checks the value of flag1. If flag1 is 0, under the most intuitive interpretation, $P1$ has not tried to enter the critical section, and it is safe for $P0$ to enter. However, this reasoning is valid, only if the machine that executes the program guarantees that accesses to flag0 and flag1 are performed in the order specified by the program. If, on the other hand, the machine allows memory operations to be reordered, it is possible for $P0$ to first read flag1 and then update flag0. Likewise, $P1$ may first read flag0 and then update flag1. As a result, both reads may potentially return the value of 0, allowing both $P0$ and $P1$ to enter the critical section simultaneously. Clearly, this is not what the programmer intended.

Therefore, many *memory consistency models* have been proposed to constrain memory behaviors with respect to read and write operations originating from multiple processors [4]. Each model specifies a contract between programmer and system (compiler and hardware), i.e., the program's behavior will be consistent with programmer's expectation if the programmer follows the rules of the underlying memory model. Of the various memory

consistency models, the *sequential consistency* (SC) model [64] is the simplest and most intuitive model. It requires that all memory operations of all processors appear to execute one at a time, and that the operations of a single processor appear to execute in the order described by the program. Moreover, to improve performance, many relaxed memory consistency models [5, 37, 47] have also been proposed that allow different subsets of memory accesses to be reordered. Each of these models offers a trade-off between *programmability* and *performance*. Manufacturers (e.g., Intel, IBM, Sun, ARM, etc.) typically choose to support relaxed consistency models, such as total store order (TSO), relaxed memory order (RMO), release consistency (RC), etc [4].

Under SC, the Dekker algorithm shown in Fig. 1.1 will behave correctly, as SC guarantees that, in $P0$ ($P1$), flag1 (flag0) is read only after flag0 (flag1) has been updated. However, under relaxed memory models, the above orderings are not guaranteed. To enforce the orderings that are critical to the correctness of programs, *fence instruction*, also known as memory fence or memory barrier, is provided to constrain memory reordering. A fence instruction guarantees that all memory accesses prior to it have completed, before its following memory accesses can be performed. Thus, programmers can insert fence instructions between the accesses to flag0 and flag1 to prevent reordering under relaxed memory models, as shown in Fig. 1.2 (Line 2 and Line 6).

P0		P1	
1	flag0 = 1	5	flag1 = 1
2	**FENCE**	6	**FENCE**
3	**if** (flag1 == 0)	7	**if** (flag0 == 0)
4	critical_section	8	critical_section

Figure 1.2: Ordering memory operations with fence instructions.

As we can see, memory consistency models and fence instructions impose constraints on memory ordering. The problem is that faithfully implementing these ordering constraints disallows many possible optimizations and sacrifices performance.

(Sequential consistency) Stronger memory consistency models (e.g., SC) impose stricter constraints on memory ordering. SC matches the programmers' expectation that a parallel program behaves as an interleaving of the memory accesses from its constituent threads. This enables programmers to understand and reason about their programs the best. But SC imposes strict memory ordering on programs that restricts many compiler optimizations (e.g., load reordering, store reordering, caching value in register) [75, 22] and hardware techniques (e.g., write buffer) that are possible in uniprocessors, and hence hurts program performance. Indeed, most of the processor families only support relaxed memory models, which impose less constraints on memory ordering. However, relaxed memory models require programmers' expertise to enforce the correctness of programs, as data sharing among threads is often subtle and complex, placing heavy burdens on programmers. This sacrifices programmability for performance.

(Fence instructions) Typically, fence instructions are substantially slower than regular instructions. The fence semantics requires ordering of memory operations before and after the fence, resulting in the stalling of the processor, which is costly [63, 102]. Under relaxed memory models, excessive use of fences will incur high overhead. In commercial applications, frequent thread synchronizations can result in significant ordering delays due to fence instructions [107]. For example, in [41], Frigo *et al.* observe that in an Intel multiprocessor, Cilk-5's THE protocol spends half of its time executing a memory fence. As we can see, memory ordering imposed by fence instructions can also hurt program performance.

Although the underlying memory models or fence instructions require a set of memory orderings should be enforced, they are not always necessary at runtime. If re-ordering of two memory operations executed on one processor is not observed by any other processor, then the reordering is safe as the program will behave the same as if they were not reordered. Thus we observe that it is possible to relax some of memory orderings that are *in general required* but are *unnecessary in the current execution*. In Fig. 1.3, all possible memory orderings at runtime are represented by ALL, and a subset of them (REQUIRED) are required orderings. However, only a smaller subset of them (NECESSARY) really need to be enforced in a given execution, as enforcing them leads to the same program behavior as if all memory orderings in REQUIRED are enforced. Therefore, we are able to relax the memory orderings in the set (REQUIRED - NECESSARY). By doing so, more memory operations can be reordered, reducing stalls and increasing performance. In fact, a vast majority of memory orderings are unnecessary dynamically [49, 63, 102], which provides a good opportunity to improve program performance.

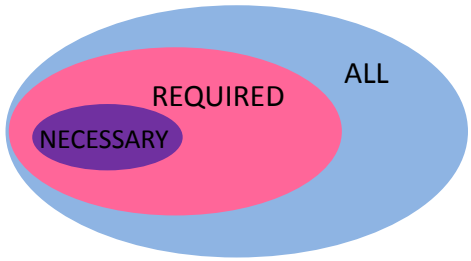

Figure 1.3: Venn diagram of memory orderings.

Fences can be utilized to enforce sequential consistency (SC) [64] for programs running on machines only supporting relaxed memory consistency models. The naive way to achieve this is by inserting a fence before each memory operation. Fig. 1.4 shows the

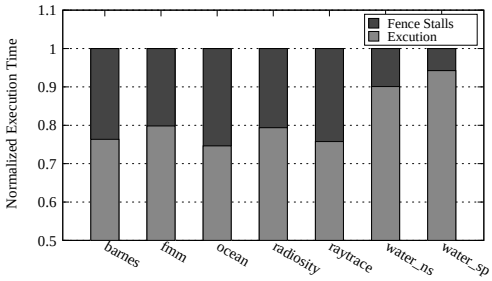

Figure 1.4: Execution overhead of fences.

Benchmarks	*Total*	*Necessary*
barnes	83M	206
fmm	5M	223
ocean	9M	383
radiosity	30M	90
raytrace	44M	239
water-ns	9M	79
water-sp	7M	88
AVG.	28.8M	218

Figure 1.5: Necessary fences.

overhead due to fences for a group of benchmark programs taken from the SPLASH-2 [108] benchmark suite. The execution time is broken down into two parts: the stall time due to fences (*Fence stalls*) and the rest of the execution time (*Execution*). As we can see, these programs experience significant overhead due to fences (about 20% on average). Although static analysis is able to reduce the number of inserted fences [95, 66, 39], the overhead incurred by fences still accounts for a significant part of the overall execution time. However, most of fence instances are unnecessary dynamically. To show this, we studied the total number of fence instances encountered during execution and the number of necessary fence instances. We say *a fence instance F is necessary*, if at runtime memory operations across the fence F conflict with memory operations across a fence F' in another processor. This is because, if the memory locations accessed by memory operations across the fence F are not simultaneously accessed by other processors, then the fence is not necessary. The results are shown in Fig. 1.5. As we can see, Column 2 shows the total number of dynamic fence instances encountered in each program. Although they only account for a small part of all instructions ($\sim$1%), they induce relatively larger execution time overhead. Column 3 shows the number of necessary fences. We can see that only very few fence instances are necessary to effect to order memory operations across them. This is because

(1) the execution of the program often already conforms to the effect that fences strive to enforce; and (2) since fences are inserted conservatively, many dynamic fence instances are unnecessary. Therefore, there are a great many opportunities for eliminating unnecessary memory orderings.

The goal of this dissertation is to dynamically identify the necessary memory orderings and thus relax the rest of the required memory orderings. The challenge of achieving this goal is to *efficiently* identify the necessary memory orderings. There has been a group of hardware techniques that aim at eliminating unnecessary stalls due to memory ordering requirements [46, 87, 49, 48, 25, 16]. Although these techniques are able to achieve good performance, the hardware complexity associated with aggressive speculation they employ can hinder widespread adoption [3]. Many compiler techniques have also been proposed to compute a set of memory access pairs whose ordering automatically guarantees SC [39, 55, 59, 60, 66, 79, 80, 95, 98]. To ensure that these memory access pairs are not reordered, memory fences are inserted by the compiler. The main issue with these compiler techniques is performance overhead, as necessary memory orderings should be identified conservatively, and thus insertion of fences can significantly slowdown the program. Besides, combined HW/SW techniques [102, 63] have also been proposed to reduce memory ordering overhead. However, they are only applicable to specific applications. This dissertation is aimed at identifying minimal memory orderings without the above limitations, by exploring programming, compiler, and hardware support for eliminating unnecessary memory orderings to improve program performance.

1.1 Dissertation Overview

This dissertation presents the following four approaches to efficient SC implementation and reducing fence overhead. The first approach is a hardware solution for implementing SC efficiently. The remaining three approaches focus on reducing the overhead of fence instructions. They represent different levels of support from programmer, compiler, and hardware.

1.1.1 Sequential consistency

SC requires that memory operations appear to complete in program order. To ensure this, prior SC implementations force memory operations to *explicitly* complete in program order. However, is program ordering necessary for ensuring SC? Let us consider the example in Fig. 1.6, which shows memory operations a_1, a_2, b_1 and b_2 from processors A and B. (a_1, b_2) and (a_2, b_1) are two pairs of conflicting accesses. Furthermore, let us assume that while memory operations b_1 and b_2 have completed, a_1 and a_2 are yet to complete. Can a_2 complete before a_1 in an execution without breaking SC? Prior SC implementations forbid this and force a_2 to wait until a_1 completes, either explicitly or using speculation. We observe that a_1 *appears* to complete before a_2, as long as a_2 is made to complete after b_1, with which a_2 conflicts. In other words, a_2 can complete before a_1 without breaking SC, as long as we ensure that a_2 waits for b_1 to complete. Indeed, if b_1 and b_2 have completed before a_1 and a_2, the execution order $b_1 \rightarrow b_2 \rightarrow a_2 \rightarrow a_1$ is equivalent to the original SC order $b_1 \rightarrow b_2 \rightarrow a_1 \rightarrow a_2$ as the values read in the two executions are identical.

This dissertation presents *conflict ordering* for implementing SC efficiently, where SC is enforced by explicitly ordering only conflicting memory operations. Conflict ordering

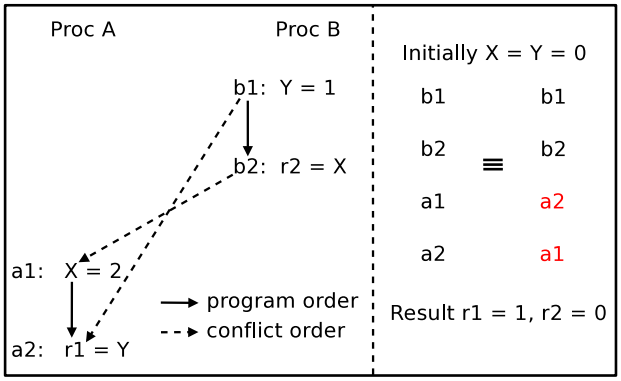

Figure 1.6: A motivating example for conflict ordering.

allows a memory operation to complete, as long as all conflicting memory operations prior to it in the global memory order have completed, instead of all memory operations. The approach can achieve SC performance comparable to RMO (relaxed memory order), without aggressive post-retirement speculation.

1.1.2 Fence instructions

This dissertation also explores compiler, programming, and hardware support for reducing memory orderings due to fences. The proposed techniques are programming, compiler, and hardware directed approaches to help hardware eliminate unnecessary memory orderings, respectively. Table 1.1 compares the proposed approaches in terms of their programming support, compiler complexity, hardware complexity, and expected precision to identify unnecessary memory orderings.

	Programming Support	Compiler Complexity	Hardware Complexity	Precision
Programming Directed	yes	medium	low	medium
Compiler Directed	none	high	medium	medium
Hardware Directed	none	none	high	high

Table 1.1: Trade-offs of solutions for eliminating fence overhead.

Programming Directed Approach

Traditional fence instructions order memory operations without being aware of 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. One common application of such ordering demands can be found in the design of concurrent lock-free algorithms/data structures, which are widely used in multithreaded programming. In these algorithms, fence instructions and atomic instructions are required to ensure correctness when they are executed under relaxed memory models. Programs using concurrent algorithms usually exhibit the pattern shown in Fig. 1.7. Such programs repeatedly access shared data controlled by concurrent algorithms and then process the accessed data. The fences in the concurrent algorithms are only supposed to guarantee the correct concurrent accesses to shared data, without being aware of how the accessed data is processed afterwards. However, due to their semantics, traditional fences also order memory operations which belong to the code 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 operations that must be ordered by a fence from the rest of memory operations.

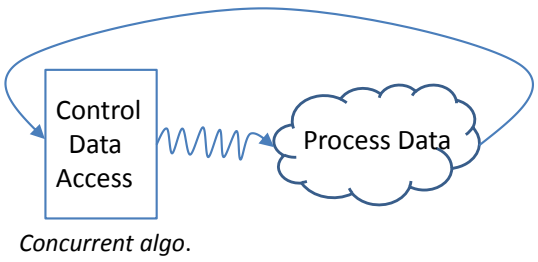

Figure 1.7: A common data access pattern.

This dissertation presents the concept of *fence scope* which constrains the effect of fences in programs. We call such a fence as *scoped fence*, S-Fence for short. S-Fence only orders memory operations in its scope, without being aware of memory operations beyond the scope. S-Fence enables programmers to specify the scope of fences using a customizable fence statement. Such 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 operations in the scope. S-Fence bridges the gap between programmers' intention and hardware execution with respect to the memory ordering enforced by fences.

S-Fence involves the programmer in helping hardware dynamically eliminate part of memory orderings in (REQUIRED - NECESSARY) (Fig. 1.3). Programmer can easily specify the fence scope using the provided programming support; compiler encodes scope information in the binary, which is straightforward; and the modification to hardware is simple, without requiring inter-processor communication. S-Fence does not order memory operations beyond the scope; however, it still has to order all memory operations in the scope, where some memory orderings may be unnecessary.

Compiler Directed Approach

We make the observation that the ordering of memory accesses that memory fences strive to enforce, is often already enforced in the normal course of program execution; this obviates the need for memory fences most of the time. For instance, consider the scenario shown in Fig. 1.8, in which the two requests for critical section are staggered in time. In particular, by the time processor $P1$ tries to enter the critical section, the update to flag0 by

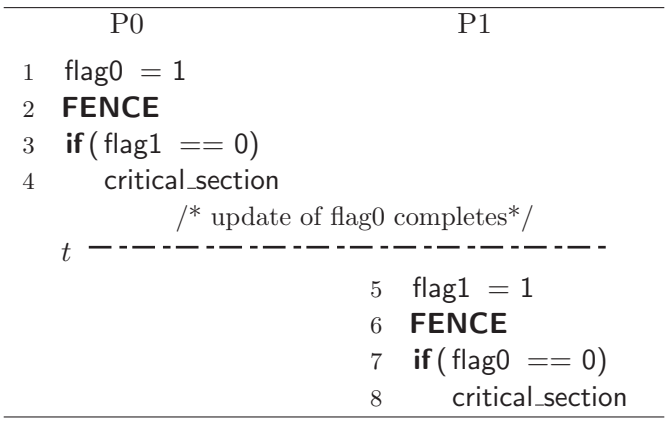

Figure 1.8: Scenario in which memory ordering is not necessary.

processor $P0$ has already completed. Clearly, under this scenario, memory accesses to flag0 (Line 1) and flag1 (Line 3) need not be ordered by the fence. Let us call the fences in Line 2 and Line 6 as *associates*, as they are used to order the same pair of conflicting accesses to flag0 and flag1. Intuitively, for each fence, as long as its associates are far away from it at runtime, we can safely execute past the fence. Furthermore, the study conducted on SPLASH-2 [108] programs with compiler-inserted memory fences shows that this is indeed the common case; on account of the conflicting accesses typically being staggered, only 8% of the executed instances of the memory fences are really needed.

This dissertation presents the *conditional fence* (C-Fence) mechanism, which utilizes compiler information to dynamically decide if there is a need to stall at each fence. The compiler helps to identify fence associates and incorporate this information in a C-Fence instruction. At runtime, to decide whether a fence can be dynamically eliminated, the hardware uses the compiler information to check if there is any associate that is executing. While traditional fence imposes intraprocessor delays between memory operations before and after the fence, C-Fence imposes interprocessor delays between fence associates, which incurs much less overhead.

C-Fence utilizes compiler information to help hardware dynamically eliminate subset of memory orderings in (REQUIRED - NECESSARY) (Fig. 1.3). It does not require programming effort, and the hardware cost is lightweight. C-Fence works as long as the compiler can provide fence associate information. On the other hand, since C-Fence relies on compiler to statically identify associate information, it is conservative, and thus it only eliminates subset of unnecessary memory orderings.

Hardware Directed Approach

C-Fence and S-Fence are only able to eliminate a subset of unnecessary memory orderings due to fences. Fig. 1.9 shows several scenarios where unnecessary fences are not eliminated by either C-Fence or S-Fence. The example in Fig. 1.9(a) contains a conditional branch. Fence instructions are inserted to ensure that memory accesses to shared variables x and y are ordered properly. However, if at runtime the condition in Proc 1 evaluates to `false`, and hence $a1$ is not executed, then the shared variable x is not accessed concurrently by multiple processors. As a result, the fence instance is not necessary to effect the order of memory accesses of x and y. In Fig. 1.9(b), there are two pointers p and q pointing to some field of a shared object (e.g., a hash table [86]) respectively – since they may point to the same field, fence instructions are inserted. However, at runtime, if they point to different fields of the shared object, i.e., $a1$ and $b2$ do not access the same field concurrently, then the fence instances are not necessary. Finally, in Fig. 1.9(c), `FENCE2` orders both $b1 \rightarrow b2$ and $b1 \rightarrow b3$. However, if z is not accessed in any other processor concurrently, $b2$ need not be delayed and hence can be reordered across the fence. Thus, even though the instance of `FENCE2` is necessary, $b2$ need not be delayed. Moreover, C-Fence and S-Fence require

programming/compiler support to help hardware detect unnecessary memory orderings. However, when there is no source code available, they cannot be applied; and performing interprocedural alias analysis in compiler is complex and conservative.

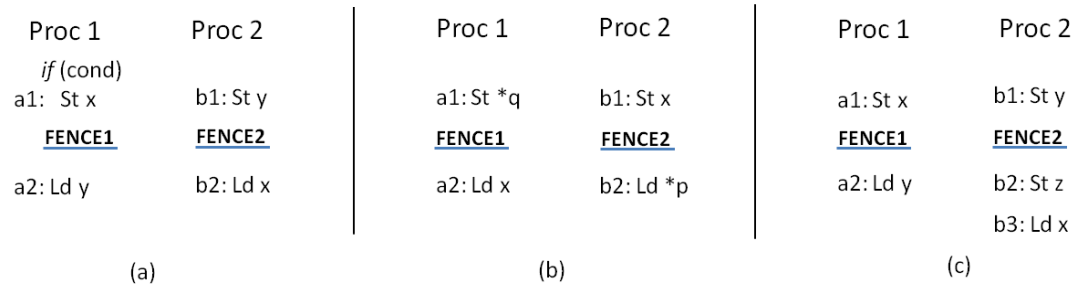

Figure 1.9: Unnecessary fence instances at runtime.

This dissertation presents *address-aware fence* mechanism, a hardware mechanism to eliminate stalling at unnecessary fence executions without resorting to speculation. Unlike a traditional fence which is processor-centric, an address-aware fence collects memory access information from other processors to decide whether it should effect the stalling of following memory accesses. In particular, an address-aware fence instance has an associated *watchlist*, which contains memory addresses that should not be accessed by the memory accesses following the fence. The completion of a memory access following the fence is allowed if its memory address is not contained in the watchlist, appearing as if the fence does not take effect. Otherwise, the memory access is delayed to ensure correctness. Therefore this approach effectively reduces overhead of fences whenever possible.

Address-aware fence eliminates memory orderings in (REQUIRED - NECESSARY) (Fig. 1.3) without the help from compiler or programmer. It is implemented in the microarchitecture without instruction set support and is transparent to programmers. Fence instructions in the executable are identified at runtime and processed by the hardware.

Address-aware fence has the highest precision, eliminating nearly all possible unnecessary memory orderings due to fences.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes *conflict ordering*, a hardware approach for efficient SC implementation. Chapter 3, Chapter 4 and Chapter 5 present *scoped fence*, *conditional fence*, and *address-aware fence*, respectively. Related work is given in Chapter 6. Chapter 7 summarizes the contributions of this dissertation and identifies directions for future work.

Chapter 2

Efficient Sequential Consistency via Conflict Ordering

Among various memory consistency models, the *sequential consistency* (SC) model in which memory operations appear to take place in the order specified by the program is most intuitive to programmers. Indeed, most works on semantics and software checking that strive to make concurrent programming easier assume SC [94]; several debugging tools for parallel programs, e.g. *RaceFuzzer* [93], also assume SC. In spite of the advantages of the SC model, processor designers typically choose to support relaxed consistency models; none of the Intel, AMD, ARM, Itanium, SPARC, or PowerPC processor families choose to support SC. This is because SC requires reads and writes to be ordered in program order, which can cause significant performance overhead. Indeed, SC requires the enforcement of all four possible program orderings: $r \rightarrow r$, $r \rightarrow w$, $w \rightarrow w$, $w \rightarrow r$, where r denotes a read operation and w denotes a write operation. Prior hardware SC implementations enforce SC by forcing memory operations to *explicitly* complete in program order, which are not

able to hide some kinds of stalls due to memory ordering requirements or require aggressive hardware speculation.

This chapter presents *conflict ordering* for implementing SC efficiently, where SC is enforced by explicitly ordering only conflicting memory operations. Conflict ordering allows a memory operation to complete, as long as all conflicting memory operations prior to it in the global memory order have completed, instead of all memory operations. The approach can achieve SC performance comparable to RMO (relaxed memory order), without post-retirement speculation.

2.1 Background

In program ordering based SC implementations, the hardware directly ensures all four possible program ordering constraints [16, 46, 48, 49, 87]; they are based on the seminal work by Scheurich and Dubois [92], in which they state the sufficient conditions for SC for general systems with caches and interconnection networks.

(Naive) A naive way to enforce program ordering between a pair of memory operations is to delay the second until the first fully completes. However, this can result in a significant number of memory ordering stalls; for instance, if the first access is a miss, then the second access has to wait until the miss is serviced.

(In-window speculation) Out-of-order processing capabilities of a modern processor can be leveraged to reduce some of these stalls. This is based on the observation that memory operations can be freely reordered as long as the reordering is not observed by other processors. Instead of waiting for an earlier memory operation to complete, the processor can use hardware prefetching and speculation to execute memory operations out

of order, while still completing in order [46, 109]. However, such reordering is within the instruction window, where the instruction window refers to the set of instructions that are in-flight. If the processor receives an external coherence (or replacement) request for a memory operation that has executed out of order, the processor's recovery mechanism is triggered to redo computation starting from that memory operation. Nonetheless, enforcing the $w \rightarrow r$ (and $w \rightarrow w$) order necessitates that the write-buffer is drained before a subsequent memory operation can be completed. Thus, a high latency write can still cause significant program ordering stalls, which in-window speculation is unable to hide. The experiments with the SPLASH-2 programs show that programs spend more than 20% of their execution time on average waiting for the write buffer to be drained. For this work, we assume in-window speculation support as part of baseline implementations of SC as well as fences in RMO.

(Post-retirement speculation) Since in-window speculation is not sufficiently able to reduce program ordering stalls, researchers have proposed more aggressive speculative memory reordering techniques [16, 25, 26, 42, 48, 49, 50, 87, 107]. The key idea is to *speculatively retire* instructions neglecting program ordering constraints while maintaining the state of speculatively-retired instructions separately. One way to do this is to maintain the state of speculatively-retired instructions at a fine granularity, which enables precise recovery from misspeculations [48, 49, 87]. This obviates the need for a load to wait until the write buffer is drained and is made to retire speculatively. More recently, researchers have proposed *chunk based* techniques [16, 25, 26, 42, 50, 107] which again use aggressive speculation to efficiently enforce SC at the granularity of coarse-grained chunks of instructions instead of individual instructions. While the above approaches show much promise, hardware complexity associated with aggressive speculation, being contrary to the design philosophy of multi-cores

consisting of simple energy-efficient cores [1, 2], can hinder wide-spread adoption. Through this chapter, we seek to show that a lightweight SC implementation is possible without sacrificing performance.

2.2 SC via Conflict Ordering

This section first informally describes the approach of enforcing SC using conflict ordering, and then formally proves that conflict ordering correctly enforces SC.

2.2.1 Conflict ordering

SC requires memory operations of all processors to appear to perform in some *global memory order*, such that, memory operations of each processor appear in this global memory order in the order specified by the program [64]. Prior SC implementations ensure this by enforcing program ordering by explicitly completing memory operations in program order. More specifically, if m_1 and m_2 are two memory operations with m_1 preceding m_2 in the program order, prior SC implementations ensure $m_1 \rightarrow m_2$ by allowing m_2 to complete only after m_1 completes (either explicitly or using speculation); in effect, m_2 is made to wait until m_1 and all of m_1's predecessors in the global memory order have completed. In other words, if $pred(m_1)$ refers to m_1 and its predecessors in the global memory order, m_2 is allowed to complete only when all memory operations from $pred(m_1)$ have completed.

While enforcing program ordering is a sufficient condition for ensuring SC [92], it is not a necessary condition [43, 95]. This chapter presents *conflict ordering*, a novel approach to SC, in which SC is enforced by explicitly ordering only conflicting memory operations. Conflict ordering allows a memory operation to complete, as long as all conflicting memory

operations prior to it in the global memory order have completed. More specifically, let m_1 and m_2 be consecutive memory operations; furthermore let $pred(m_1)$ refer to memory operations that include m_1 and those before m_1 in the global memory order. Conflict ordering allows m_2 to safely complete as long as those memory operations from $pred(m_1)$, which conflict with m_2, have completed. It is worth noting that the above condition generally evaluates to true, since in well-written parallel programs conflicting accesses are relatively rare and staggered in time [49]. For example, in the scenario shown in Fig. 1.6, a_1 *appears* to complete before a_2, as long as a_2 is made to complete after b_1, with which a_2 conflicts. In other words, a_2 can complete before a_1 without breaking SC, as long as we ensure that a_2 waits for b_1 to complete.

2.2.2 Proof of correctness

We prove that conflict ordering enforces SC using the formalism of Shasha and Snir [95]. For the following discussion, we assume an invalidation based cache coherence protocol for processors with private caches. A read operation is said to complete when the returned value is bound and can not be updated by other writes; a write operation is said to complete when the write invalidates all cached copies, and the generated invalidates are acknowledged [46]. Furthermore, we assume that the coherence protocol serializes writes to the same location and also ensures that the value of a write not be returned by a read until the write completes – in other words, we assume that the coherence protocol ensures *write atomicity* [4].

Definition 1. *The program order* $\mathbf{P}$ *is a local (per-processor) total order which specifies the order in which the memory operations appear in the program. That is,* $m_1 \mathbf{P} m_2$ *iff the memory operation* m_1 *occurs before* m_2 *in the program.*

Definition 2. *The conflict relation* $\mathbf{C}$ *is a symmetric relation on* $\mathbf{M}$ *(all memory operations) that relates two memory operations (of which one is a write) which access the same address.*

Definition 3. *An execution* $\mathbf{E}$ *(or execution order or conflict order) is an orientation of* $\mathbf{C}$*. If* $m_1 \mathbf{C} m_2$*, then either* $m_1 \mathbf{E} m_2$ *or* $m_2 \mathbf{E} m_1$ *holds.*

Definition 4. *An execution* $\mathbf{E}$ *is said to be atomic, iff* $\mathbf{E}$ *is a proper orientation of* $\mathbf{C}$*, that is, iff* $\mathbf{E}$ *is acyclic.*

Definition 5. *An execution* $\mathbf{E}$ *is said to be sequentially consistent, iff* $\mathbf{E}$ *is consistent with the program order* $\mathbf{P}$*, that is, iff* $\mathbf{P} \cup \mathbf{E}$ *has no cycles.*

Definition 6. *The global memory order* $\mathbf{G}$ *is the transitive closure of the program order and the execution order,* $\mathbf{G} = (\mathbf{P} \cup \mathbf{E})^+$*.*

Remark. In the scenario shown in Fig. 1.6, b_1 appears before a_1 in the global memory order, since b_1 appears before b_2 in program order, and b_2 is ordered before a_1 in the execution order. That is, $b_1 \mathbf{G} a_1$ since $b_1 \mathbf{P} b_2$ and $b_2 \mathbf{E} a_1$.

Definition 7. *The function* $pred(m)$ *returns a set of memory operations that appear before* m *in the global memory order* $\mathbf{G}$*, including* m*. That is,* $pred(m) = \{m\} \cup \{m' : m' \mathbf{G} m\}$*.*

Definition 8. *An SC implementation constrains the execution by enforcing certain conditions under which a memory operation can be completed. An SC implementation is said to be correct if it is guaranteed to generate an execution that is sequentially consistent.*

Definition 9. *Conflict ordering is the proposed SC implementation in which a memory operation* m_2 *(whose immediate predecessor in program order is* m_1*) is allowed to complete*

iff those memory operations from $pred(m_1)$ *which conflict with* m_2 *have completed. That is,* m_2 *is allowed to complete iff* $\{m \in pred(m_1) : m\mathbf{C}m_2\}$ *have already completed.*

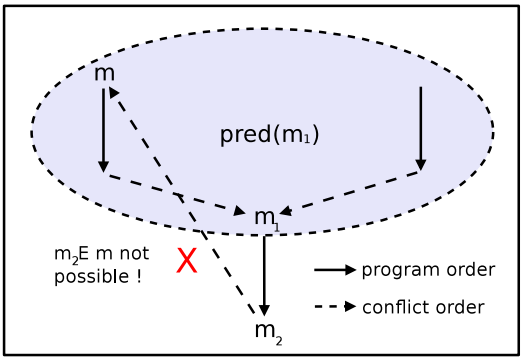

Figure 2.1: Conflict ordering ensures SC.

Lemma 1. *Any execution* $\mathbf{E}$ *(Definition 3) is atomic.*

Proof. Let w and r with a subscript be a write operation and a read operation. Write-serialization ensures that $w_1\mathbf{E}w_2$ iff w_1 completes before w_2; furthermore, $w_1\mathbf{E}w_2$ and $w_2\mathbf{E}w_3$ result in $w_1\mathbf{E}w_3$. Write-atomicity also ensures that $w_1\mathbf{E}r_2$ iff w_1 completes before r_2. Since reads are atomic, $r_1\mathbf{E}w_2$ iff r_1 completes before w_2. Furthermore, $w_1\mathbf{E}r_2$ and $r_2\mathbf{E}w_3$ result in $w_1\mathbf{E}w_3$. Thus, $\mathbf{E}$ is acyclic; therefore, from Definition 4, $\mathbf{E}$ is atomic. □

Theorem 1. *Conflict ordering is correct.*

Proof. We need to prove that any execution $\mathbf{E}$ generated by conflict ordering is sequentially consistent. That is, to prove that the graph $\mathbf{P} \cup \mathbf{E}$ is acyclic (from Definition 5). Let us attempt a proof by contradiction, and assume that there is a cycle in $\mathbf{P} \cup \mathbf{E}$. Since $\mathbf{E}$ is acyclic (from Lemma 1), the assumed cycle should contain at least one program order edge. Without loss of generality, let us assume that the program order edge that contains a cycle is $m_1\mathbf{P}m_2$ as shown in Fig. 2.1. To complete the cycle, there must be a conflict order

edge $m_2\mathbf{E}m$, where $m \in pred(m_1)$. Conflict ordering, however, ensures that those memory operations from $pred(m_1)$, which conflict with m_2 would have completed before m_2 (from Definition 9). Since $m \in pred(m_1)$, m would have completed before m_2. Thus, the conflict order edge $m_2\mathbf{E}m$ is not possible, which results in a contradiction. Thus, conflict ordering is correct. □

2.3 Basic Hardware Design

This section describes the hardware design that incorporates conflict ordering to enforce SC efficiently. For the basic design, all memory operations will have to check for conflicts before they can complete. In the next section, an enhanced design will be described, where we can determine phases in program execution where memory operations can complete without checking for conflicts.

(System Model) For the following discussion we assume a tiled chip multiprocessor, with each tile consisting of a processor, a local L1 cache and a single bank of the shared L2 cache. We assume that the local L1 caches are kept coherent using a directory based cache coherence protocol, with the directory distributed over the L2 cache. We assume that addresses are distributed across the directory at a page granularity using the first touch policy. We assume a directory protocol in which the requester notifies the directory on transaction completion, so each coherence transaction can have a maximum of four steps. Thus, each coherence transaction remains active in the directory until the time at which it receives the notification of transaction completion. Furthermore, we assume that the cache coherence protocol provides write atomicity. We assume each processor core to be a dynamically scheduled ILP processor with a reorder buffer (ROB) which supports

in-window speculation. All instructions, except memory writes, are made to complete in program order, as and when they retire from the ROB. Writes on the other hand, are made to retire into the write-buffer, so that the processor need not wait for them to complete. Finally, it is worth noting that conflict ordering is also applicable to bus based coherence protocols; indeed, we report experimental results for both bus based and directory based protocols.

2.3.1 Basic conflict ordering

We next describe the conflict ordering implementation which allows memory operations (both reads and writes) to complete past pending writes, while ensuring SC; it is worth noting, however, that the system model does not allow memory operations to complete past pending reads. Recall that conflict ordering allows a memory operation m_2, whose immediate predecessor is m_1, to complete as long as m_2 does not conflict with those memory operations from $pred(m_1)$ which are pending completion – let us call such pending operations as $pred_p(m_1)$. Thus, for m_2 to safely complete, we need to be sure that m_2 does not conflict with any of the memory operations from $pred_p(m_1)$. Alternatively, if $addr_p(m_1)$ refers to the set of addresses accessed by memory operations from $pred_p(m_1)$, we can safely allow m_2 to complete if its address is not contained in $addr_p(m_1)$. The challenge is to determine $addr_p(m_1)$ as quickly as possibly – and in particular without waiting for m_1 to complete. Next, we show how we compute $addr_p(m_1)$ for a write-miss, cache-hit, and a read-miss respectively.

(Write-misses) Our key idea for determining $addr_p(m_1)$ for a write-miss m_1 is to simply get this information from the directory. We show that those memory operations from

$pred_p(m_1)$, if any, would be present in the directory (as pending memory operations that are currently being serviced in the directory), provided write-misses such as m_1 are issued to the directory, before subsequent memory operations are allowed to complete. In other words, if *addr-list* refers to the set of addresses of the cache misses being serviced in the directory, $addr_p(m_1)$ would surely be contained in *addr-list*. In addition to this, since the system model does not allow memory operations to complete past pending reads, we are able to show that $addr_p(m_1)$ would surely be contained in *write-list* – where *write-list* refers to the set of memory addresses of the write-misses being serviced in the directory. Consequently, when the write-miss m_1 is issued to the directory, the directory replies back with the *write-list*, containing the addresses of the write-misses which are currently being serviced by the directory (to minimize network traffic we safely approximate the write-list by using a bloom filter). A subsequent memory operation m_2 is allowed to complete, only if m_2 does not conflict with any of the writes from the write-list. If m_2 does conflict, then the local cache block m_2 maps to is invalidated, and the memory operation m_2 and its following instructions are replayed when necessary. During replay, m_2 would turn out to be a cache miss and hence the miss request would be sent to the directory. This would ensure that m_2 would be ordered after its conflicting write that was pending in the directory. *It is worth noting that memory operations that follow a pending write, will now need to wait only until the write is issued to the directory and get a reply back from the directory, as opposed to waiting for the write-miss to complete.* While the former takes as much time as the round trip latency to the directory, the latter can take a significantly longer time – since it may involve the time taken to invalidate all shared copies, and also access memory if it is a miss.

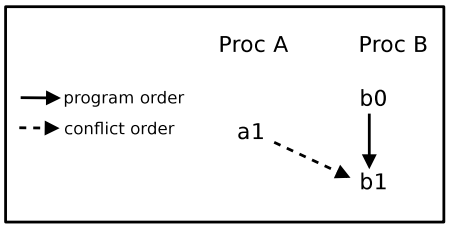

Figure 2.2: $pred_p(b_1) = \{b_1\} \cup pred_p(b_0) \cup pred_p(a_1)$.

(Cache-hits) Fig. 2.2 shows how $pred_p(b_1)$ relates to $pred_p(b_0)$, where b_0 is the immediate predecessor of b_1 in the program order, and a_1 is the immediate predecessor of b_1 in the conflict order. As we can see, $pred_p(b_1) = \{b_1\} \cup pred_p(b_0) \cup pred_p(a_1)$; this is because the global memory order is the union of the program order and the conflict order. If, however, b_1 is a cache hit, then its immediate predecessor in the conflict order must have completed and hence cannot be pending; if it were pending, b_1 would become a cache-miss. Thus $pred_p(b_1) = \{b_1\} \cup pred_p(b_0)$. Since memory operations that follow a cache-hit are allowed to complete only after the cache hit completes, $pred_p(b_1)$ remains the same as $pred_p(b_0)$. Consequently, $addr_p(b_1)$ remains the same as $addr_p(b_0)$ and thus, the write-list for a cache-hit need not be computed afresh.

(Read-misses) One important consequence of completing memory operations past pending writes is that, when a read-miss completes, there might be writes before it in the global memory order that are still pending. As shown in Fig. 2.3(a), if b_2 is allowed to complete before b_1, b_1 might still be pending when a_1 completes. Now, conflict ordering mandates that memory operations that follow a_1 can complete, only when they do not conflict with $pred_p(a_1)$. To enable this check, read-misses are also made to consult the directory and determine the write-list. Accordingly, when read-miss a_1 consults the directory, it fetches the write-list and replies to the processor. Having obtained the write-list, a_2 which does

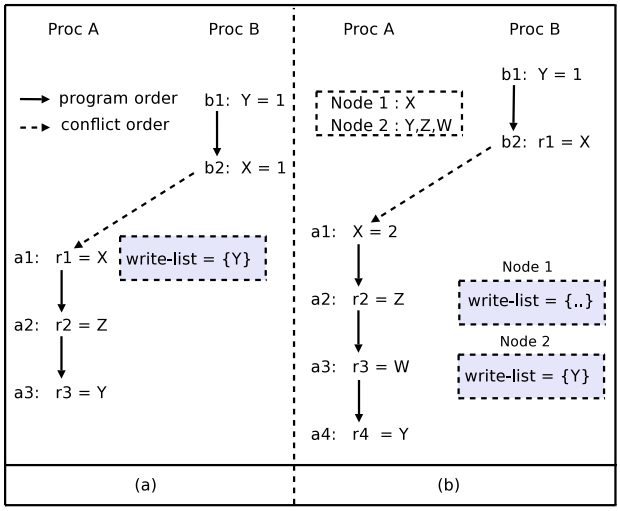

Figure 2.3: Basic conflict ordering: Example.

not conflict with memory operations from the write-list, is allowed to complete. Memory operation a_3, since it conflicts, is replayed obtaining its value from b_1 via the directory.

2.3.2 Handling distributed directories

To avoid a single point of contention, directories are typically distributed across the tiles, with the *directory placement policy* deciding the mapping from the address to the corresponding *home* directory; we assume the *first touch* directory placement policy. With a distributed directory, a write-miss (or a read-miss) m_1 is issued to its home directory and can only obtain the list of pending writes from *that* directory. This means that a memory operation m_2 that follows m_1 can use the write-list to check for conflicts only if m_2 maps to the same home directory as m_1. If m_2 does not map to the same home node as m_1, then m_2 will have to be conservatively assumed to be conflicting. Consequently m_2 will have to go to its own home directory to ensure that it does not conflict. If it does not conflict, then m_2 can simply commit like a cache hit; otherwise, it is treated like a miss and replayed. When

m_2 goes to its home directory to check for conflicts, we take this opportunity to fetch the write-list from m_2's home directory, so that future accesses to the same node can check for conflicts locally. To accommodate this, the local processor has multiple write-list registers which are tagged by the tile id.

The scenario shown in Fig. 2.3(b) illustrates this. Let us assume that the variable X maps to Node 1, while variables Z, W and Y map to Node 2. Note that the processor also has two write-list registers. When a_1 is issued to its directory, it returns the pending write-misses from Node 1. Consequently, this is put into one of the write-list register which is tagged with the id of Node 1. The contents of the other write-registers are invalidated as they might contain out-of-date data. When a_2 tries to commit, the tags of the write-list registers are checked to see if any of the write-list registers is tagged with the id of Node 2, which is the home node for a_2. Since none of the write-list registers have the contents of Node 2, a_2 is sent to its home directory (Node 2) to check for conflicts. After ensuring that a_2 does not conflict, the pending stores of the home directory (from Node 2) are returned and inserted into the write-list register. This allows us to commit a_3 which maps to the same node. Likewise, when a_4 tries to commit, it is found to be conflicting and hence replayed.

Although the distributed directory could potentially reduce the number of memory operations that can complete past pending writes, the experiments show that, due to locality, most of the nearby accesses tend to map to the same home node, which allows us to complete most memory operations past pending writes.

2.3.3 Correctness

The correctness of conflict ordering implementation hinges on the fact that when a cache-miss m_1 is issued to the directory, the write-list returned by the directory is correct; that is, the write-list should include $addr_p(m_1)$, the addresses of all pending memory operations that occur before m_1 in the global memory order. We prove this formally, next.

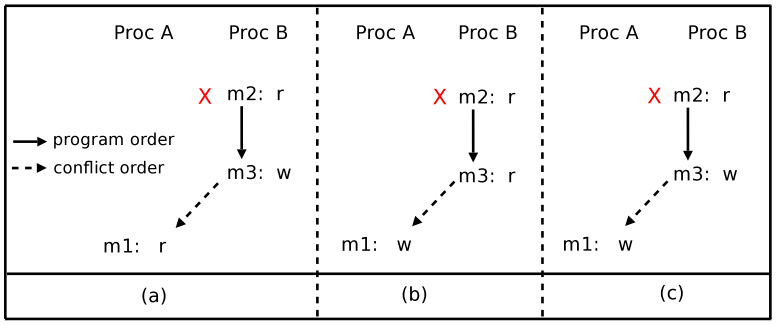

Figure 2.4: Correctness of conflict ordering implementation: m_2 cannot be a read in each of the 3 scenarios.

Lemma 2. *When a cache-miss m_1 is issued to the directory, all pending memory operations prior to it in the global memory order must be present in the directory . That is, when m_1 is issued to the directory, $\{m : m\mathbf{G}m_1\}$ must be present in the directory.*

Proof. (1) m_1 is a write-miss. When m_1 is issued to the directory, conflict ordering ensures that all pending memory operations prior to m_1 in the program order have been issued to the directory. (2) m_1 is a read-miss. m_1 is issued to the directory to fetch the write-list only when it is retired, and all memory operations prior to it have in turn been issued. Thus, whether m_1 is a write-miss or a read-miss, when m_1 is issued to the directory, $\{m : m\mathbf{P}m_1\}$ must be present in the directory. Likewise, when m_1 is issued to the directory all memory operations prior to m_1 in the conflict order, $\{m : m\mathbf{E}m_1\}$ must have been issued to the directory. Thus, when m_1 is issued to the directory, $\{m : m\mathbf{G}m_1\}$ is in the directory. □

Lemma 3. *When a read-miss m_1 is issued to the directory (to obtain the write-list), none of the pending memory operations prior to it in the global memory order are read-misses.*

Proof. Let us attempt a proof by contradiction and assume that such a pending read-miss exists. Let m_2 be the assumed read-miss such that $m_2 \in pred_p(m_1)$. Now the read-miss m_2 cannot occur before m_1 in the program order, as all such read-misses would have retired and hence cannot be pending. m_2 cannot occur before m_1 in the conflict order, as two reads do not conflict with each other. Thus, the only possibility is as shown in Fig. 2.4(a), where there is a write-miss m_3 such that $m_2\mathbf{P}m_3$ and $m_3\mathbf{E}m_1$ (without loss of generality). This again, however, is impossible since the write m_3 will be issued only after all reads before it (including m_2) complete. □

Lemma 4. *When a write-miss m_1 is issued to the directory, all pending read-misses prior to it in the global memory order, conflict with m_1.*

Proof. As shown in Fig. 2.4(b), it is possible that there exists m_3, a read-miss, such that $m_3\mathbf{G}m_1$ since $m_3\mathbf{E}m_1$. There cannot be, however, any read-miss m_2 such that $m_2\mathbf{G}m_1$ with m_2 not conflicting with m_1. Proof is by contradiction, along similar lines to Lemma 3 (m_2 in Fig. 2.4 (b) and Fig. 2.4 (c) cannot be reads). □

Theorem 2. *When a cache-miss m_1 is issued to the directory, the write-list returned will contain $addr_p(m_1)$, the addresses of all pending memory operations that occur before m_1 in the global memory order.*

Proof. When a cache-miss m_1 is issued to the directory, the addresses of the memory operations serviced in the directory (addr-list) will contain $addr_p(m_1)$ (from Lemma 2). All

pending read-misses which occur before m_1 in the global memory order, if any, would conflict with m_1 and hence access the same address as m_1. (from Lemma 3 and Lemma 4). Thus, the write-list is guaranteed to contain $addr_p(m_1)$. □

2.4 Enhanced Hardware Design

This section describes the enhanced conflict ordering design, in which we identify phases in program execution where memory operations can complete without checking for conflicts. But first, we discuss the limitations of basic conflict ordering, to motivate the enhanced design.

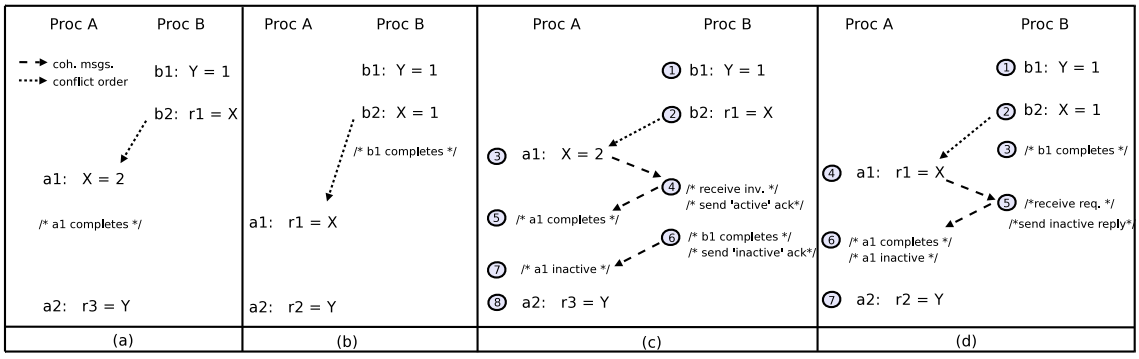

Figure 2.5: (a) and (b): Limitations of basic conflict ordering; (c) and (d): Enhanced conflict ordering.

2.4.1 Limitations of basic conflict ordering

We illustrate the limitations of basic conflict ordering with the examples shown in Fig. 2.5(a) and (b). As we can see, in Fig. 2.5(a), the write-miss a_1 estimates $pred_p(a_1)$ by consulting the directory and obtaining the write-list. All memory operations that follow the pending write-miss a_1 need to be checked with the write-list for a conflict, before they can complete. It is important to note that even after a_1 completes, memory operations

following a_1 will still have to be checked for conflicts. This is because b_1, which precedes a_1 in the global memory order, could still be pending when a_1 completes. In other words, the completion of the store a_1 is no longer an indicator of all memory operations belonging to $pred(a_1)$ completing. Thus, in basic conflict ordering, all memory operations that follow a write miss (or a read miss) need to be continually checked for conflicts before they can safely complete.

Another limitation stems from the fact that a read-miss (or a write-miss) *conservatively* estimates its predecessors in the global memory order by accessing the directory. Let us consider the scenario shown in Fig. 2.5(b), in which the read-miss a_1 estimates $pred_p(a_1)$ from the directory. However, if b_2 and a_1 are sufficiently staggered in time, which is common [102], then memory operations belonging to $pred(b_2)$ (including b_1) might have already completed by the time a_1 is issued. In such a case, there is no need for a_1 to compute its write list; indeed, memory operations such as a_2 can be safely completed past a_1.

2.4.2 Enhanced conflict ordering

(HW support and operation) The approach is to keep track of the (local) memory operations that have retired from the ROB, but whose predecessors in the global memory order are still pending. We call such memory operations as *active* and we track such memory operations in a per-processor *augmented write buffer* (AWB). The AWB is like a normal write-buffer, in that, it buffers write-misses; unlike a conventional write-buffer, however, it also buffers the addresses of other memory operations (including read-misses, read-hits and write-hits) that are active. Therefore, an empty AWB indicates an executing phase in which all preceding memory operations in the global memory order have completed; this allows

us to complete succeeding memory operations without checking for conflicts. To reduce the space used by AWB, the consecutive cache hits to the same block are merged.

We now explain how we keep track of active memory operations in the AWB. A write-miss that retires from the ROB is marked active by inserting it into the AWB, as usual. The write-miss, however, is not necessarily removed from the AWB (i.e. marked *inactive*) when it completes; we will shortly explain the conditions under which a write-miss is removed from the AWB.

A memory operation which retires from the ROB while the AWB is non-empty is active by definition. Consequently, a cache hit which retires while the AWB is non-empty is marked active by inserting its cache block address into the AWB; it is subsequently removed when the memory operation becomes *inactive* – i.e., when all prior entries in the AWB have been removed.

A cache miss which retires while the AWB is non-empty, like a cache hit, is marked active by inserting its cache block address into the AWB. However, unlike a cache hit, its block address is not necessarily removed when all prior entries in the AWB have been removed. Indeed, a cache miss remains active, and hence its cache block address remains buffered in the AWB, until its predecessors in the conflict order become inactive. Accordingly, even if a write-miss completes, it remains buffered in the AWB until all the cache blocks that it invalidates turn inactive. Likewise, a read-miss that completes is inserted into the AWB, if it gets its value from a cache block that is marked active. Here, a cache block is said to be active if the corresponding cache block address is buffered in the AWB, and inactive otherwise.

Thus, to precisely keep track of cache misses that are active, we need to somehow infer whether its predecessors in the conflict order are active. We implement this by tagging coherence messages as active or inactive. When a processor receives a coherence request for a cache block, we check the AWB to see if it is marked active. If it is marked active, the processor responds to this coherence request with a coherence response that is tagged active. This would enable the recipient (cache miss) to infer that the coherence response is from an active memory operation. At the same time, when a processor receives a coherence request to an active cache block, it is buffered; when the cache block eventually becomes inactive, we again respond to the buffered coherence request with a coherence response that is tagged *inactive*. This would enable the recipient (cache miss) to infer that the response is from a block that has transitioned to inactive. A cache miss, when it receives a coherence response, is allowed to complete irrespective of whether it receives an active or an inactive coherence response; however, it remains active (and hence buffered in the AWB) until it gets an inactive response. Finally, by not allowing an active cache block to be replaced, we ensure that if a cache miss has predecessors in the conflict order, it will surely be exposed via the tagged coherence messages.

(Write-miss: Example) We now explain the operation with the scenario shown in Fig. 2.5(c) which addresses the limitation shown in Fig. 2.5(a). First, the write-miss b_1 is issued from the ROB of processor B and is inserted into the AWB (step 1), after which the read-hit b_2 is made to complete past it. Since, the AWB is non-empty when b_2 completes, b_2 is marked active by inserting the cache block address X into the AWB (step 2). Then, write-miss a_1 to same address X is issued in processor A, inserted into the AWB, and issued to the directory (step 3). The directory then services this write-miss request, sending an invalidation request

for cache block address X to processor B. Since cache block address X is active in processor B (cache block address X is buffered in processor B's AWB), processor B sends an active invalidation acknowledgement back to processor A (step 4); at the same time, processor B buffers the invalidation request so that it would be able to respond again when cache block address X eventually becomes inactive. When processor A receives the invalidation acknowledgement, a_1 completes (step 5), and so it sends a completion acknowledgement to the directory. It is worth noting that a_1 has completed at this point, but is still active and hence is still buffered in processor A's AWB. Let us assume that b_1 completes at step 6; furthermore let us assume that at this point all the predecessors of b_1 in the global memory order have also completed – in other words, b_1 has becomes inactive. Since b_1 has become inactive, cache block address Y is removed from the AWB. This, in turn, causes b_2 (cache block address X) to be removed from the AWB, because all those entries that preceded b_2 (including b_1) have been removed from the AWB. Since cache block address X has become inactive, processor B again responds to the buffered invalidation request by sending an inactive invalidation acknowledgement to processor A. When processor A receives this inactive acknowledgement, the recipient a_1 is made inactive and hence removed from the AWB (step 7). This will allow succeeding memory operations like a_2 to safely complete without checking for conflicts (step 8).

(Read-miss: Example) Fig. 2.5(d) addresses the limitation shown in Fig. 2.5(b). First, the write-miss b_1 is issued and made active by inserting it into the AWB (step 1). Then, the write-hit b_2 which completes past it is made active by inserting address X into the AWB (step 2). Let us assume that b_1 then completes, and also becomes inactive (step 3). This causes b_1 to be removed from the AWB, which in turn causes b_2 (cache block address X) to

be removed. When the read-miss a_1 is issued to the directory (step 4), it sends a data value request for address X to processor B. Since the cached block address X is inactive, processor B responds with an inactive data value reply to processor A (step 5). Upon receiving this value reply, a_1 completes. Furthermore, since the reply is inactive, it indicates that all the predecessors of a_1 in the conflict order have completed. Thus, the write-list is not computed and a_1 is not inserted into the AWB. Indeed when a_2 is issued (step 8), assuming the AWB is empty, it can safely be completed.

(Misses to uncached blocks) When a cache-miss is issued to the directory and it is found to be uncached in each of the other processors, this indicates that the particular cache block is inactive in each of the other processors. This is because an active block is not allowed to be replaced from the local cache. This allows us to handle a miss to an uncached block similar to a cache hit, in that, we do not need to compute a new write-list for such misses. Indeed, if there are no other pending entries in the AWB, then memory operations that come after a miss to an uncached block can be completed without checking for conflicts. It is worth noting that misses to local variables, which account for a significant percentage of misses, would be uncached in other processors. This optimization would allow us to freely complete memory operations past such local misses.

(Avoiding AWB snoops) In the above design, all coherence requests and replacement requests must snoop the AWB first to see if the corresponding cache block is marked active. To avoid this, we associate an *active bit* with every cache block; the active bit is set whenever the corresponding cache block address is inserted into the AWB and reset, whenever the cache block address is not contained in the AWB.

(Deadlock-freedom) Deadlock-freedom is guaranteed because memory operations that have been marked active will eventually become inactive. It is worth recalling that a memory operation becomes inactive when all its predecessors in the global memory order in turn become inactive. In theory, since conflict ordering guarantees SC, there cannot be any cycles in the global memory order, which ensures deadlock-freedom. The fact that an active cache block is not allowed to be replaced, however, can potentially cause the following subtle deadlock scenario. In this scenario, an earlier write-miss is not allowed to bring its cache block into its local cache, since all the cache blocks in its set have been marked active by later memory operations which have completed. In such a scenario, the later memory operations that have completed are marked active, and are waiting for the earlier write-miss to turn inactive; the earlier write-miss, however, cannot complete (and become inactive), since it is waiting for the later memory operations to turn inactive. We avoid such a scenario by forcing a write-miss to invalidate the cache block chosen for replacement and set its active bit, before the write-miss is issued to the directory. This will ensure that later memory operations which complete before the write-miss will not be able to use the same cache block used by the write-miss, as this cache block has been marked active by the write-miss. Thus, the deadlock is prevented.

2.4.3 Hardware summary

(Hardware support) Fig. 2.6 summarizes the hardware support required by conflict ordering. First, each processor core is associated with a set of write-list registers. Second, each processor core is associated with an augmented write buffer (AWB), which replaces a conventional write-buffer. Like a conventional write-buffer, each entry of the AWB is tagged

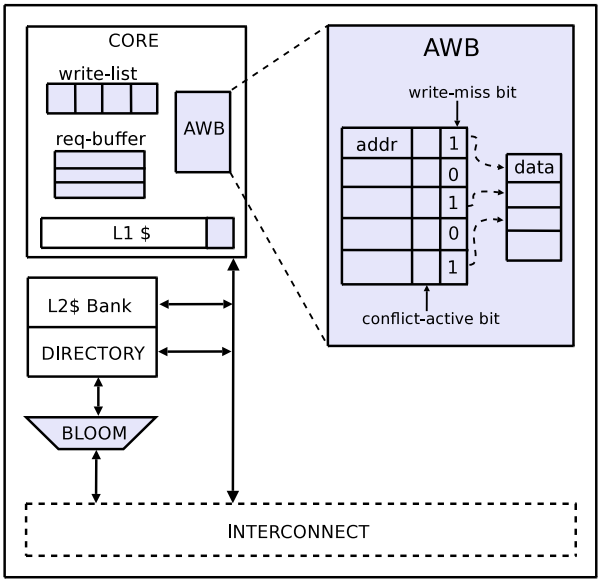

Figure 2.6: Hardware support: conflict ordering.

by the cache block address; unlike a conventional write buffer, however, it also buffers read-misses, read-hits and write-hits. To distinguish write-misses from other memory operations, each entry of the AWB is associated with a *write-miss bit*. If the current entry corresponds to a write-miss, i.e. the write-miss bit is set, the entry additionally points to the data that is written by the current write-miss. Each entry is also associated with a *conflict-active* bit, which is set if any of its predecessors which conflict with it are active. Third, an *active* bit is added to each local cache block, which indicates whether that particular cache block is active. Lastly, conflict ordering requires minor extensions to the cache coherence subsystem. Each processor is associated with a *req-buffer* which is used to buffer coherence requests to active cache blocks. Each coherence response message (data value reply and invalidation acknowledgement) is tagged with an active bit, to indicate whether it is an active response or an inactive response. Also, conflict ordering involves the exchange of additional coherence message. While additional coherence messages are exchanged as part of conflict ordering, it

is worth noting that the coherence protocol and its associated transactions for maintaining coherence are left unchanged. We summarize the additional transactions next.

(Write-miss actions) The actions performed when the write-miss retires from the ROB are as follows:

- Insert into the AWB. The write-miss is first inserted into the AWB as follows: the inserted entry is tagged with the cache block address of the write-miss; since the entry corresponds to a write-miss, the write-miss bit is set to 1 and the entry points to the data written by the write-miss; the conflict-active bit is set to 1 since its predecessor in the conflict order may be active.

- Invalidate replacement victim. Once the write-miss is inserted into the AWB, the cache block chosen for replacement is invalidated and its active bit is set to 1 – it is worth recalling that this is done to prevent the deadlock scenario discussed earlier.

- Issue request to home-directory. Now, the write-miss is issued to its home directory. If the corresponding cache block is found to be uncached in any of the other processors, the directory replies with an empty write-list. If the cache block is indeed cached in some other processor, the directory replies with the list of pending write-misses being serviced in the directory. To minimize network traffic, before sending the write-list, a bloom filter is used to compress the list of addresses.

- Process response. Once the processor receives the write-list, if it indeed receives a non-empty write-list, it updates the local write-list register. When the write-miss receives all of its invalidation acknowledgements, it completes. When the write-miss completes, it sends a completion message to the directory, so that the directory knows about its

completion. When a write-miss receives all of its acknowledgements and each of the acknowledgements are tagged inactive, its corresponding conflict-active bit (in the AWB) is reset to 0. When the conflict-active is reset, the write-miss checks the AWB to see if there are any entries prior to it. If there are none, it implies that the write-miss has become inactive and can be removed from the AWB.

- Remove write-miss and other inactive entries from the AWB. Before removing the write-miss entry, we first identify those memory operations which follow the original write-miss that have also become inactive. To identify such inactive memory operations, the AWB is scanned sequentially starting from the original entry, selecting all those entries whose conflict-active bit is reset to 0; the scanning is stopped when an entry whose conflict-active bit is set to 1 is encountered. All such selected entries are removed from the AWB, and the active bits of their respective cache blocks are reset to 0.

(Cache-hit actions) The actions performed when a cache-hit reaches the head of the ROB are as follows:

- AWB empty. The AWB is first checked to see if it is empty. If the AWB is empty, the cache-hit completes without checking for conflicts.

- Check for conflicts. If the AWB is not empty, the write-list registers are checked to see if the cache-hit conflicts with it. If the cache-hit conflicts (or if write-list register does not cache the directory entries of the cache-hit's home node), the cache-hit is treated like a miss and is issued to its home directory.

- No conflicts. If the cache-hit does not conflict, it is allowed to safely complete. Since the AWB is non-empty, the cache-hit is marked active by inserting it into the AWB with

the write-miss bit set to 0, the conflict-active bit set to 0 (since it is a cache-hit, its predecessors in the conflict order must have completed), and the active bit of the cache block is set to 1.

(Read-miss actions) The actions performed when a read-miss reaches the head of the ROB are as follows:

- AWB empty. The AWB is first checked to see if it is empty. If the AWB is empty, the read-miss completes without checking for conflicts. Then, the data value reply that the read-miss received as a coherence response is examined. If it is tagged active, it is inserted into the AWB with the write-miss bit set to 0, the conflict-active bit set to 1 (since it obtained an active response), and the active bit of the cache block is set to 1. Later, when the read-miss receives its inactive response, the read-miss entry is removed along with subsequent inactive entries, as discussed earlier.

- Check for conflicts. If the AWB is not empty, the write-list registers are checked to see if the read-miss conflicts with it. If the read-miss conflicts (or if write-list register does not cache the directory entries of the read-miss' home node), the cache-miss is re-issued to its home directory.

- No conflicts. If the read-miss does not conflict, the read-miss is allowed to safely complete. Since the AWB is non-empty, the read-miss is marked active by inserting into the AWB with the write-miss bit set to 0. The conflict-active bit is set to 0 or 1 depending on whether the read-miss received an inactive or an active data value response, respectively. If it received an active response – later, when the read-miss receives the inactive response, the read-miss entry is removed along with subsequent inactive entries, as discussed earlier.

(Coherence and replacement requests) When a coherence request (invalidate or data value request) is received for a cache block that is marked active/inactive, the coherence response is in turn tagged active/inactive. Additionally, if the coherence request is for an active cache block, the coherence request is buffered in the req-buffer. When the cache block is eventually reset to inactive, the corresponding entry is removed from the req-buffer and the processor again responds to the buffered request – but now with an inactive response. When a coherence request is received for an active cache block and the req-buffer is full, the coherence response is delayed until a space in the req-buffer frees up. As discussed earlier, this cannot cause a deadlock, since active cache blocks will eventually turn inactive. Finally, a cache block that is marked active is not allowed to be replaced. When a replacement request is received, and all cache blocks in the set are marked active, the response is delayed until once of the blocks in the set turns inactive – again, without the risk of a deadlock.

2.5 Experimental Evaluation

We performed experiments with several goals in mind. First and foremost, we want to evaluate the benefit of ensuring SC via conflict ordering in comparison with the baseline SC and RMO implementations. We then study the effect of varying the values of the parameters in the HW implementation, on the performance. Since the performance of conflict ordering is dependent on how fast requests can get back replies from directories, we study the sensitivity towards the network latency. We also vary the size of write-lists to evaluate their effects on performance. We then study the characterization of conflict ordering to see how it reduces the overhead of using in-window speculation technique.

Furthermore, we measure the additional network bandwidth that is used up and finally, we also measure the on-chip hardware resources that conflict ordering utilizes. However, before we present the results of the evaluation, we briefly describe the implementation.

2.5.1 Implementation

Processor	8, 16 and 32 core CMP, out of order
ROB size	176
L1 Cache	private 32 KB 4 way 2 cycle latency
L2 Cache	shared 8 MB 8 way 9 cycle latency
Memory	300 cycle latency
Coherence	directory based invalidate
# of AWB entries	50 per core
# of req-buffer entries	8 per core
# of write-list registers	6 per core
write-list size	160 bits
Interconnect	2D torus (2×4 for 8-core, 4×4 for 16-core, 4×8 for 32-core) 5 cycle hop latency

Table 2.1: Architectural parameters.

Benchmark	Inputs
barnes	16K particles
fmm	16K particles
ocean	258× 258
radiosity	batch
raytrace	car
water-ns	512 molecules
water-sp	512 molecules
cholesky	tk15.O
fft	64K points
lu	512×512
radix	1M integers

Table 2.2: Benchmarks.

We implemented conflict ordering using SESC [88] simulator, targeting the MIPS architecture. The simulator is a cycle-accurate, execution-driven multi-core simulator with

detailed models for the processor and the memory systems. To implement conflict ordering, we added the associated control logic to the simulator. We considered conflict ordering in the context of a CMP with local caches kept coherent using a distributed directory based protocol. The architectural parameters for the implementation are presented in Table 2.1. The default architectural parameters were used in all experiments unless explicitly stated otherwise. We measured performance with 8, 16 and 32 processors in Section 2.5.2, and for other studies, we assumed 32 processors. We used the SPLASH-2 [108], a standard multithreaded suite of benchmarks for the evaluation. We could not get the program *volrend* to compile using the compiler infrastructure that targets the simulator and hence we omitted it. We used the input data sets described in Table 2.2 and ran the benchmarks to completion.

(Baseline SC implementation) The SC baseline, referred to as conventional SC in the experiments below, uses in-window speculation support. It is an aggressive implementation that uses hardware prefetching and support for speculation in modern processors to speculatively reorder memory operations while guaranteeing SC ordering using replay. However, such speculation is within the instruction window – where the instruction window refers to the set of instructions that are in-flight. The implementation we use is similar to ones used as SC baselines in proposals such as [16, 49, 107].

(Baseline RMO implementation) The baseline RMO implementation allows memory operations to be freely reordered, enforcing memory ordering only when fences are encountered. Even when fences are encountered, in-window speculation support (as used in the SC baseline) is used to mitigate the delays. The RMO implementation is similar to ones used in recent works such as [16, 49, 107].

2.5.2 Execution time overhead

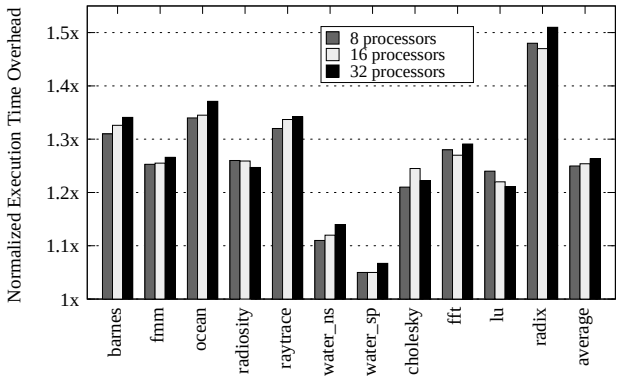

Figure 2.7: Conventional SC normalized to RMO.

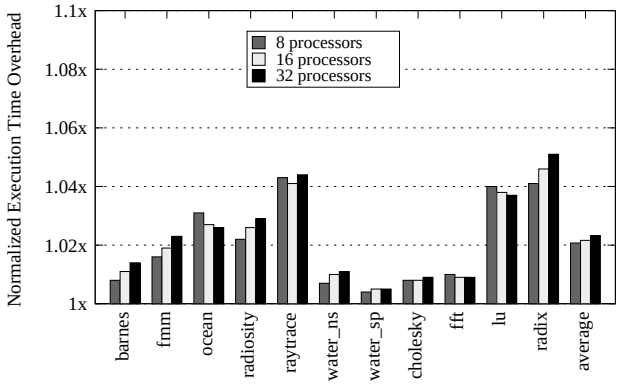

Figure 2.8: Conflict ordering normalized to RMO.

We measure the execution time overhead of ensuring SC via conflict ordering and compare it with the corresponding overhead for conventional SC. We conducted this experiment for 8, 16 and 32 processors using the default hardware implementation presented in Table 2.1. Fig. 2.7 and Fig. 2.8 show the execution time overheads for conventional SC and conflict ordering, respectively. The execution time overheads are normalized to the performance achieved using RMO. As we can see, most benchmark programs experience significant slowdown for conventional SC (more than 20% overhead on average for all num-

bers of processors). In particular, *radix* has the highest overhead. This is because *radix* has a relatively high store-miss rate which forces the following loads to wait longer before they can be retired. As we can see, with conflict ordering the overhead of ensuring SC is significantly reduced. On average, the overhead is just 2.0% for 8 processors, 2.2% for 16 processors and 2.3% for 32 processors, and thus the performance of conflict ordering is comparable to RMO. With conflict ordering loads and stores do not need to wait until outstanding stores complete; they can mostly retire as soon as the pending stores get replies back from directories. Since the time to get replies from directories is significantly less than the time for outstanding stores to complete, loads do not end up causing stalls and stores can also be performed out of order. This explains why the performance with conflict ordering is significantly better than conventional SC. Furthermore, it is worth noting that, with different numbers of processors, the performance does not vary significantly. Therefore, conflict ordering appears scalable.

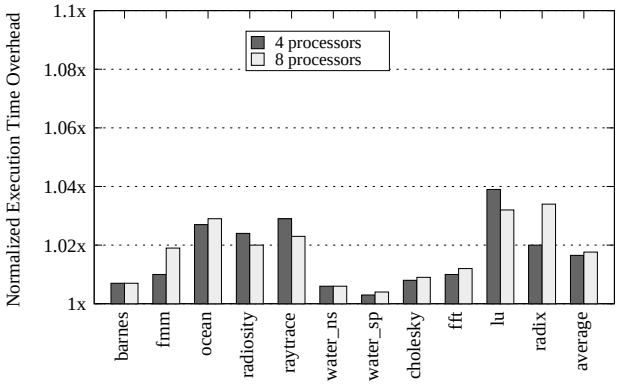

Figure 2.9: Performance for bus-based implementation.

(Bus-based implementation) In addition to the distributed directory-based implementation, we also evaluated a bus-based implementation to find out if conflict ordering is

applicable to current multi-core processors which are predominantly bus based. For the bus based implementation, we just implement basic conflict ordering; we maintain a centralized on-chip structure called *write-list-buffer* (WLB), which records the addresses of the write-misses which are currently pending. When a write-miss retires into a local write-buffer it is sent to the WLB; upon receiving the write-miss, the WLB inserts its address into the WLB and replies back with the write-list – containing all the addresses that are currently present in the WLB except the addresses from the source processor. Similar to the directory-based implementation, a bloom filter is used to compress the addresses. Memory operations which attempt to complete past the write-miss, check the received write-list to decide whether they can be completed safely, without violating SC. We conducted experiments for bus-based implementation with 4 and 8 processors, and set the round-trip latency for accessing WLB as 5 cycles. Fig. 2.9 shows the execution time normalized to RMO. As we can see, the execution time of conflict ordering is close to RMO for both 4 and 8 processors, with the overhead less than 2% on average. This shows that conflict ordering is also applicable to small scale multi-core processors that use a bus for coherence.

2.5.3 Sensitivity study

(Sensitivity towards network latency) The performance of conflict ordering hinges on the time for a miss to get replies from the directory. It is desirable that conflict ordering is reasonably tolerant to the network latency. In the experiments, we used 2D torus network and varied the latency of each hop with values of 3 cycles, 5 cycles, and 8 cycles, as shown in Fig. 2.10. The performance is measured with 32 processors. As we can see, the overhead does not vary significantly when the latency is increased from 3 cycles to 5 cycles. Even

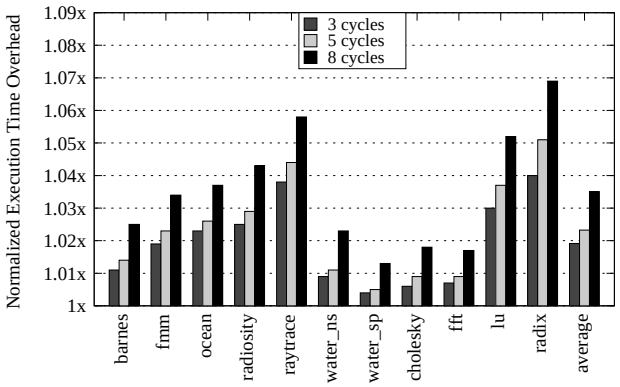

Figure 2.10: Varying the latency of network.

when the latency is increased to 8 cycles, the average performance drops only slightly, to 3.5%. This shows conflict ordering is reasonably tolerant to the network latency. In the default design, we choose 5 cycles as each hop latency, assuming 2 cycle wire delay between routers and 3 cycle delay per pipelined router.

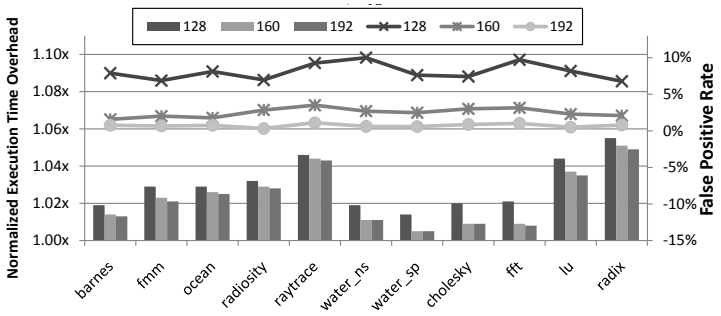

Figure 2.11: Varying number of bits of write-list.

(Sensitivity towards size of write-list) Recall that, to minimize network bandwidth, the addresses within replies to cache miss requests are compressed to form a write-list using a bloom filter. While a smaller size is beneficial as far as saving network bandwidth is concerned, it could also result in false positives. A false positive can lead us to falsely conclude that a load or store conflicts with a pending store, causing the load to re-execute or the store to stall. In this experiment, we evaluated the minimum size of the write-list that

does not result in performance loss. We varied the number of bits of write-list with the value 128, 160, and 192 to evaluate the performance and corresponding false positive rates with 32 processors. In the implementation, we used a bloom filter with 4 hash functions. Fig. 2.11 shows the results, where lines represent false positive rates and bars represent execution time overheads. As the number of bits increases from 128 to 192, the false positive rate decreases (on average, 8.01%, 2.42%, and 0.69% respectively), and the execution overhead also decreases (on average, 2.97%, 2.33% and 2.29% respectively). A size of 160 bits performs slightly better than 128 bits and very close to 192 bits. Therefore, we choose 160 bits (20 bytes) as the size of the write-list in the implementation.

2.5.4 Characterization of conflict ordering

In this experiment, we wanted to examine how conflict ordering reduces the overhead of using conventional SC. Recall that, in the conventional SC implementation, loads cannot be retired if there are pending stores and stores cannot be performed out of order, which leads to the gap between the performance of SC and RMO. On the other hand, using conflict ordering, we can safely retire loads and complete stores past pending stores by checking the write-list. Hence, performance hinges on how often loads and stores can be reordered with their prior pending stores.

Fig. 2.12 shows the breakdown for all checks against write-lists. We categorize these checks into three types: checks against empty write-lists, checks against non-empty write-lists without finding any conflict, and checks against non-empty write-lists with finding a conflict. When a request to the directory finds that the targeted block is not cached, it indicates that no active memory operation has accessed the block and the requesting

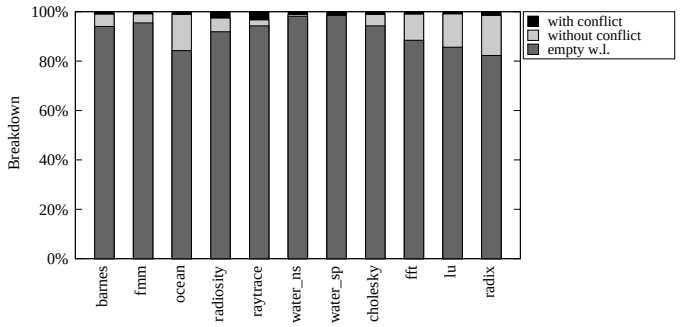

Figure 2.12: Breakdown of checks against write-lists.

processor can get a reply with an empty write-list. As we can see, most checks are against empty write-lists (around 90% on average). For the checks against non-empty write-lists, there is almost no conflict. Hence, conflict ordering allows almost all memory operation to be reordered, achieving performance comparable to RMO.

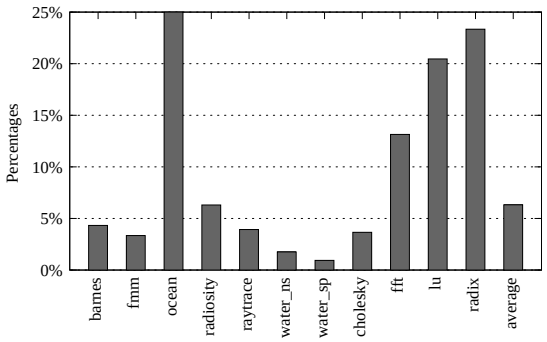

Figure 2.13: Reduced accesses to directories.

Fig. 2.13 shows the reduced directory accesses using enhanced conflict ordering, compared to basic conflict ordering. The performance of conflict ordering also hinges on the frequency of directory accesses. Hence, it is important to have fewer directory accesses. As we can see, using enhanced conflict ordering, on average we only have about 6% directory accesses of basic conflict ordering. This is because, for most benchmark programs, AWB is empty most of the time. For some benchmarks, such as *ocean*, the access frequency does not

reduce as much as other benchmarks. However, the access frequency for these benchmarks in basic conflict ordering is already low. Therefore, their overhead is still low.

2.5.5 Bandwidth increase

Benchmark	Bandwidth increase (%)
barnes	2.49
fmm	0.98
ocean	2.64
radiosity	8.36
raytrace	1.21
water-ns	2.08
water-sp	2.21
cholesky	0.24
fft	1.16
lu	1.96
radix	1.71

Table 2.3: Bandwidth increase.

In this experiment, we measure the bandwidth increase due to write-lists that need to be transferred and extra traffic introduced by conflict ordering for 32 processors. Table 2.3 shows the bandwidth increase compared to RMO. As we can see, for most benchmarks, the bandwidth increase is less than 3% (2.27% on average). *radiosity* has relatively higher bandwidth increase. This is because its write miss rate is relatively higher and requests to directories usually get non-empty write-lists, resulting in write-lists taking up a relatively high proportion of bandwidth.

2.5.6 HW resources utilized

Recall that the additional HW resources utilized by conflict ordering are AWB, req-buffers, write-list registers, and the active bits added to each cache block. Each AWB

entry contains the cache block address, the conflict-active and the write-miss bits; we do not count the storage required for data written by the write-miss, since it is already present in conventional write buffers. Since we use 50 AWB entries per core, each of size 5 bytes, the total size per processor core amounts to 250 bytes for the AWB. Since we use 8 entries for the req-buffer, each of size 6 bytes (for storing the cache block address and the processor id), the total size per processor core amounts to 48 bytes. With 6 write-list registers per core, each of size 20 bytes, the total size per processor core amounts to 120 bytes. In addition, we also require active bits to be added to each cache block in the L1 cache. This amounts to an additional 64 bytes of on-chip storage per processor core. Thus the total additional on-chip storage amounts to 482 bytes per core. In addition to this we require the hardware resources needed for in-window speculation which is also required by the SC and RMO baselines. Thus, the additional hardware resources utilized for conflict ordering is nominal.

2.6 Summary

Should hardware enforce SC? Researchers have examined this question for over 30 years, with no definite answers, yet. While there has been no clear consensus on whether hardware should support SC [3, 52], it is important to note, however, that the benefits of supporting SC are widely acknowledged [3]. Indeed, critics of hardware enforced SC, question it based on whether the costs of supporting SC justify its benefits – all prior SC implementations needing to employ aggressive speculation and its associated complexity for supporting SC.

This chapter demonstrates that the benefits of SC can indeed be realized using nominal hardware resources. While prior SC implementations guarantee SC by explicitly

completing memory operations within a processor in program order, we guarantee SC by completing conflicting memory operations, within and across processors, in an order that is consistent with the program order. More specifically, we identify those shared memory dependencies whose ordering is critical for the maintenance of SC and intentionally order them. This allows us to non-speculatively complete memory operations past pending writes and thus reduce the number of stalls due to memory ordering. experiments with SPLASH-2 suite showed that SC can be achieved efficiently, incurring only 2.3% additional overhead compared to RMO.

Chapter 3

Scoped Fence

This chapter resorts to *programming support* to help hardware dynamically eliminate subset of unnecessary memory orderings, and proposes *scoped fence* (S-Fence). Fence instructions used by programmers are usually only intended to order memory accesses within a limited scope. Based on this observation, the concept *fence scope* is introduced to define the scope within which a fence enforces the order of memory accesses. S-Fence enables programmers to express their ordering demands by specifying the scope of fences when they only want to order part of the 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. In particular, this chapter provides two kinds of fence scope (i.e., *class scope* and *set scope*), and describes their programming, compiler and hardware support. S-Fence only makes changes locally in each processor core, without adding inter-processor communication for multiprocessors, thereby S-Fence does not have the issue of scalability.

3.1 Motivation

Fig. 1.7 in Chapter 1 shows a pattern of using concurrent algorithms. The fences used in concurrent algorithms are only supposed to guarantee the correct concurrent accesses to shared data, without being aware of how the accessed data is processed afterwards. The following example illustrates this pattern in detail.

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

7  TASK take(){
8    tail = TAIL − 1;
9    TAIL = tail;
10   FENCE //store−load
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 }
```

Figure 3.1: Simplified Chase-Lev work-stealing queue [27].

Fig. 3.1 shows a simplified C-like pseudo code of Chase-Lev work-stealing queue [27]. 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 Fig. 3.1. 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 [69, 61]. 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 [69]. In addition, there is need for two compare-and-swap instructions: at Line 20 and Line 32.

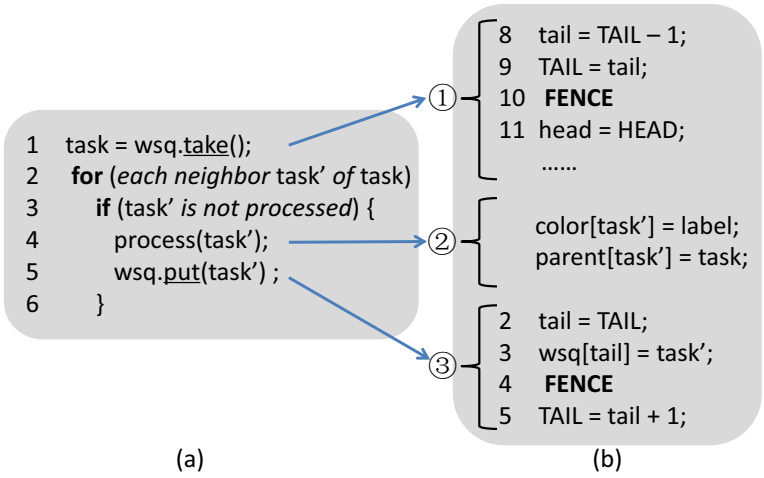

Figure 3.2: Example – parallel spanning tree algorithm.

Let us consider an application of work-stealing queue – parallel spanning tree algorithm, an important building block for many graph algorithms [11, 78]. The parallel spanning tree algorithm is discussed in [11], 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. Fig. 3.2(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 Fig. 3.2(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 [11], 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 Fig. 3.2(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.

3.2 Scoped Fence

This section introduces *scoped fence*, S-Fence for short, that constrains the effect of a fence to a limited scope. The semantics of S-Fence is first defined, followed by the scope of a fence and how programmers can specify the scope.

3.2.1 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 chapter.

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 fence, it is different from other finer fences in current commercial architectures [4], 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.

3.2.2 Scope of a fence

The *scope* of a fence defines the context in which memory accesses should be ordered by the fence. There are 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*. Programming support is also provided to allow programmers to specify the fence scope they want to use, as shown in Fig. 3.3. 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.

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

Figure 3.3: 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 evaluation we consider RMO memory model.

3.3 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, Fig. 3.4 shows the semantics of fences with class scope. The semantics only focuses on the memory operations and fence operations. Let us 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. $next(pc)$ is used to denote the instruction following pc.

The formulation in Fig. 3.4 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.

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

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

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

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

Figure 3.4: 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))*=$\emptyset$.

Fig. 3.5 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 Fig. 3.1. Assume those operations are implemented in a class. To only order data accesses related to work-stealing

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

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

Figure 3.5: An example of class scope.

queue, we can apply class scope to the fences by specifying them as **S-FENCE**[class], which forces the fences to only order memory accesses inside the class. Hence, for the parallel spanning tree algorithm in Fig. 3.2, 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.

3.3.1 Implementation design

Let us 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.

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 3.1.

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

Table 3.1: 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.

Hardware support. Fig. 3.6 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

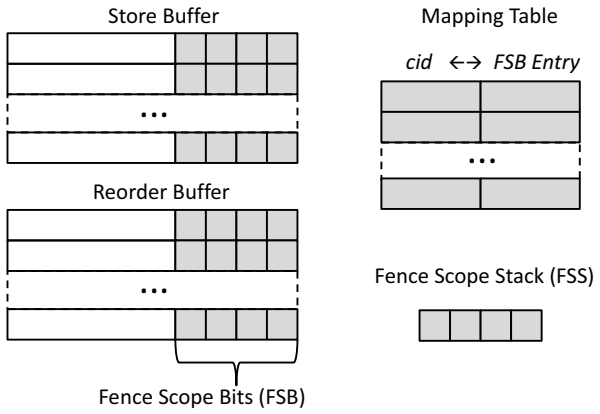

Figure 3.6: Hardware support for class scope.

fence scope bits (FSB) as shown in Fig. 3.6, 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.

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.

The compiler-inserted instructions fs_start and fs_end are utilized to flag memory accesses in the class scope of a fence. Fig. 3.7 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

```
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();
```

Figure 3.7: Micro-operations on fs_start and fs_end.

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.

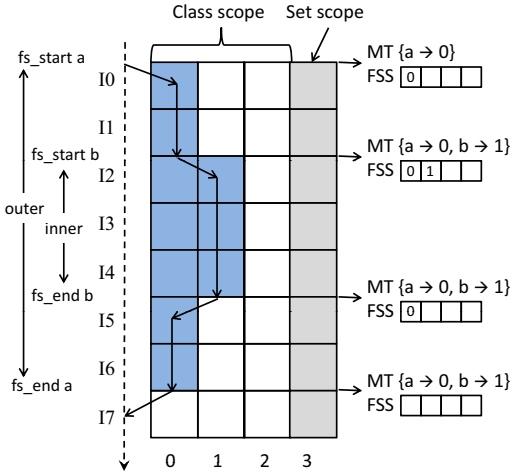

Figure 3.8: Setting fence scope bits.

Fig. 3.8 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.

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.

3.3.2 An example

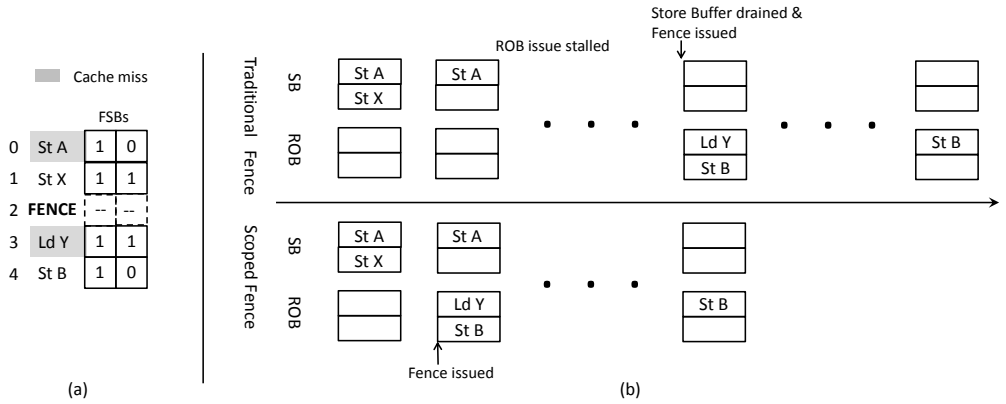

Figure 3.9: Comparison between traditional fence and S-Fence.

Fig. 3.9 depicts an example to illustrate the performance advantages of S-Fence. Fig. 3.9(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. Fig. 3.9(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.

3.4 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, ...*}.

	P0		P1
7	m0 = ...	13	m1 = ...
8		14	
9	flag0 = 1	15	flag1 = 1
10	**FENCE**	16	**FENCE**
11	**if** (flag1 == 0)	17	**if** (flag0 == 0)
12	critical_section	18	critical_section

Figure 3.10: Simplified Dekker algorithm.

For example, Fig. 3.10 shows Dekker's algorithm [34], which is designed to allow only one processor to enter the critical section at a time. The purpose of fences (Lines 10 and 16) 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 7) before the store to flag0 (Line 9), 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 7) is a long latency memory access, the fence (Line 10) does not have to wait for the store to complete. As long as the store to `flag0` (Line 9) has been completed, the fence will allow the following memory accesses to proceed.

3.4.1 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 3.2: The extension of ISA for set scope.

Compiler support. For set scope, we only need the ISA extension shown in Table 3.2. 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.

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 the current 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 Fig. 3.8) 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.

3.5 Using S-Fence

S-Fence is easy for programmers to use. It does not require programmers any additional effort to reason about their programs. The only extra work for programmers is to specify the constraints on fences, which will be conveyed to the compiler and hardware. Class scope is consistent with the principle of the concept of class in object-oriented programming. Class is used to encapsulate the data and functions that manipulate the data, keeping them safe from outside interference and misuse. If one implements an algorithm encapsulated in a class and only wants to use fences to order memory accesses inside the class (e.g., the example in Fig. 3.1), then one can simply use fences with class scope. Moreover, if one only wants to order memory accesses to a certain set of variables (e.g., the example in Fig. 3.10), then one can choose to use fences with set scope providing the set of variables. In other cases, if one does not want to specify the scope of the fence, one can simply use a traditional fence.

On the other hand, when users call functions written by others (e.g., libraries implementing concurrent lock-free algorithms), the correctness of the users' programs should not rely on the order provided by the callees. The users should guarantee the correctness of their parts of the code themselves. In fact, this is also a good programming style, following the principle of modularity in object-oriented programming.

(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 (Fig. 3.1), 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.

3.6 Experimental Evaluation

The goals of the 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 3.3: Architectural parameters.

(Simulation) We implemented S-Fence in the simulator SESC [88] targeting the MIPS architecture. The simulator is a cycle-accurate, execution-driven multicore 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 3.3 shows the default architectural parameters used in all experiments unless explicitly stated otherwise.

Benchmarks	**Type**	**Description**
dekker	set	Dekker algorithm [34]
wsq	class	Work-stealing queue [27]
msn	class	Non-blocking Queue [77]
harris	class	Harris's set [51]
barnes	set	Barnes-Hut n-body [108]
radiosity	set	Diffuse radiosity method [108]
pst	class	Parallel spanning tree [11]
ptc	class	Parallel transitive closure [40]

Table 3.4: Benchmark description.

(Benchmarks) We evaluate the technique using benchmarks in Table 3.4. 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. 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 [11] and parallel transitive closure algorithm [40] using work-stealing queue [27]. *barnes* and *radiosity* are from SPLASH-2 [108], and they are inserted with fences to enforce sequential consistency [95].

3.6.1 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 [20, 69, 61]. We use these applications to obtain a preliminary understanding of the performance of S-Fence. Moreover, the workload between fences may affect the benefit of S-Fence. We developed the harness programs to control the workload. We varied the workload of each task in the applications, from low workload to more expensive computations, to evaluate the performance of S-Fence. We only measured the execution time of the parallel sections in the programs. Fig. 3.11 shows the speedups of S-Fence over traditional fence, where the x axis represents different amounts of computations, from low to high.

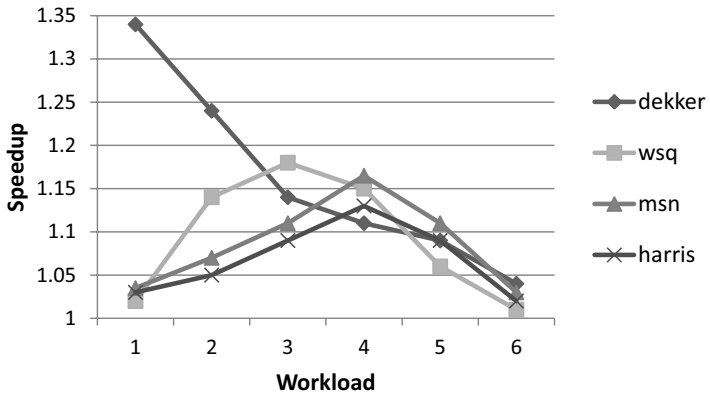

Figure 3.11: Impact of workload.

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 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.

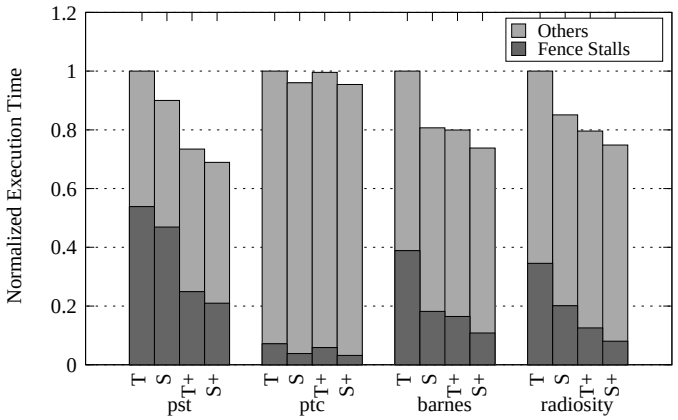

Figure 3.12: Normalized execution time (T – traditional fence; S – S-Fence; $T+$ – traditional fence with in-window speculation; $T+$ – S-Fence with in-window speculation) .

3.6.2 Performance on full applications

We now evaluate the performance of S-Fence on several full applications. Fig. 3.12 shows the normalized execution time of applications with traditional fence and S-Fence, with and without in-window speculation [46]. 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 [11]) and *ptc* (parallel transitive closure [40]) 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 Fig. 3.12, 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 Fig. 3.2) 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 [95]. This can be done by compilers to identify memory pairs which have to be ordered based on delay set analysis [95]. 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 Fig. 3.12, 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 [97], 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 [95]. 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 [46], 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. Fig. 3.12

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.

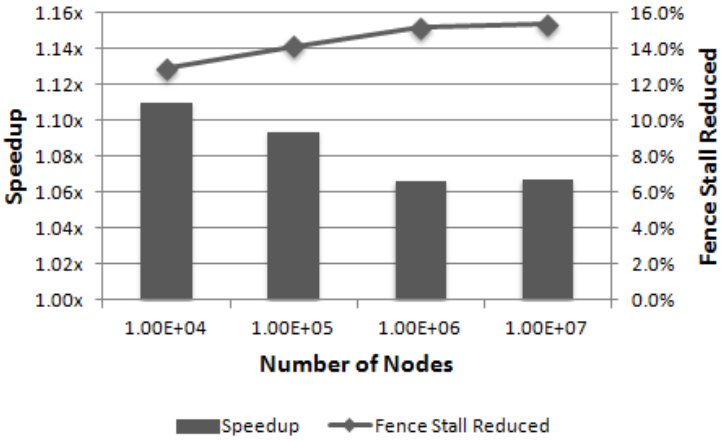

Figure 3.13: Performance on *pst* (parallel spanning tree).

(Input size vs. performance) To better understand the behaviors of S-Fence in *pst*, we varied the input graph with the number of nodes from 10^4 to 10^7, and measured the improvement of S-Fence. Fig. 3.13 shows the results, where the line represents the percentage of reduced fence stalls by using S-Fence and the bars show the execution time speedup. As we can see, S-Fence reduces stalls by 12.9% to 15.3% compared to traditional fence; and *pst* achieves 1.07x to 1.11x speedup in the overall execution time. It is worth noting that, the speedup decreases as the size of graph increases, although the percentage of reduced fence stalls increases slightly. This is because, as the size of graph increases, the existence of the full fence mentioned above results in more stalls, which account for larger portion of the overall execution time. Therefore, we see the decrease on the overall performance speedup using S-Fence.

3.6.3 Class scope *vs.* set scope

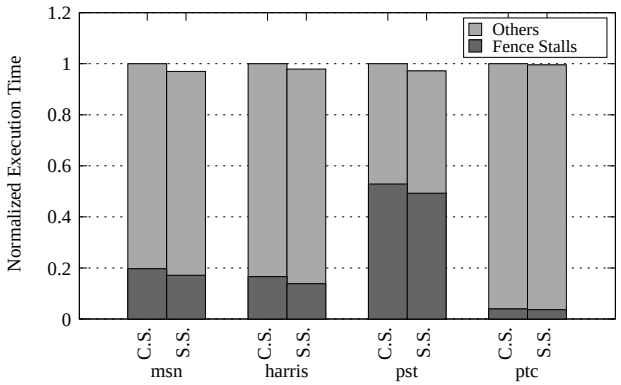

Figure 3.14: Performance comparison between class scope and set scope ($C.S.$ – class scope; $S.S.$ – set scope) .

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. Fig. 3.14 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.

3.6.4 Sensitivity study

This section studies how the architecture parameters affect the performance of S-Fence, including memory access latency and reorder buffer size.

(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,

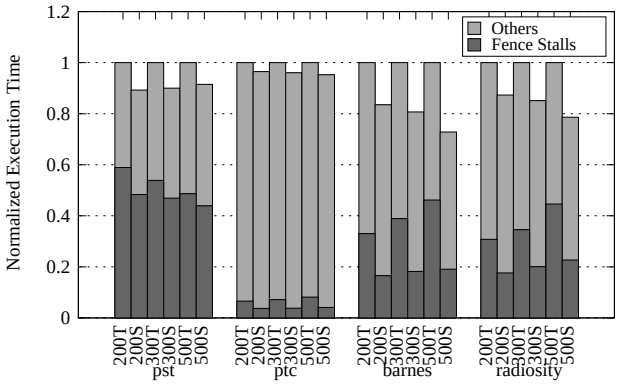

Figure 3.15: Performance for varying memory access latencies.

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 cycles, 300 cycles, and 500 cycles. Fig. 3.15 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 (*200T* and *200S* represent the execution time with 200 cycles latency for traditional fence and S-Fence respectively, and so on). 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.

(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,

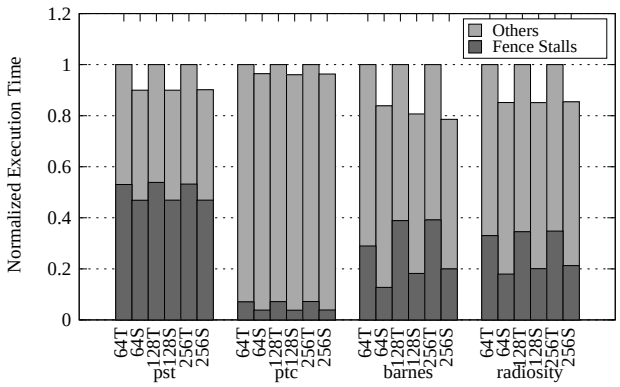

Figure 3.16: Performance with different ROB Sizes.

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 Fig. 3.16, 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. We present the results as in Fig. 3.15 (*64T* and *64S* represent the execution time with 64 entries of ROB for traditional fence and S-Fence respectively, and so on). 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.

3.7 Summary

In this chapter, we proposed 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. Experiments conducted on a group of lock-free algorithms and the other group of full applications 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.

Chapter 4

Conditional Fence

This chapter resorts to *compiler support* to help hardware dynamically eliminate subset of unnecessary memory orderings, and proposes *conditional fence* (C-Fence). The compiler helps to identify *associate fences*, which order the same pair of conflicting memory accesses. Then, at runtime, we can safely execute past a fence, as long as its associates are far away from this fence. The fact that most fence associates are typically staggered at runtime allows most fence execution to be eliminated, resulting in significant performance improvement. For the hardware design, a centralized on-chip structure is first introduced, which is simple and satisfactory for a lower number of processors; then a distributed on-chip structure is designed for a higher number of processors to ameliorate the contention in the centralized structure.

4.1 Fence Order

While most processors do not enforce strict memory ordering automatically, they provide support in the form of *fence instruction* (i.e., memory fence), which forces a strict

ordering between memory accesses that precede it and those following it. A traditional fence enforces this by stalling the processor's execution till all memory accesses encountered before the fence have completed. However, the above approach is more restrictive than what is really required. The purpose of a fence is to prevent memory accesses from being reordered and *observed by other processors*. In other words, if the reordering is not observed by other processors, the reordering is allowed and hence stalling at the fence is not necessary. Let us introduce the following definitions.

Definition 10. *The memory order enforced by a fence is called* fence order.

Definition 11. *A fence execution is said to be* correct if the execution appears to maintain fence orders.

Obviously, the traditional fence execution is correct. However, it is inefficient due to unnecessary stalls. Therefore, to improve the performance, we can relax the execution of a fence such that the execution is correct, maintaining an illusion that fence orders are guaranteed. Next, we provide a *sufficient condition* for a correct fence execution.

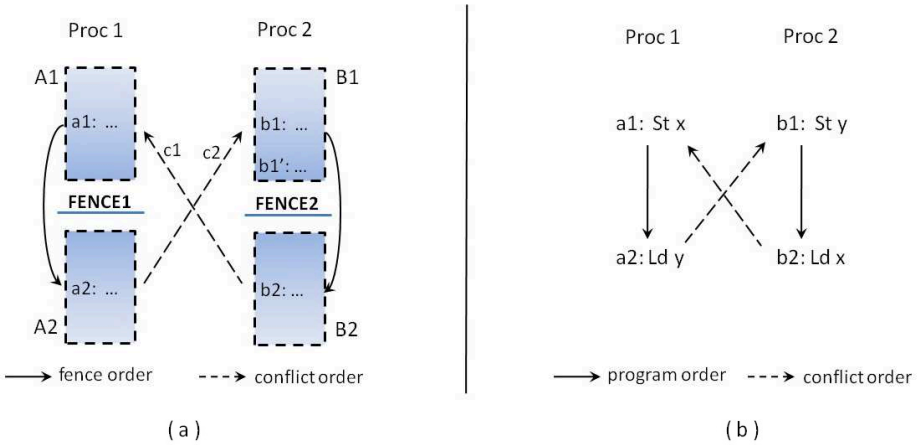

Figure 4.1: (a) Violation of fence order; (b) Violation of program order.

4.1.1 The condition for enforcing fence orders

Consider Fig. 4.1(a), where there are two fences in two processors respectively. $A1, A2, B1$, and $B2$ represent blocks of instructions separated by fences. Furthermore, let us assume $a1$, $a2$, $b1$, and $b2$ are memory accesses in $A1$, $A2$, $B1$, and $B2$ respectively. Due to the fence, any memory access in $A1$ is ordered before any memory access in $A2$, represented by the solid line with arrow; and the same is the case for $B1$ and $B2$. Thus, the fence orders $a1 \rightarrow a2$ and $b1 \rightarrow b2$ are enforced by the fences. A correct fence execution should make these fence orders *appear* to be enforced. In [95], Shasha and Snir have shown the condition for enforcing program orders in context of sequential consistency (SC) enforcement. Extending that condition, we can have the condition for enforcing fence orders, as all fence orders is a subset of all program orders.

Recall that sequential consistency requires that all memory accesses *appear* to take place in the program order which is specified by the program, i.e., all program orders should be enforced. Shasha and Snir [95] have shown that an execution does not violate sequential consistency iff *program orders* ($\mathbf{P}$) and *conflict orders* ($\mathbf{E}$) do not form any cycle (i.e., no cycle in $\mathbf{P} \cup \mathbf{E}$). Here, a *conflict order* is an execution order of conflicting memory accesses [95]. Two memory accesses are said to conflict if they can potentially access the same memory location and at least one of them is a write. We use $\mathbf{C}$ to denote conflict relations, thereby $\mathbf{E}$ is an orientation of $\mathbf{C}$. Fig. 4.1(b) shows such a cycle $a1 \rightarrow a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, where $a1$ conflicts with $b2$ and $a2$ conflicts with $b1$. The execution sequence $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$ will violate sequential consistency, because no sequentially consistent execution can generate the same result, where both x and y read their old values before stores. Intuitively, the cycle indicates that the reordering of $(a1, a2)$ is observed by $(b1, b2)$

in the other processor. However, if any one of the conflict orders (i.e., $a2 \rightarrow b1$ and $b2 \rightarrow a1$) is in the opposite direction, the cycle will be broken and there is no sequential consistency violation.

In the case of fence orders in Fig. 4.1(a), we have to enforce all fence orders (**F**), analogous to program orders for SC in Fig. 4.1(b). In particular, all fence orders is a subset of all program orders ($\mathbf{F} \subseteq \mathbf{P}$). Accordingly, we can have the following condition for enforcing fence orders.

Corollary 1. *Fence orders are enforced iff fence orders (***F***) and conflict orders (***E***) do not form any cycle, i.e., no cycle in* $\mathbf{F} \cup \mathbf{E}$.

This is because, without cycles, $\mathbf{F} \cup \mathbf{E}$ can be extended to a *total order* [31], which indicates all fence orders **F** are enforced. Therefore, in Fig. 4.1(a), to enforce fence orders, we have to prevent such cycles as $a1 \rightarrow a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, assuming ($a1$, $b2$) and ($b1$, $a2$) are conflict relations. Note that, even if there is another memory access $b1'$ after $b1$ in $B1$, and ($a1$, $b1'$) and ($a2$, $b1$) are conflict relations, there is no violation of fence orders involving $a1, a2, b1$, and $b1'$. This is because there is no fence order between $b1$ and $b1'$ and hence there is no cycle formed by fence orders and conflict orders.

4.1.2 Associate fences

The traditional fence execution breaks such cycles by enforcing *delays* between memory operations in the same processor. For example, in Fig. 4.1(a), if the issuing of second instructions in the pairs (a_1, a_2) and (b_1, b_2) is delayed until the first completes, fence orders will be enforced. The execution $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, which violates fence orders, is no longer possible, because now $a2$ can not complete before $a1$.

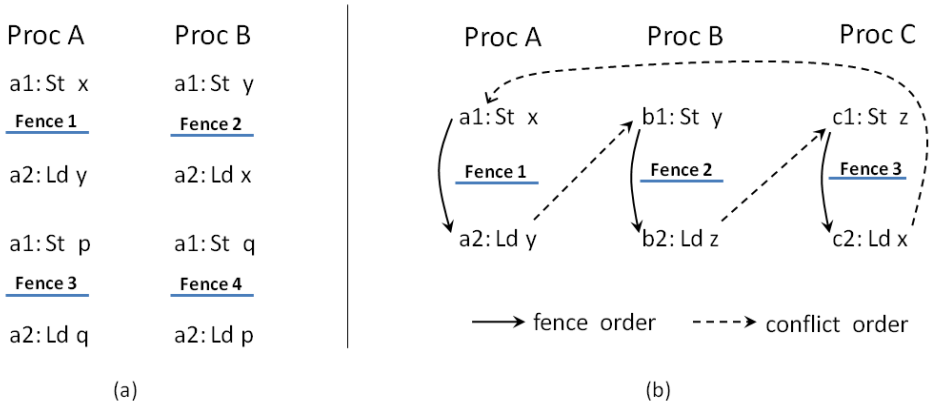

Figure 4.2: Associate fences.

Definition 12. *We call those fences as* **associate fences** *or simply* **associates**, *if they appear in the same cycle in* **F**∪**E**, *ensuring delays between memory accesses involved in the cycle.*

As we can see from Fig. 4.2(a), $Fence_1$ and $Fence_2$ which order conflicting accesses to x and y are associates; similarly, $Fence_3$ and $Fence_4$ are associates. Moreover, a cycle can involve more than two processors [95], in which case more than two fences are associates. Fig. 4.2(b) shows this case, where $Fence_1$, $Fence_2$ and $Fence_3$ are associates.

4.2 Conditional Fence

This section describes *conditional fence* for reducing fence overhead. We first motivate the approach by showing that fences introduced statically may be superfluous dynamically, then describe an empirical study which shows that most fence executions are indeed superfluous. *Conditional fence* is then proposed by taking advantage of this property.

4.2.1 Interprocessor delays to ensure fence orders

While fences in the program are used to ensure required memory orderings, it might be possible to remove the fences dynamically and still maintain an illusion that all required memory orderings are guaranteed. More specifically, we observe that if two fences are *staggered* during the course of program execution, they may not need to stall.

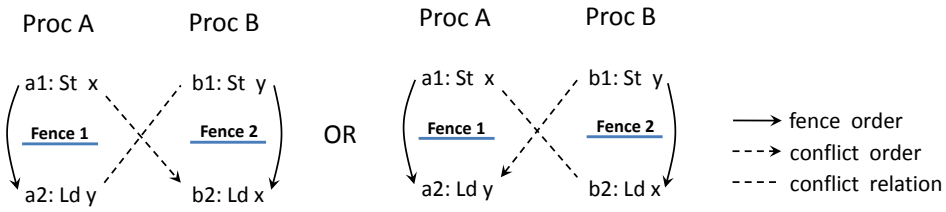

Figure 4.3: Interprocessor delay is able to ensure fence orders.

Let us consider the cycle shown in Fig. 4.1(a), According to the previous definition, $Fence_1$ and $Fence_2$ are associates. Moreover, let us denote delay relation by **D**, in which $u\mathbf{D}v$ indicates that u must complete before v is issued. $Fence_1$, by enforcing the delay $a_1\mathbf{D}a_2$, and $Fence_2$, by enforcing the delay $b_1\mathbf{D}b_2$, are able to break the cycle $\mathbf{F}\cup\mathbf{E}$. While each of the above two *intraprocessor* delays are needed to enforce fence orders, we observe that fence orders can be alternately ensured with just one *interprocessor* delay. More specifically, we observe that if either b_2 is issued after a_1 completes, or if a_2 is issued after b_1 completes, fence orders are ensured. In other words, either $a_1\mathbf{D}b_2$ or $b_1\mathbf{D}a_2$ is sufficient to guarantee fence orders. To see why let us consider Fig. 4.3 that shows the execution ordering resulting from $a_1\mathbf{D}b_2$. With this ordering, we can now see that $\mathbf{F}\cup\mathbf{E}$ becomes acyclic and thus fence orders are ensured. Likewise, $b_1\mathbf{D}a_2$ makes the graph acyclic. *Thus, for every cycle, even if one of these interprocessor delays is ensured, then there is no need*

to stall in either fence. It is easy to prove the correctness by exploring that no cycles will exist when interprocessor delays are enforced.

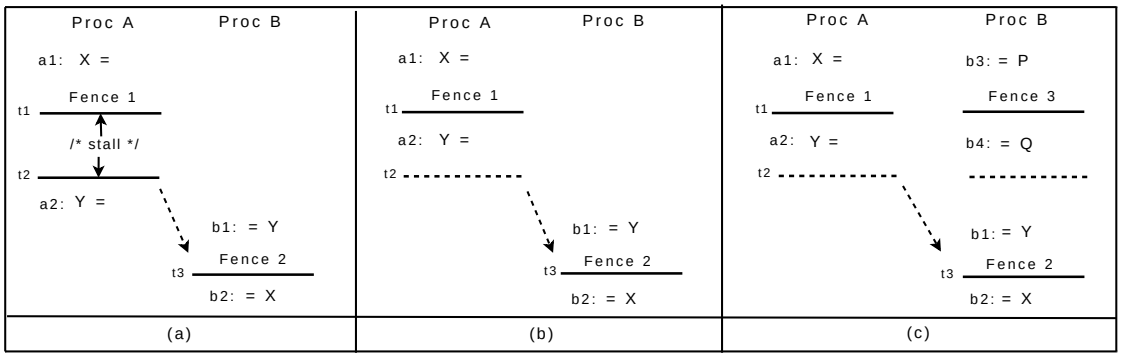

Figure 4.4: (a) If $Fence_1$ has already finished stalling by the time $Fence_2$ is issued, then there is no need for $Fence_2$ to stall. (b) In fact, there is no need for even $Fence_1$ to stall, if all memory instructions before it complete by the time $Fence_2$ is issued. (c) No need for either $Fence_1$ or $Fence_3$ to stall as they are not associates, even if they are executed concurrently.

We now describe execution scenarios in which one of these interprocessor delays is naturally ensured, which obviates the need for stalling at fences. Fig. 4.4(a) illustrates the scenario in which $Fence_1$ has finished stalling by the time $Fence_2$ is issued (indicated by a dashed arrow). This ensures that the write to X would have been completed by the time $Fence_2$ is issued and hence instruction b_2 is guaranteed to read the value of X from $a1$. As we already saw, this interprocessor delay ($b2$ being executed after $a1$) is sufficient to ensure fence orders. As long as $b2$ is executed after $a1$, it does not really matter if $b1$ and $b2$ are reordered. In other words, there is no need for $Fence_2$ to stall. Furthermore, note that even $Fence_1$ needn't have stalled, provided we can guarantee that all memory operations before $Fence_1$ (including $a1$) complete before the issue of $b2$. This is illustrated in Fig. 4.4(b), which shows that all memory operations prior to $Fence_1$ have completed (at time $t2$) before $Fence_2$ is issued (at time $t3$), ensuring that $b2$ is executed after $a1$ completes. Thus, for

any two concurrently executing fence instructions, if all memory instructions prior to the earlier fence complete before the later fence is issued, then there is no need for either fence to stall. In other words, *an interprocessor delay between two fences obviates the need for either fence to stall.*

An interprocessor delay is only necessary for fences that are associates. As shown in Fig. 4.4(c), while $Fence_1$ is enforcing the ordering of variables X and Y, $Fence_3$ is enforcing the ordering of variables P and Q. Thus, even if the above memory accesses are reordered, there is no risk of fence order violation; consequently, in this scenario, the two fences need not stall. *Even if two concurrently executing fences are not staggered, there is no need for the fences to stall, if they are not associates.*

4.2.2 Empirical study

(Fences for ensuring SC) Fences can be utilized to enforce sequential consistency (SC) for programs running on machines only supporting relaxed memory consistency models. This is based on the technique called *delay set analysis*, developed by Shasha and Snir [95], which finds a minimal set of execution orderings that must be enforced to guarantee SC, and inserts memory fences to prevent violation. Various compiler techniques [35, 39, 55, 66, 98] have been proposed to minimize the number of fences required to enforce SC. For example, Lee and Padua [66] developed a compiler technique that reduces the number of fence instructions for a given delay set, by exploiting the properties of fence and synchronization operations. Later, Fang *et al.* [39] also developed and implemented several fence insertion and optimization algorithms in their Pensieve compiler project. In spite of the above optimizations, the program can experience significant slowdown due to

the addition of memory fences, with some programs being slowed down by a factor of 2 to 3 because of the insertion of fences [35, 39, 55].

Program	# of fences	# of fences that needn't stall	% fences that needn't stall
barnes	128222492	98053631	76.48
fmm	1997976	1976759	98.51
ocean	22881352	17720437	77.45
radiosity	57072817	52513870	92.02
raytrace	91049516	84955849	93.31
water-ns	39738011	38383903	96.59
water-sp	41763291	40569645	97.15
cholesky	659910	644208	99.88
fft	214568	214326	99.88
lu	58318507	44048176	75.54
radix	751375	619139	83.41

Table 4.1: Study: A significant percentage of fence instances need not stall.

(Study) We have seen examples of why fences may not be necessary (i.e., they need not stall) dynamically. In this study, we want to check empirically how often the fence instances are not required to stall. SPLASH-2 benchmark programs are used for this study, where fences are inserted to ensure SC using Shasha and Snir's delay set analysis. As shown in Table 4.1, for each benchmark program, the first column shows the total (dynamic) number of fences encountered; the second column shows the total number of fences which were not required to stall, since the interprocessor delay was already ensured during the course of execution; the third column shows the percentage of dynamic fence instances that were not required to stall. As we can see, about 92% of the total fences executed do not need to stall. This motivates the proposed technique for taking advantage of this observation and reducing the time spent stalling at fences.

4.2.3 Conditional Fence (C-Fence)

As opposed to the conventional memory fence that enforces fence orders through intraprocessor delays between memory operations, a C-Fence ensures fence orders through interprocessor delays between associate fences. This gives the C-Fence mechanism an opportunity to exploit interprocessor delays that manifest in the normal course of execution and *conditionally* stall only when required to. C-Fence provides ISA support to let the compiler convey information about associate fences to the hardware. Using this information, the *C-Fence ensures that there is a delay between two concurrently executing associate fences*. In other words, it ensures that all the memory operations prior to the earlier fence complete before the later fence is issued.

Execution of C-Fence$_i$

(i) Mark C-Fence$_i$ as active;
(ii) *while*(any associate is active) stall;
(iii) Continue executing instructions;
. . .
. . .
(iv) Mark C-Fence$_i$ inactive when all memory instructions prior to it have completed

Figure 4.5: The semantics of C-Fence.

(Semantics of C-Fence) A fence instruction (either conventional or C-Fence) is said to be *active* as long as memory operations prior to it have not yet fully completed. However, unlike a conventional fence which necessarily stalls the processor while it is active, a C-Fence can allow instructions past it to execute even while it is active. More specifically, when a C-Fence is issued, the hardware checks if any of its associates are active; if none of its associates are active, the C-Fence does not stall and allows instructions following it to be

executed. If however, some of its associates are active when a C-Fence is issued, the C-Fence stalls until none of its associates are active anymore (Fig. 4.5).

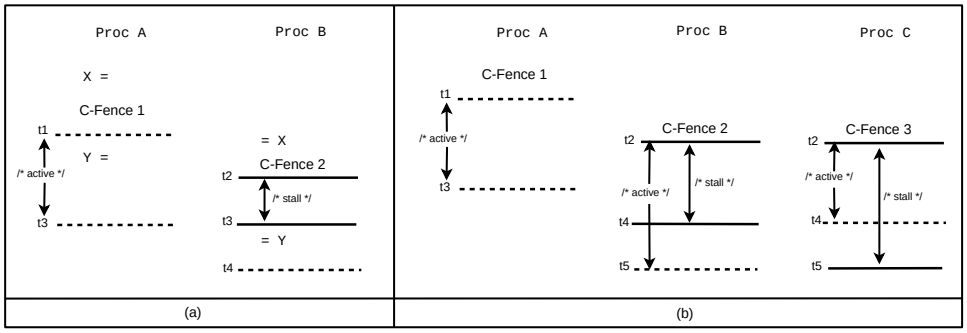

Figure 4.6: (a) Example with 2 C-Fences; (b) Example with 3 C-Fences.

Fig. 4.6(a) shows an example to illustrate its semantics with 2 C-Fences. At time t_1, when C-Fence 1 is issued, none of its associates are active and so it does not stall, allowing instructions following it to be executed. At time t_2, when C-Fence 2 is issued, its associate C-Fence 1 is still active and so C-Fence 2 is stalled. C-Fence 1 ceases to be active at time t_3, at which time all memory operations prior to it have been completed; this allows C-Fence 2 to stop stalling and allows processor 2 to continue execution past the fence.

Now let us consider another example with 3 C-Fences that are associates of each other, as shown in Fig. 4.6(b). As before, C-Fence 1 does not stall when it is issued at time t_1. At time t_2, both C-Fence 2 and C-Fence 3 are made to stall as their associate C-Fence 1 is still active. At time t_3 although C-Fence 1 ceases to be active, the fences C-Fence 2 and C-Fence 3 which are still active, continue to stall. At time t_4, C-Fence 3 ceases to be active, which allows C-Fence 2 to stop stalling and allows processor 2 to continue execution past C-Fence 2. At time t_5, C-Fence 2 ceases to be active, which allows C-Fence 3 to stop stalling and allows processor 3 to continue execution past C-Fence 3. It is important to note from this example that although the two fences C-Fence 2 and C-Fence 3 are waiting

for each other to become inactive, there is no risk of a deadlock. This is because while they are waiting for each other (stalling), each of the processors still continue to process memory instructions before the fence which allows each of them to become inactive at some point in the future.

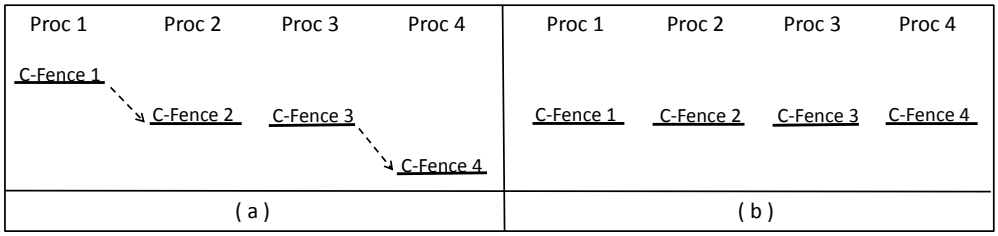

Figure 4.7: Scalability of C-Fence.

(Scalability of C-Fence) The C-Fence mechanism exploits interprocessor delays that manifest between fence instructions during normal course of execution. In other words, it takes advantage of the fact that fence instructions are staggered in execution, as the study showed. However, with increasing number of processors, the more likely it is that *some* of the fences will be executed concurrently. As shown in Fig. 4.7(a), C-Fence 2 and C-Fence 3 are executed concurrently, which necessitates that each of the fences should stall while the other is active. Thus the performance gains of C-Fence mechanism are likely to decrease with increasing number of processors. However, it is expected to perform better than conventional fence because it is very unlikely that all C-Fences execute concurrently, in which case each of the C-Fences are required to stall, as shown in Fig. 4.7(b). In fact, the experiments do confirm that even with 16 processors the C-Fence mechanism performs significantly better than the conventional fence.

4.3 C-Fence Hardware: Centralized Active Table

In this section, we discuss how the C-Fence mechanism is implemented in hardware with a centralized active table. We first describe the idealized HW implementation, and then describe the actual hardware implementation that tries to mimic the idealized hardware implementation, while using much lesser hardware resources.

4.3.1 Idealized hardware

The key step in the C-Fence mechanism is to check if the C-Fence needs to stall the processor when it is issued. To implement this we have a global table that maintains information about currently active fences, called the *active-table*. We also have a mechanism to let the compiler convey information about associate fences to the hardware. Once this is conveyed, when a C-Fence is issued, the active-table is consulted to check if the fence's associates are currently active; if so, the processor is stalled. We now explain what information is stored in the active-table, and what exactly is conveyed to the processor through the C-Fence instruction to facilitate the check. Instead of maintaining the identity of the currently active fences in the active-table, we maintain the identities of the *associates of the currently active fences*. This way, when a fence is issued we can easily check if its associates are active. To this end, each static fence is assigned a *fence-id*, which is a number from 1 to N, where N is the total number of static fences. Then each fence is also given an *associate-id*, which is an N bit string; bit i of the associate-id is set to 1 if the fence is an associate of the i^{th} fence. The fence-id and the associate-id are conveyed by the compiler as operands of the C-Fence instruction. When a C-Fence instruction is issued, its associate-id is stored in the active-table. Then using its fence-id i, the hardware checks the i^{th} bit of all

the associate-ids in the active table. If none of the bits are a 1, then the processor continues to issue instructions past the fence without stalling; otherwise, the processor is made to stall. While the processor stalls, it periodically checks the i^{th} bit of all the associate-ids in the active table, and it proceeds when none of the bits are a 1. Finally, when the fence becomes inactive, it is removed from the active table.

4.3.2 Actual hardware

The hardware described above is idealized in the following two respects. First, the number of static fences is not bounded; in fact, the number of static fences was as high as 1101 in the experiments. Clearly, we cannot have that much bits either in the active-table, or as an instruction operand. Second, we implicitly assumed that the active-table has an unbounded number of entries; since, each processor can have multiple fences that are active, it is important to take care of the scenario in which the active-table is full. We deal with this issue by taking advantage of the fact that although there can be an unbounded number of static fences, a small (bounded) number of the fences typically constitute the major chunk of the dynamic execution count. In fact, the study shows that just 50 static fences account for 90% of dynamic execution count as shown in Fig. 4.8.

Thus, in the actual hardware implementation we implement the C-Fence mechanism for the frequent-fences and the conventional-fence mechanism for the rest. Likewise, when a C-Fence is issued and the active-table is full, we make the fence behave like a conventional fence.

(C-Fence + Conventional Fence) The general idea is to have the frequent-fences behave like C-Fences and the rest to behave like conventional fences. More specifically,

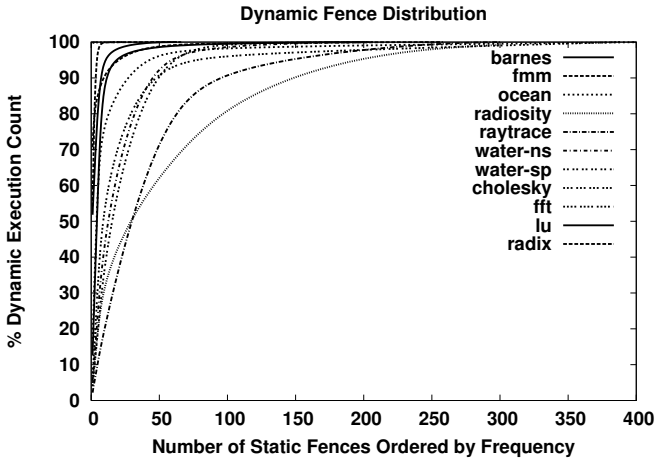

Figure 4.8: Distribution of fences

when a C-Fence is issued, it will stall while any of its associates are active and then proceed. When a conventional fence is issued it will stall until all memory operations prior to it have completed. However, a complication arises if a nonfrequent-fence f (which is supposed to behave like a conventional fence), has a frequent-fence as one of its associates. When such a Fence f is issued, it is not enough for it to stall until all of its memory operations prior to it have completed. It should also stall while any of its associate frequent-fences are still active. To see why let us consider the example shown in Fig. 4.9(a) which shows a frequent-fence C-Fence A first issued in processor A. Since none of its associates are active, it proceeds without stalling. While C-Fence A is yet to complete, nonfrequent-fence Fence B is issued in processor B. It is worth noting that if Fence B is merely made to stall until all its memory operations have completed, there is a possibility of the execution order (a2, b1, b2, a1) manifesting. This clearly is a violation of fence orders. To prevent this situation, Fence B has to stall while C-Fence A is active.

(Compiler and ISA Support) The compiler first performs delay set analysis to determine the fence insertion points. Once the fences are identified, then a profile based approach

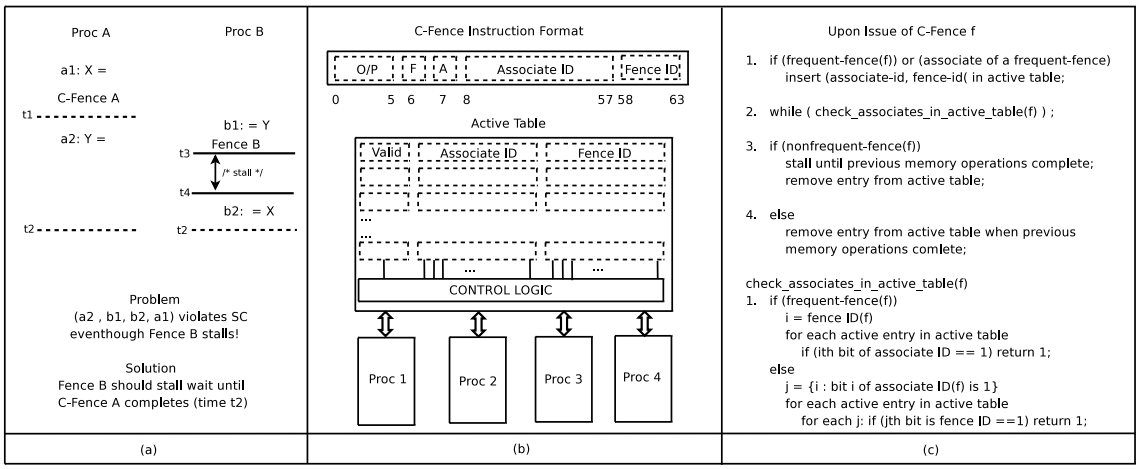

Figure 4.9: (a) C-Fence + Conventional fence. (b) HW support. (c) Action upon issue of a C-Fence.

is used to determine the N most frequent fences; in the experiments we found that a value of 50 was sufficient to cover most of the executed instances. Once the frequent-fences are identified the compiler assigns fence-ids to the frequent-fences. Every nonfrequent-fence is assigned a fence-id 0. For every fence, either frequent or nonfrequent, its associates are identified and then its associate-id is assigned. Fig. 4.9(b) describes the instruction format for the newly added C-Fence instruction. The first 6 bits are used for the opcode; the next two bits are the control bits. The first of the control bits specifies if the current fence is a frequent-fence or a nonfrequent-fence. The next control bit specifies if the current fence has an associate-id. It is worth noting that the current fence has an associate-id if it is an associate of some frequent-fence. The next 50 bits are used to specify the associate-id, while the final 6 bits are used to specify the fence-id for a frequent-fence. Finally, it is worth noting that the value of N is fixed since it is part of the hardware design; in the experiments we found that a value of 50 is sufficient to cover most of the executed instances and we could indeed pack the C-Fence instruction within 64 bits. However, if workloads do necessitate a

larger value of N, the C-Fence instruction needs to be redesigned; one possibility is to use register operands to specify the associate-id.

(C-Fence: Operation) To implement the C-Fence instruction, we use a HW structure known as active-table, which maintains information about currently active fences. Each entry in the active table has a valid bit, an associate-id and a fence-id, each totaling 50 bits, as shown in Fig. 4.9(b). Note that 50 bits are used to represent fence-id in the active table, where bit i is set to 1 if its fence-id is i. We shall later see why this expansion is helpful. To explain the working of the C-Fence instruction, let us consider the issuing of C-Fence instructions shown in Fig. 4.9(a). Let us suppose that the first is a frequent-fence (A) with a fence-id of 5, which is an associate of a nonfrequent-fence (B). When C-Fence A is issued, its fence-id and associate-id are first inserted into the active-table. It is worth noting that its associate-id is a 0 since it is not an associate of any of the frequent-fences. While writing the fence-id to active-table, it is decoded into a 50 bit value (since its id is 5, the 5th bit is set to 1 and the rest are set to 0). Then the active table is checked to verify if any of its associates are currently active; to perform this check the 5th bit (since its fence-id is 5) of all the associates are examined. Since this is the first fence that is issued, none of them will be a 1 and so C-Fence A does not stall. When Fence B is issued, its fence-id and associate-id are first written to the active-table. Since this is a nonfrequent-fence, its fence-id is a 0. Since it is an associate of frequent-fence A (with fence-id 5), the 5th bit of its associate-id is set to 1. Even though this is a normal fence instruction (nonfrequent-fence), we cannot simply stall, since it is an associate of a frequent-fence. More specifically, we need to check if its associate A is active; to do this, bits 5 of all the fence-ids in the active-table are examined. This also explains why the fence-ids were expanded to 50 bits, as this would enable

this check to be performed efficiently. Consequently, this check will return true, since fence A is active; thus Fence B is made to stall until this check returns a false. In addition to this stall, it also has to stall until all of its local memory operations prior to the fence have completed, since it is a nonfrequent-fence. The actions that are performed for the issue of each C-Fence instruction are generalized and illustrated in Fig. 4.9(c).

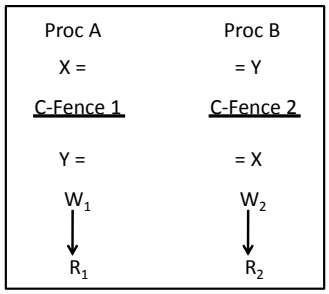

Figure 4.10: Coherence of active-table.

(Active Table) The active-table is a shared structure that is common to all processors within the chip, and in that respect it is similar to a shared on-chip cache, but much smaller. In the experiments we found that a 20 entry active-table was sufficient. Each entry in the active table has a valid bit, an associate-id and a fence-id. There are three operations that can be performed on the active table. First, when a C-Fence is issued the information about the C-Fence is written to the active table. This is performed by searching the active-table for an inactive entry, and inserting fence information there. If there are no invalid entries, it means that the table is full and we deal with this situation by treating the fence like a conventional fence. The second operation on an active table is a read to check if the associates of the issued C-Fence are currently active. Finally, when a C-Fence becomes inactive, the active table entry is removed from the C-Fence. To remove an entry from the active table, the valid bit is cleared. Since the active-table is a shared structure it

is essential that we must provide a coherent view of the active table to all processors. More specifically, we must ensure that two associate C-Fences that are issued concurrently are not allowed to proceed simultaneously. This scenario is illustrated in Fig. 4.10 with fences C-Fence 1 and C-Fence 2. As we can see, the two C-Fences are issued at the same time in two processors. This will result in a write followed by a read to the active-table from each of the processors as shown in Fig. 4.10. To guarantee a consistent view of the active-table, we should prevent the processors from reordering the reads and writes to the active-table. This is enforced by ensuring that requests to access the active-table from each processor are processed in order. It is important to note that it is not necessary to enforce atomicity of the write and read to the active table. This allows us to provide multiple ports to access the active-table for the purpose of efficiency.

(C-Fence: Implementation in Processor Pipeline) Before discussing the implementation of the C-Fence in the processor pipeline, let us briefly examine how the conventional memory fence is implemented. The conventional fence instruction, when it is issued, stalls the issue of future memory instructions until memory instructions issued earlier complete. In other words once the fence instruction is issued, future memory instructions are stalled until (a) the memory operations that are pending within the LSQ (load/store queue) are processed and (b) the contents of the write buffer are flushed. On the contrary, upon the issue of the C-Fence instruction, the processor sends a request to the active table to see if the fence's associates are currently residing in the active table. The processor starts issuing future memory instructions upon the reception of a negative response, which signals the absence of associates in the active-table. The presence of associates in the active-table (positive response), however, causes the processor to repeatedly resend requests to the active-table,

until a negative response is received. Thus the benefit of the C-Fence over the conventional fence is realized when a negative response is received before the completions of tasks (a) and (b). It is thus important that the processor can receive a response from the active-table as soon as possible. To enable this, we maintain a small buffer containing the pairs of instruction address and corresponding fence-id of decoded C-Fence instructions. The buffer behaves as an LRU cache. This enables us to send a request to the active-table even while the fence instruction is fetched (if the instruction address is present in the buffer), instead of waiting till it is decoded. To decide the number of entries necessary in the buffer, we conducted experiments with 5 entries and 10 entries. The results are shown in Fig. 4.11. As we can see, both 5 entries and 10 entries result in high hit rates (94.22% and 95.16% on average, respectively). Hence, we think only 5 entries are enough, which result in a hit rate of 94.22%. That means 94.22% fences can send lookup request before decoding, because the instruction addresses in the buffer indicate the instruction is a fence instruction. The buffer of 5 entries is very small storage, which only requires 45 bytes.

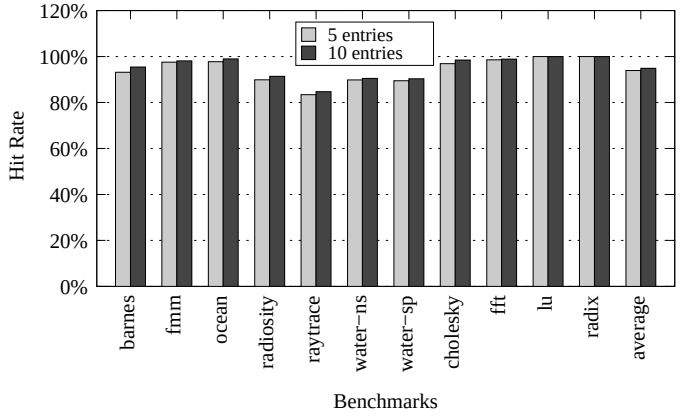

Figure 4.11: Hit rates of the instruction buffer.

(Discussion) For performance reasons, different architectures that use relaxed memory models provide various types of fence instructions that enforce different memory orders

[4, 39]. For example, in Intel IA-32, there exists three types of fence instructions, i.e., *mfence*, *lfence* and *sfence*. Specifically, C-Fences can also be subdivided into different types corresponding to conventional finer grain fences. During execution, new type C-Fences also consult the active table to determine whether they can be issued. If not, they only order specific memory instructions as conventional fences. Hence, better performance can be achieved as finer grain fences are inherently cheaper.

4.4 C-Fence Hardware: Distributed Active Table

While the centralized active table is simple and satisfactory for a lower number of processors, its centralized nature makes it become a bottleneck for a higher number of processors. In this section, we first study the scalability of C-Fence with centralized active table, and then propose a distributed design, *distributed active table*, to ameliorate the request contention arising from the increasing number of processors.

4.4.1 Scalability analysis of C-Fence

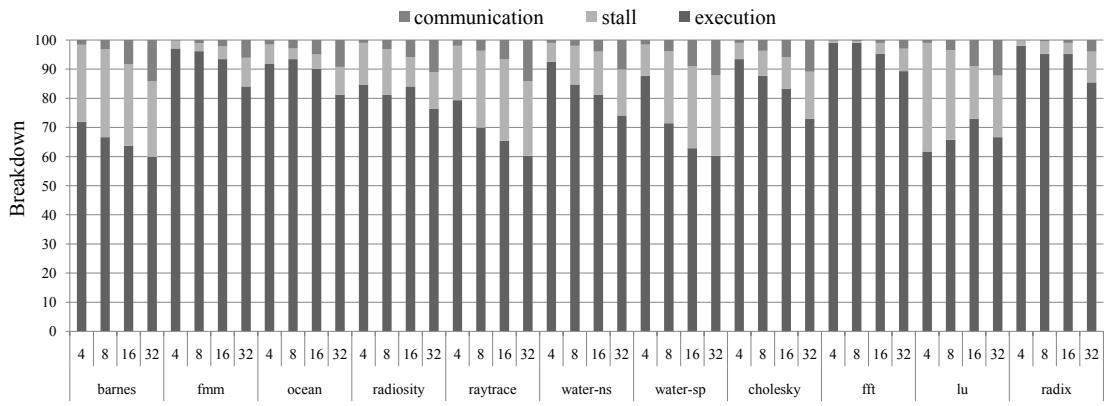

Figure 4.12: Breakdown of runtime.

For the C-Fence mechanism, the runtime spent of each processor can be broken down into three different categories: *execution* time when the processor is busy running programs or it stalls due to reasons unrelated to fences; *stall* time when the processor stalls because of some executing associates; and *communication* time when the processor is waiting for a response from the active table. Fig. 4.12 shows the study results on SPLASH-2 benchmarks, which are measured and averaged across all processors (4, 8, 16 and 32). As we can see, for most programs, the time spent on *communication* and *stall* accounts for a significant portion of the total runtime. This part of time contributes to the overhead of enforcing fence orders. Let us consider the *communication* portion. As the number of processors increases, *communication* accounts for a greater portion of the total runtime. In particular, the *communication* portion is very small for 4 and 8 processors. However, it increases significantly when the number of processors increases to 16 and 32. For some programs (e.g., *barnes*, *raytrace* and *lu*), *communication* can account for around 15% of the total runtime.

The reason for the increasing *communication* time is that, with more processors, more requests are sent to the active table concurrently. The centralized structure can only process these requests in order, which causes the *communication* time to increase. Moreover, as the number of processors increases, a fence has greater chance to stall due to its executing associates. In this case, the processor needs to check the active table periodically, sending the same request to the active table. This further aggravates the contention problem.

To motivate the new design of active table, let us first analyze the requests to the active table. A fence instance can be issued either after only one request or multiple requests to the active table. We categorize these requests into two types: *first-request* which

is sent for a fence instance for the first time, and *subsequent-request* (i.e., periodic request) which is sent as a fence instance is stalled due to negative responses of previous requests. Hence, a fence instance can have only a first-request or both of them. In the latter case, the fence has some executing associates, so it keeps sending requests and checking until there is no executing associate. However, these repetitive subsequent-requests contend with those first-requests and hence increase the average cost of all fences.

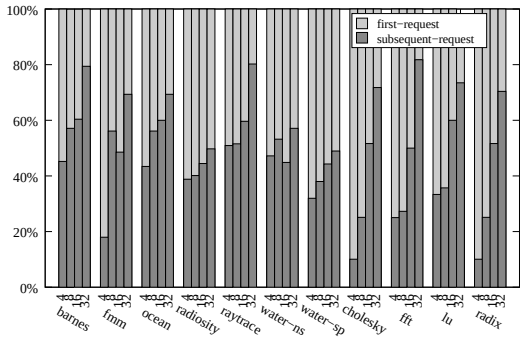

Figure 4.13: Percentages of request types.

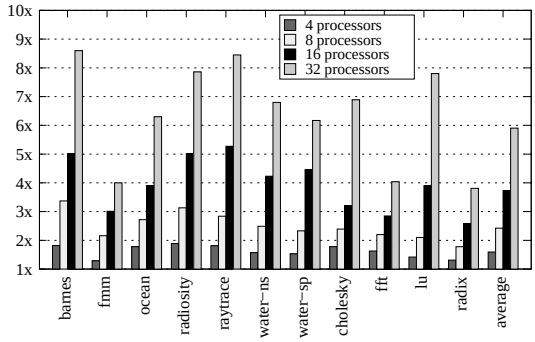

Figure 4.14: Normalized time of processing a request.

Fig. 4.13 shows the percentages of the two request types. As we can see, percentages of subsequent-requests increase as the number of processors increases. The percentage can be as high as 80% for 32 processors (e.g., *barnes*, *raytrace* and *fft*). These repetitive requests place a great deal of burden on the active table, resulting in a longer time for the active table to process a request. Fig. 4.14 shows the average time of processing a request for different numbers of processors, normalized to the time required by a request without waiting. We can see that, as the number of processors increases, a request requires more time to be processed, as it is delayed by its previous requests. For 32 processors, the slowdowns can be higher than 8x for *barnes* and *raytrace*. This results in the increase of the *communication* time. Hence, the centralized active table imposes restrictions to the

scalability of C-Fence mechanism. This observation motivates us to design a distributed active table, which can process subsequent-requests locally and ameliorate contention in the centralized one.

4.4.2 Design of distributed active table

In this section, we introduce the *distributed active table*, distributing the table into multiple sub-tables, each of which handles a subset of fences. Each sub-table is logically associated with a processor, and the information of fences issued by the processor is stored in its corresponding sub-table. All sub-tables are connected by an on-chip network. Fig. 4.15 provides an overview of the distributed active table.

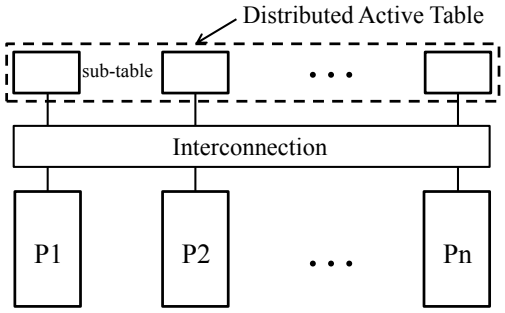

Figure 4.15: Overview of distributed active table.

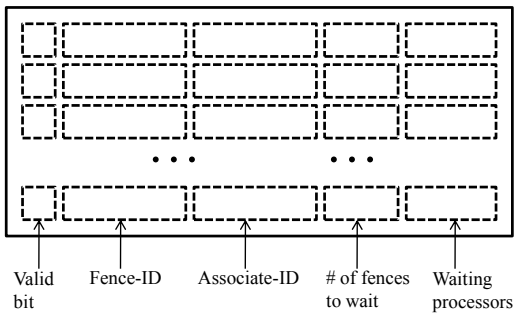

Figure 4.16: Distributed active table – one sub-table.

Fig. 4.16 shows the detail design of each sub-table. There are several entries, each of which consists of five fields.

1. *Valid* bit (1 bit), which indicates whether the entry is available.
2. *Fence-ID* (50 bits), the ID of the frequent fence. Note that fence-id is represented using 50 bits, where bit i is set to 1 if its fence-id is i.
3. *Associate-ID* (50 bits), the associate information of the fence. Bit i is set to 1 if Fence i is one of its associates.

4. *Waiting-Number* (8 bits), indicating how many associates that the fence needs to wait for. The processor only checks this field locally in the corresponding sub-table after it has detected the number of fences it needs to wait for.
5. *Waiting-Processors*. The number of bits equals to the number of processors. This field indicates which processors are waiting for the fence. Bit i is set to 1 if processor i is waiting for this fence to complete.

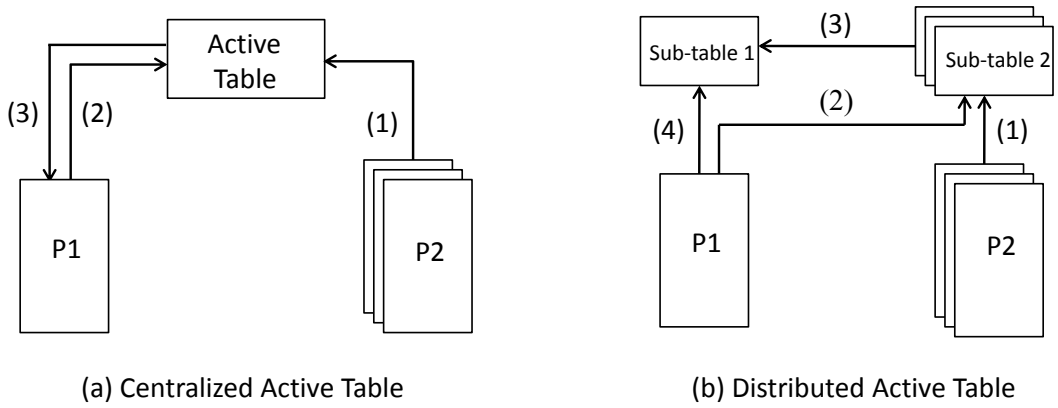

Figure 4.17: The operations of active tables.

(Operations) Fig. 4.17 shows the operations of both centralized and distributed active table. To see the differences between them, let us first recall the operations of the centralized active table, which is shown in Fig. 4.17(a). When a C-Fence instruction is issued, its fence-id and associate-id are stored in the active table (1). Upon a processor is about to issue a C-Fence, it sends a request to consult the active table using the fence-id (2). The active table processes the request by checking associate-ids in the table, and then responds to the processor whether the C-Fence can be issued (3). If the response is positive, the processor issues the C-Fence. Otherwise, the processor stalls and keeps consulting the active table until it can issue the C-Fence. Therefore, if the processor needs to wait for its

executing associates in other processors to complete, it periodically checks the active table, repeating step (2).

In the distributed active table, the goal is to reduce the contention due to periodic requests to the active table from waiting fences. The operations are shown in Fig. 4.17(b), described as follows.

- When a C-Fence instruction is issued, its information (i.e., fence-id and associate-id) is stored in its corresponding sub-table, indicating it is active now (1).
- When a processor is about to issue a C-Fence, it puts the fence-id and associate-id in its sub-table, and then consults other sub-tables by sending a message, including fence-id, associate-id, and the processor number (2). Other sub-tables will then give responses to the consulting processor.
- When a sub-table receives a request from another processor, it uses the incoming fence-id i to check how many associates of the requesting fence are active. To do this, the sub-table checks the i^{th} bits of all the associate-ids of active fences, counting how many bits have been set to 1. After that, the sub-table responds with the number to the requesting processor. Meanwhile, for each associate in the sub-table, the corresponding bit of the *Waiting-Processors* is set to indicate that the requesting processor is waiting for this fence to complete. This is step (3).
- After the requesting processor has received all other sub-tables' responses about the number of active associates, it stores the total number in the field *Waiting-Number*. Then, the processor periodically checks this field, repeating step (4). This is to detect when the field becomes 0, which indicates the processor can issue this fence because there is no active associate at that time.

- For each active fence, when it has completed, the *Valid* bit is set to 0. Meanwhile, according to the *Waiting processors*, the processor sends the completion message to those waiting processors. When other processors receive one such message, they decrease the numbers in *Waiting-Number* by 1 respectively.

(Comparison) The key difference between the centralized and distributed active tables is that, when a processor needs to periodically consult the active table for a stalling fence, the request is always sent to the global table in the centralized design (step (2) in Fig. 4.17(a)), while in the distributed design the request is only sent to the local corresponding sub-table (step (4) in Fig. 4.17(b)). In the distributed design, a fence instance's first-request and its subsequent-requests are processed in different ways. Although the first-request is still sent to other sub-tables, for the subsequent-request, the processor only checks locally the field *Waiting-Number* in its corresponding sub-table. This is beneficial when subsequent-request accounts for a large portion. Furthermore, a first-request consults the field *Associate-ID*, while a subsequent-request consults the field *Waiting-Number*. This indicates that, when a sub-table has these two types of requests at the same time, they can be processed in parallel, which also ameliorates the contention.

4.5 Experimental Evaluation

We performed experimental evaluation with several goals in mind. First and foremost we wanted to evaluate the benefit of using the C-Fence mechanism in comparison with the conventional fence mechanism, as far as ensuring fence orders is concerned. Second, we also wanted to evaluate as to how close the performance of the HW implementation was

with respect to the idealized HW. On a related note, we also wanted to study the effect of varying the values of the parameters in the HW implementation, such as active-table size, the number of frequent fences, etc. Once we figure out the optimal values of these parameters, we also wanted to evaluate the hardware resources that C-Fence mechanism utilizes. Finally, we also wanted to study how the number of processors affects the performance of C-Fence and how distributed active table ameliorates the contention. However, before we present the results of the evaluation, we briefly describe the implementation.

4.5.1 Implementation

Processor	2 processors, out of order, 2 issues
ROB size	104
L1 Cache	32 KB 4 way 2 cycle latency
L2 Cache	shared 1 MB 8 way 9 cycle latency
Memory	300 cycle latency
Coherence	Bus based invalidate
# of active table entries	20
Active-table latency	5 cycles
# of frequent fences	50

Table 4.2: Architectural parameters.

We implemented C-Fence mechanism in the SESC [88] simulator, targeting the MIPS architecture. The simulator is a cycle accurate multicore simulator which also simulates primary functions of the OS including memory allocation and TLB handling. To implement ISA changes, we used an unused opcode of the MIPS instruction set to implement the newly added C-Fence instruction. We then modified the decoder of the simulator to decode the C-Fence instruction and implemented its semantics by adding the active-table and the associated control logic to the simulator. The architectural parameters for the implementation are presented in Table 4.2. The default architectural parameters were used in

all experiments unless explicitly stated otherwise. Note that, the term *processor* used here refers to a logical processor. Our experiments assume a CMP with multiple cores. In the implementation, we did not model the network bandwidth, but focused on the contention in the active table. For the numbers of cores in our experiments, we think the network bandwidth is not a bottleneck, considering fence requests are not so frequent.

4.5.2 Benchmark characteristics

We used the SPLASH-2 [108], a standard multithreaded suite of benchmarks for the evaluation. We could not get the program *volrend* to compile using the compiler infrastructure that targets the simulator and hence we omitted *volrend* from the experiments. We used the input data sets prescribed in [108] and ran the benchmarks to completion. For each benchmark program, fences are inserted to enforce SC using Shasha and Snir's delay set analysis. Fence associates are also identified during analysis. However, since it is hard to perform interprocedural alias analysis for these set of C programs as they extensively use pointers, we used dynamic analysis to find the conflicting accesses as in [35].

Table 4.3 lists the characteristics of the benchmarks. As we can see, the number of static fences varies across the benchmark programs, from 25 fences for *fft* to 1101 fences for *ocean*. Since the number of static fences added can be significant (as high as 1101), we can not store all associate information in hardware and this motivates the technique for applying C-Fence mechanism for just the most frequent fences. We also measured the average number of associates for each fence. As we can see, the average number of associates per fence is around 10, a small fraction of the total number of fences. This provides evidence of why the associate information can be crucial for performance; since a given fence has relatively fewer

Benchmark	# of static fences	Avg # of associates	# of dynamic fences(×1000)
barnes	202	6.65	128222
fmm	259	7.00	1997
ocean	1101	9.59	22881
radiosity	632	19.61	57072
raytrace	301	9.78	91049
water-ns	204	5.70	39738
water-sp	208	5.53	41763
cholesky	388	11.19	659
fft	25	4.16	214
lu	63	4.38	58318
radix	66	3.76	751

Table 4.3: Benchmark characteristics.

number of associates, it is likely that two fences that are executing concurrently will not be associates and each of them can escape stalling. We then measured the number of dynamic instances of fences encountered during execution. As we can see, the dynamic number of fences can be quite large, which explains why fences can cause significant performance overhead.

4.5.3 Execution time overhead : Conventional fence vs C-Fence

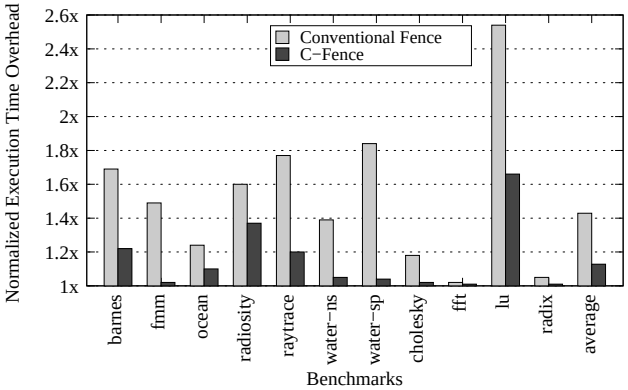

Figure 4.18: Execution time overhead of ensuring SC: Conventional fence vs C-Fence.

In this section, we measure the execution time overhead of ensuring SC with C-Fence mechanism and compare it with the corresponding overhead for conventional fence. For this experiment, we used actual hardware implementation with 50 frequent fences and 20 active-table entries. Fig. 4.18 shows the execution time overheads for ensuring SC normalized to the performance achieved using release consistency. As we can see, programs can experience significant slowdown with a conventional fence, with as high as 2.54 fold execution time overhead (for *lu*). On an average, ensuring SC with a conventional fence causes a 1.43 fold execution time overhead. With the C-Fence, this overhead is reduced significantly to 1.12 fold execution time reduction. In fact, for all the programs (except *lu*, *radiosity*, *raytrace* and *barnes*) SC can be achieved with C-Fence for less than 5% overhead. Since, the C-Fence mechanism is able to capitalize on the natural interprocessor delays that manifest in program execution, most fences can proceed without stalling.

4.5.4 Sensitivity study

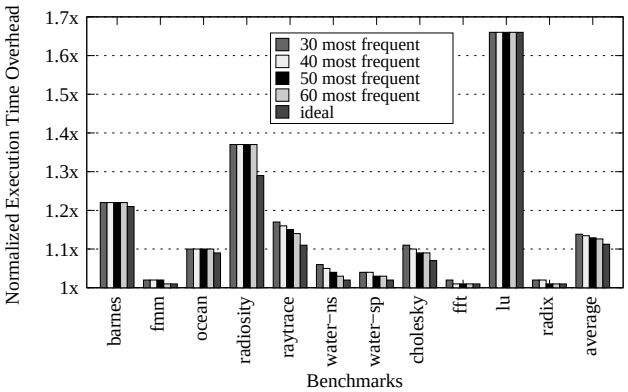

Figure 4.19: Varying the number of frequent fences.

(Sensitivity towards number of frequent fences) Recall that the HW implementation applies the C-Fence mechanism for the N most frequent fences and a conventional fence

mechanism for the rest of the fences. In this experiment, we study the performance by varying N; we vary the value from 30 to 60 in increments of 10. We compare the performance with the *ideal* scheme in which the C-Fence mechanism is applied for all fences. As we can see from Fig. 4.19 the performance achieved even with 30 frequent fences is as good as the ideal performance for most benchmarks. The only exceptions are *radiosity* and *raytrace* in which the performance of ideal is markedly better. For these benchmarks, the dynamic execution counts are more evenly distributed across static fences and because of this a small number of static fences is not able to cover most of dynamic instances. On an average, the performance of the various schemes are as follows: with 30, 40, 50 and 60 frequent fences the respective slowdowns are 1.14x, 1.13x, 1.12x and 1.116x. The performance achieved with the idealized HW corresponds to 1.11x. Thus we observe that with 50 frequent fences, we are able to perform close to the idealized HW implementation.

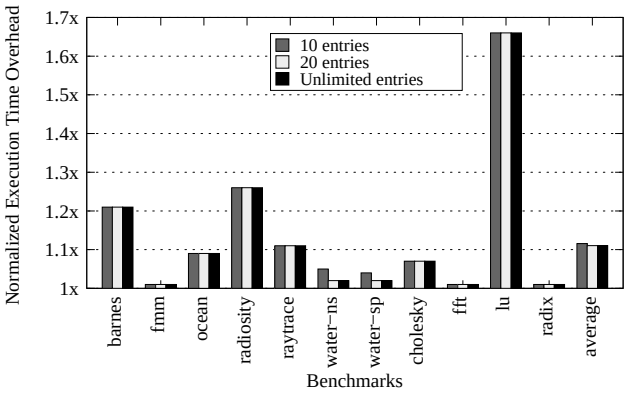

Figure 4.20: Varying the size of active table.

(Sensitivity towards number of active-table entries) The number of active-table entries can potentially influence the performance. This is because, lesser the number of active-table entries, greater the chance that the active-table will be full. Recall that if the active-table is full, then an issued fence cannot utilize the advantage of the C-Fence mechanism

and has to behave like a conventional fence. In this experiment, we wanted to estimate the least possible value of the number of entries of the active-table that can achieve performance close to the ideal. For C-Fence with centralized active table, we varied the number of active-table entries with the values of 10 and 20 and compared it with an idealized HW implementation with unlimited active-table entries. As we can see from Fig. 4.20, even with 10 entries in the active table, we can achieve the idealized HW performance in all but 2 benchmarks (*water-sp and water-ns*). With 20 entries we are able to achieve the idealized HW performance across all benchmarks. In addition, we conducted the same experiment for C-Fence with distributed active table, and found out that 10 entries per sub-table are enough to perform close to idealized HW.

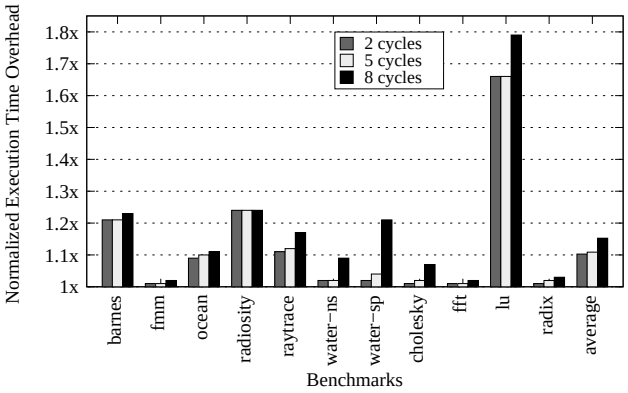

Figure 4.21: Varying the latency of accessing active-table.

(Sensitivity towards active-table latency) The processor can issue memory instructions past C-Fence, only when it receives a negative response from active-table. Thus the round-trip latency of accessing the active-table is crucial to the performance. We varied the latency with values of 2 cycles, 5 cycles and 8 cycles as shown in Fig. 4.21. We observed that the performance stays practically the same as the latency is increased from 2 cycles to 5 cycles. Recall that we send the request to the active-table even as the C-Fence instruction

is fetched; the small buffer (5 entry buffer was used) containing instruction addresses of decoded C-Fence instructions allowed us to do this. However, for an 8 cycle latency, the performance of C-Fence decreases slightly. The active-table is a shared structure similar to the L2 cache, whose latency is 9 cycles. However, the size of the active-table, which is 252 bytes, is much smaller compared to the shared L2 cache, which is 1MB. Furthermore, the active-table is a simpler structure as opposed to the L2 cache; for instance, the L2 tag lookup which takes around 3 cycles is not required. Hence, we think a 5 cycle latency for the active-table is reasonable.

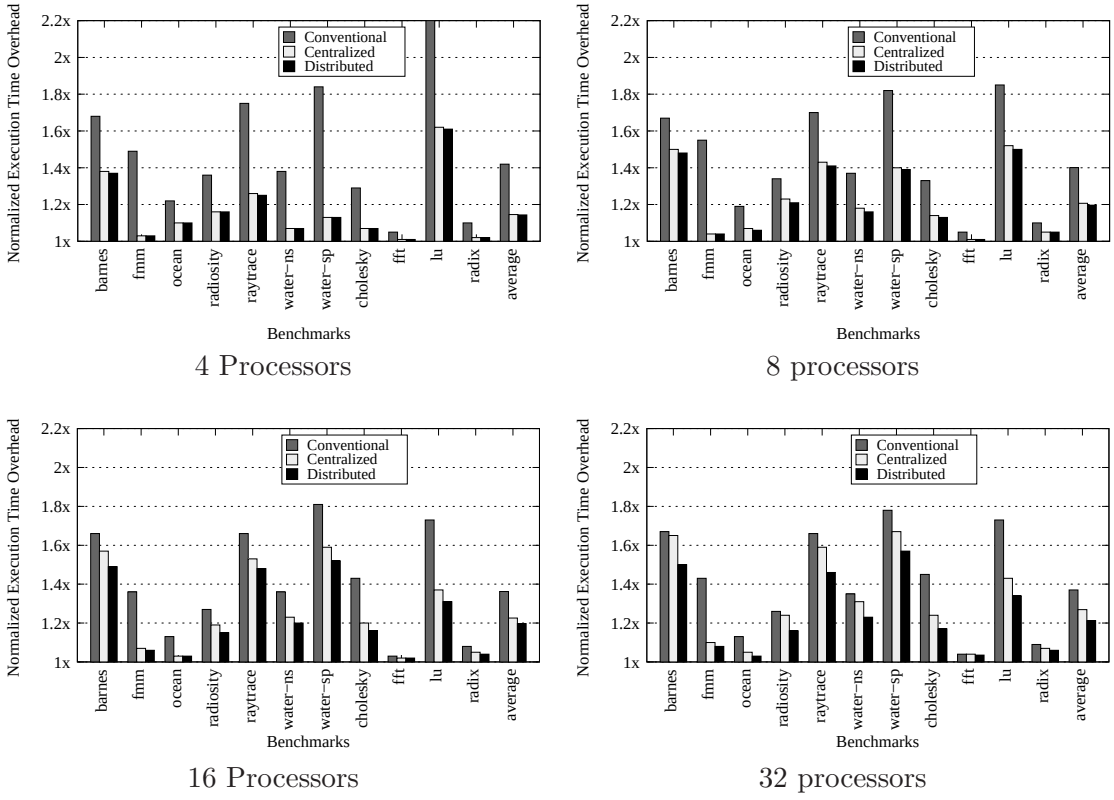

Figure 4.22: Performance comparison for 4, 8, 16 and 32 processors.

(Sensitivity towards number of processors) Here, we wanted to study the performance variation as the number of processors is varied. Fig. 4.22 shows the execution time

overhead of the conventional fence mechanism and C-Fence mechanism (with both centralized and distributed active table) as the number of processors is varied across 4, 8, 16 and 32 processors. For conventional mechanism, we can see that in general the performance remains almost the same as the number of processors is varied. For C-Fence mechanism, although programs such as *lu* show a decrease in execution time, as the number of processors is increased, in general, the execution time overhead of C-Fence increases slightly as the number of processors is increased. As we can see from the average values, with centralized active table, the execution time overhead increases from 1.12x for 2 processors, to 1.15x for 4 processors, to 1.2x for 8 processors, to 1.23x for 16 processors and 1.29x for 32 processors. This is because, as the number of processors increases, there is greater chance of associates executing concurrently, which in turn requires each individual fence to stall. In addition, the contention in the centralized active table also slowdowns the performance. The distributed active table is introduced to ameliorate this contention. With the distributed design, we can further improve the performance of C-Fence mechanism (1.15x, 1.19x, 1.21x and 1.23x for 4, 8, 16 and 32 processors respectively). We can observe that the execution time increases are modest. Consequently, even for 32 processors the C-Fence mechanism is able to reduce the execution time overhead significantly compared to the conventional fence (from 1.38x to 1.23x). Thus, while the relative performance gain reduces as the number of processors increases, the C-Fence mechanism still performs significantly better than conventional fence.

4.5.5 Impact of distributed active table

As *communication* time increases significantly as the number of processors increases to a high number, we redesigned the active table to be distributed, intended to

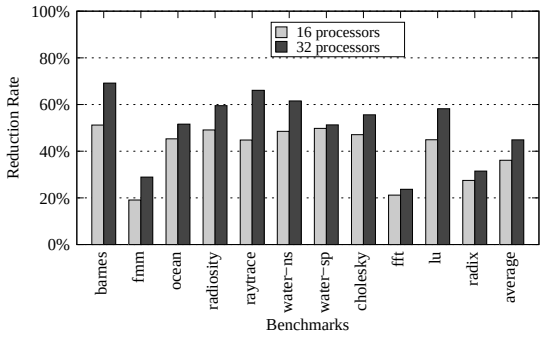

Figure 4.23: *Communication* time reduction.

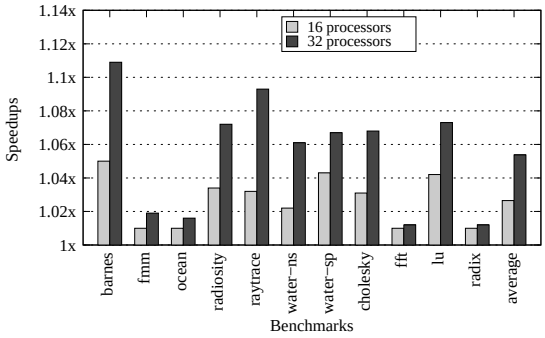

Figure 4.24: Speedups of distributed active table.

reduce the contention. Fig. 4.23 shows the reduction of *communication* time for 16 and 32 processors. Since the *communication* portion for 4 and 8 processors is relatively small, their reduction rates are not interesting. As we can see, the *communication* time reduces a lot for most programs. In particular, *barnes*, *raytrace* and *water-ns* have the reduction rates greater than 60% for 32 processors, as they experience more contention Fig. 4.14) in the centralized active table and the distributed design greatly reduces the contention. On the contrary, the reduction rates for *fmm*, *fft* and *radix* are relatively smaller than others, because their requests have less contention in the centralized active table. On average, the reduction rate is 38.7% for 16 processors and 46.9% for 32 processors. This reduction results in the performance improvement. Fig. 4.24 shows the speedups of C-Fence with distributed active table over C-Fence with centralized active table. On average, the performance increases 2.5% for 16 processors and 5.4% for 32 processors. In particular, for 32 processors, the performance of *barnes* and *raytrace* increases the most (10.9% and 9.3% respectively), as they have a greater portion of *communication* time and the time is reduced a lot using distributed active table. Thus, C-Fence with distributed active table can ameliorate the contention in centralized structure and hence improve the performance.

4.5.6 HW resources utilized by C-Fence

In this section, we want to estimate the amount of HW resources that the C-Fence mechanism utilizes. Recall that the main HW resource that C-Fence utilizes is the active-table. For the centralized design, in the previous experiment we found out that with a 20 entry active-table, we are able to perform as well as ideal HW. We also found out that with 50 frequent fences, we are able to achieve close to the ideal performance. Recall that each entry in the active-table consists of three fields: a valid bit, a fence-id and an associate-id. For 50 frequent fences, an entry would correspond to $1 + 50 \times 2 = 101$ bits. Hence, 20 entries amount to 252 bytes. In addition, we use a 5 entry buffer (requiring 45 bytes) which caches the fence instruction addresses and corresponding fence-ids to speedup the requests to the active-table. Therefore, we would require an on-chip storage of less than 300 bytes. Similarly, for the distributed design, each sub-table requires 10 entries according to the experiments. Each entry requires $1 + 50 \times 2 + 8 + 8 = 117$ bits based on the design described in Section 4.4.2 (assuming 8 processors) and hence each sub-table requires 147 bytes. With a 5 entry buffer, we only require less than 200 bytes of additional on-chip storage per processor core.

4.6 Summary

Fence instructions can significantly reduce the program performance. This chapter proposes a novel fence mechanism known as C-Fence that can ensure fence orders at a significantly lesser performance overhead. While conventional fence enforces *intraprocessor* delays to prevent memory reordering, the C-Fence enforces *interprocessor* delays to ensure

fence orders. However, we observe that most of the interprocessor delays are already ensured during the normal course of execution. The C-Fence mechanism takes advantage of this by conditionally imposing an interprocessor delay only when required to. The C-Fence mechanism requires modest hardware resources, requiring less than 300 bytes of additional on-chip storage. Our cycle accurate simulation results show that C-Fence can be used to significantly reduce the overhead involved in ensuring SC. More specifically, for a dual core processor running SPLASH-2 programs, we found that the C-Fence mechanism can reduce the performance overhead for ensuring SC from 43% to 12%.

Chapter 5

Address-aware Fence

This chapter introduces a *hardware* mechanism, *address-aware fence*, to dynamically eliminate unnecessary memory orderings without aggressive speculation, without the help from compiler or programmer. Unlike a traditional fence which is processor-centric [102], an address-aware fence collects memory access information from other processors to form a *watchlist* associated with the fence instance. The fence now becomes porous, allowing memory accesses whose address is not contained in the watchlist to complete before the fence. The key challenge is how to collect watchlist efficiently and clear watchlist properly. To do this, the implementation of address-aware fence leverages directory-based cache coherence protocol, piggybacking required information on coherence transactions. It is implemented in the microarchitecture without instruction set support and is transparent to programmers. Compared with *scoped fence* (Chapter 3) and *conditional fence* (Chapter 4), address-aware fence is broadly applicable, and has the highest precision, eliminating nearly all possible unnecessary memory orderings due to fences.

5.1 Motivation

In Chapter 4, we have illustrated that, a fence execution is correct if the execution appears to maintain *fence orders*, where reordering memory operations across fences is allowed if the reordering is not observed by other processors. Section 4.1.1 gives a sufficient condition for enforcing fence orders – *fence orders (***F***) and conflict orders (***E***) do not form any cycle, i.e., no cycle in* $\mathbf{F} \cup \mathbf{E}$. C-Fence exploits *fence associates* provided by compiler to break such cycles, by ensuring fence associates do not execute concurrently. However, since C-Fence relies on compiler to statically identify associate information, it is conservative, and thus some unnecessary fence executions are not eliminated. Several such scenarios have been shown in Fig. 1.9. In particular, they illustrate unnecessary fence executions due to conditional branches, dynamic data structures, etc. C-Fence are not able to eliminate them in those scenarios, as those fences are associates which are identified statically and conservatively. Moreover, C-Fence requires compiler support to identify fence associates. However, when there is no source code available, they cannot be applied; and performing interprocedural alias analysis in compiler is complex and conservative.

Techniques	Applicability			
	Dekker-like	Lock-free algo.	Lock	Improving SC
CMO [102]	No	No	Yes	No
l-mfence [63]	Yes	No	No	No
C-Fence	Yes*	Yes*	Yes*	Yes*
Address-aware fence	Yes	Yes	Yes	Yes

Table 5.1: Comparison of existing approaches and address-aware fence (*Compiler support required for identifying fence associates).

There are also two existing techniques proposed to reduce fence overhead non-speculatively *by exploiting program execution in other processors*, i.e.,*conditional memory ordering* (CMO) [102] and *location-based memory fences* (`l-mfence`) [63]. Table 5.1 compares the applicability of these non-speculative techniques in terms of the kind of applications they can handle. In [102], Praun *et al.* observed that memory ordering instructions used on acquire and release of a lock are often unnecessary. They proposed a combined HW/SW mechanism *conditional memory ordering* (CMO) to omit unnecessary memory ordering instructions. CMO only focuses on memory orderings associated with lock acquire and lock release, but cannot handle the other cases. Ladan-Mozes *et al.* [63] proposed *location-based memory fences* to reduce fence overhead. A new instruction (`l-mfence`) is introduced, but it is limited to Dekker-like algorithms. Besides, although C-Fence can be applied in all situations in Table 5.1, it requires compiler support to provide fence associates.

The non-speculative solution address-aware fence *is superior to the above non-speculative techniques in two respects. It is more effective as it is able to exploit all the optimization opportunities shown in Fig. 1.9. It is also broadly applicable as it can be applied in all situations in Table 5.1.*

5.2 Address-aware Fence

Address-aware fence exploits memory operations being performed in other processors to eliminate the above restrictions and optimize the fence executions. It only orders memory operations involving certain memory addresses that must be ordered to maintain fence orders. This is why it is named address-aware fence. At runtime, it is done by detecting and avoiding cycles formed by fence orders and conflict orders. If no cycles can be

formed, fences do not stall the memory operations following them; otherwise, fences will function so as to maintain fence orders.

Address-aware fence is implemented in the microarchitecture without introduction of any new instructions and is thus completely transparent to the programmer. Fence instructions are introduced, as usual, either by the compiler or by the programmer. Fence instructions in the executable are identified at runtime and processed by the hardware. Address-aware fence handles all encountered fences without being aware of how they are used, and hence naturally handles all cases listed in Table 5.1.

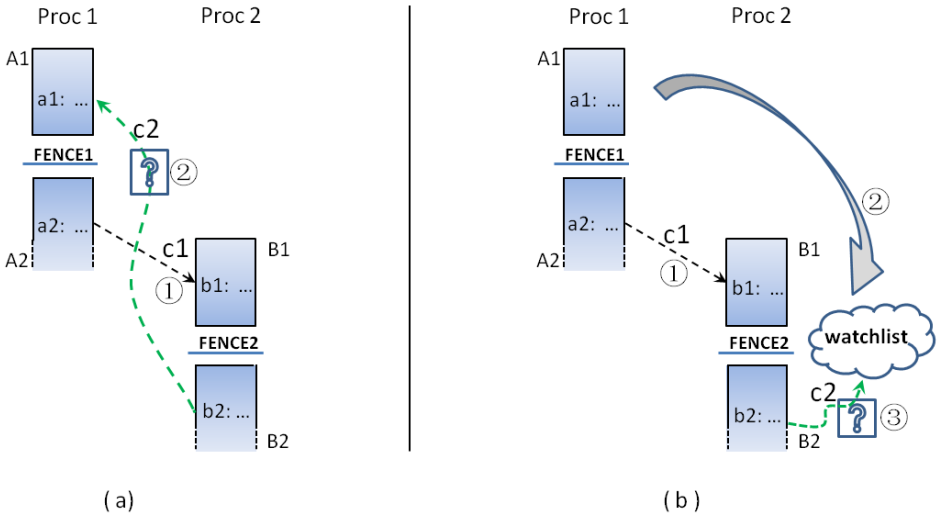

Figure 5.1: (a) Cycle detection; (b) Address-aware fence with watchlist.

This section presents the high-level algorithm of address-aware fence; while the next section will describe the detailed hardware design. The key problem is how to detect possible cycles at runtime. During execution, fence orders are easily known, and hence we have to detect the conflict orders that can form a cycle along with the fence orders. Fig. 5.1(a) shows how cycles can be detected. In Proc 1, suppose all instructions in $A1$ have completed except some pending memory operations, and `FENCE1` does not stall the following

instructions in $A2$ initially. Hence, some memory operations are reordered across `FENCE1` at runtime. At this point, in Proc 2, there is a memory operation in $B1$ which detects a conflicting memory operation in $A2$ (in the next section we will show how the detection is performed relying on cache coherence transactions). It forms a conflict order $c1$ from $A2$ to $B1$ as shown in the figure (①). This event triggers the following event: every memory operation after `FENCE2` has to detect whether there is any pending memory operation prior to `FENCE1` that conflicts with it, forming the conflict order $c2$ (②). Memory operations in $B2$ that do not conflict with pending memory operations in $A1$ can complete without being stalled by `FENCE2` even when there is any pending memory operation in $B1$. However, if there does exist a potential conflict order $c2$ from $B2$ to $A1$, `FENCE2` will delay the involved memory operation in $B2$ to break this potential conflict order. Moreover, if $c1$ does not exist, the detection of $c2$ is unnecessary, as no cycle will be formed without $c1$. Detecting $c2$ is only triggered when there is a possible $c1$.

The cycle detection discussed above does not consider how to relate $c1$ and $c2$. In particular, we should know where to check conflict, e.g., memory operations in $B2$ should check conflict with pending memory operations in $A1$. Furthermore, detecting $c2$ requires every memory operation in $B2$ to check conflict in $A1$ which is on a different processor and thus this is inefficient. Fig. 5.1(b) shows the approach to address this problem. If a conflict order $c1$ is detected (①), memory addresses of all pending memory operations before `FENCE1` are collected to form a *watchlist*, which is associated with `FENCE2` (②). Now, the memory operations after `FENCE2` only need to check the local watchlist to detect conflict, i.e., to detect $c2$ (③). If the address of a memory operation is not contained in the watchlist, it indicates there is no conflict to form $c2$, which further indicates there is no cycle detected

to violate fence orders. Hence, those memory operations whose addresses are not contained in the watchlist can still complete without being stalled by the fence even when there are pending memory operations from before the fence.

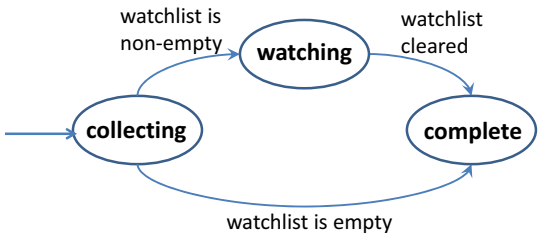

Figure 5.2: States of address-aware fence.

(Life cycle of a fence instance) The execution of a fence instruction proceeds according to the state transition graph in Fig. 5.2. There are three states: *collecting* state, *watching* state, and *complete* state. When the fence is issued it starts out being in *collecting* state where it waits for its watchlist. The memory operations following the fence cannot complete, as at this time the watchlist that is needed to detect cycles is not available. After the fence has collected all necessary addresses to form its watchlist, it switches to next state: the fence switches to *watching* state if the watchlist is non-empty; otherwise it switches directly to *complete* state. In *watching* state the memory operations following the fence must check the watchlist before completion. The watchlist is cleared when the corresponding pending memory operations have completed and the fence switches to *complete* state, where the fence has completed execution. If a fence directly switches from *collecting* to *complete* state, it causes no stalls; thus, it can be viewed as being dynamically eliminated.

(An Example) Fig. 5.3 shows an example to illustrate the execution of address-aware fences. In Fig. 5.3(a), both $a1$ and $a2$ are long latency store misses in Proc 1. At time $t0$, `FENCE1` gets an empty watchlist. Hence, $a3$ can complete even though stores $a1$

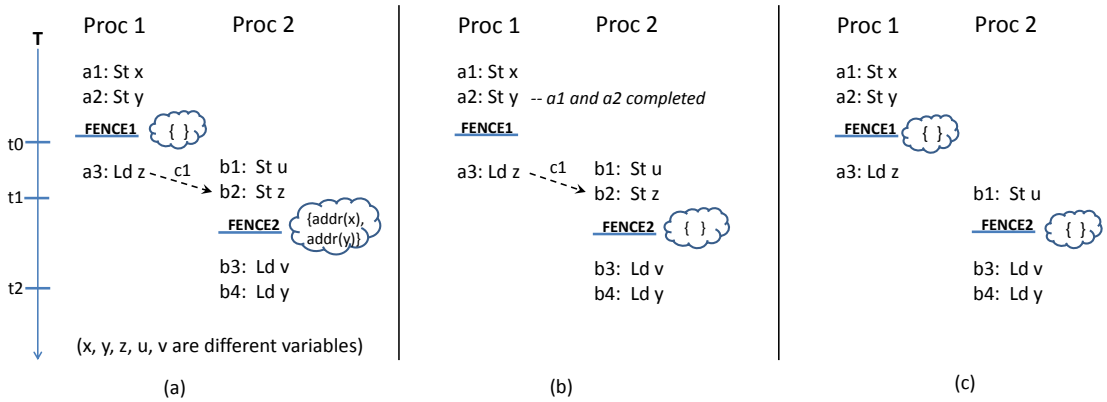

Figure 5.3: An example of address-aware fence.

and $a2$ are still pending. In Proc 2, $b1$ is also a long latency store miss, and $b2$ detects a conflict with $a3$ at time $t1$. Hence, the memory addresses of pending stores prior to `FENCE1` are sent to Proc 2 as `FENCE2`'s watchlist – $\{addr(x), addr(y)\}$. After that, $b3$ can complete even when $b1$ is pending, as its address is not contained in `FENCE2`'s watchlist. On the other hand, $b4$ cannot complete – it is delayed until $a1$ and $a2$ have completed and `FENCE2`'s watchlist is cleared, as shown in Fig. 5.3(b). However, if there is no access to the variable z in Proc 2 as shown in Fig. 5.3(c)(i.e., no instruction $b2$), then there is no conflict order $c1$ and `FENCE2`'s watchlist is empty. In this case, both $b3$ and $b4$ can complete even though $b1$ is pending. As we can see, address-aware fence only stalls necessary memory accesses to maintain a correct fence execution (delaying $b4$ in this example), which reduces stalling overhead due to fences.

5.3 Hardware Design

In this section, we describe the hardware design for address-aware fence. The challenge is to detect and avoid cycles efficiently. In the following discussion, we present

the detailed hardware design in context of a scalable CMP with distributed directory-based invalidation cache coherence protocol. Each processor has a local L1 cache, a bank of L2 cache, and a portion of directory. Each coherence transaction is kept in the directory until it receives the notification of the transaction completion. We assume each processor core to be a dynamically scheduled ILP processor with a reorder buffer (ROB). All instructions retire from ROB in program order. At the head of ROB, loads can retire when they complete, while stores can retire as soon as the value and destination address are available through store buffering technique [45], which allows stores to retire from the head of ROB even before they complete. We will see later that we use an augmented buffer which incorporates the function of store buffer, in which stores are allowed to complete out of order. The store buffering relaxes Store-Load and Store-Store orders, but the traditional fence requires the store buffer to be drained before executing following memory accesses.

Moreover, the processors also support in-window speculation [46], which guarantees Load-Load order by speculative load execution (a speculative load in ROB is squashed if its loaded data is invalidated or replaced before it retires). Besides, Load-Store order is naturally guaranteed by ROB. Therefore, memory operations cannot complete past a pending load, and hence the pending memory operations that can be bypassed can only be *pending stores*.

5.3.1 Operations on address-aware fence

Each processor functions as usual when there is no fence executing. However, when a fence is issued, the processor initiates the process of handling the fence using address-aware fence mechanism. The key operations are *collecting* and *clearing* watchlist for a fence,

which are introduced in this section. After the watchlist has been collected, the fence can retire when it reaches the ROB head. If any load/store following the fence tries to *retire* while there are still pending stores from before the fence, the processor will check whether the address of the load/store is contained in the watchlist.

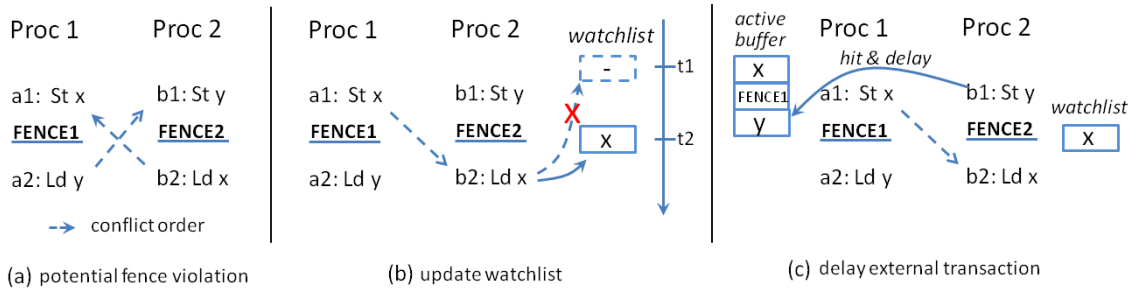

Figure 5.4: Collecting and clearing watchlists.

Collecting watchlist

When a fence is issued, the processor starts to collect watchlist for the fence, which is now in *collecting* state as shown in Fig. 5.2. As described in Section 5.2, a watchlist consists of the memory addresses of a set of pending memory operations that are all pending stores. It is important to obtain the set of pending stores quickly, as memory operations following the fence cannot retire until the watchlist is obtained. To do this, we utilize the directory, where we are able to find all pending stores being serviced by the directory. In the following discussion for simplicity we assume a centralized directory; however, later we describe the modifications needed for a distributed directory. When a fence is issued, the processor sends a request to the directory to fetch the addresses (block addresses) of all pending stores in other processors. The directory compresses the addresses into a watchlist using a bloom filter and sends it back to the requesting processor. Then

the requesting processor records the replied watchlist locally for checking conflict. In this way, we conservatively assume there is a conflict order $c1$ (Fig. 5.1(b)①), and obtain the watchlist by fetching all pending stores (Fig. 5.1(b)②) from the directory.

The processor starts to collect the watchlist as soon as a fence is issued, because we would like to obtain the watchlist as early as possible, so that the fence does not stall the pipeline. However, the collected watchlist may become stale before it is used for checking conflict. Let us take Fig. 5.4(a) as an example for the following discussion. Suppose $a1$ is a store miss, and $a2$ completes past $a1$; then $b1$ completes after $a2$. If we do not take care of $b2$, the execution order $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$ will form a cycle and violate the fence order. In Fig. 5.4(b), let us assume `FENCE2` is issued at time $t1$, when no pending store is present in the directory (i.e., cache miss $a1$ has not reached the directory). So the watchlist obtained for `FENCE2` will be empty. If we use this empty watchlist, we cannot avoid the execution order $a2 \rightarrow b1 \rightarrow b2 \rightarrow a1$, as the empty watchlist will allow $b2$ to complete before $a1$. This is because the watchlist becomes stale and it does not contain x accessed by the pending store $a1$.

To address this problem, we observe that *the watchlist only needs to be updated when there is a cache miss before the fence*. To see why, let us recall cycle detection in Fig. 5.1(a). A cache miss in $B1$ would indicate that there is a potential conflict order like $c1$, so we need to avoid conflict order $c2$. The stale watchlist may not contain all pending stores in $A1$, as new pending stores may be generated after the stale watchlist was obtained. Hence, we have to update the watchlist. On the other hand, a cache hit does not create a conflict order like $c1$, so there is no need to update the watchlist. Therefore, if there is any memory operation before the fence that is a cache miss (load/store miss) in the local cache, the

cache miss transaction sent to the directory will also require the directory to reply with an updated watchlist. Note that, the fence can only retire after it has received the above reply from the directory. For the case in Fig. 5.4(b), if $a2$ completes past $a1$, $a1$ will be present in the directory if it has not completed. Thus, at time $t2$, $b1$ is found to be a miss, so the transaction sent to the directory will obtain an updated watchlist which includes x. Now the updated watchlist containing x will stall $b2$, enforcing the order $a1 \rightarrow b2$ and avoiding fence order violation.

(Unnecessary update) Although a cache miss may create the conflict order $c1$ as shown in Fig. 5.1(a), $c1$ does not really exist if the directory indicates the target block is not cached in other processors. In this case, the watchlist does not need to be updated. In particular, cache misses to local variables will fall into this case, which reduces much unnecessary traffic.

Clearing watchlist

A fence will be in *watching* state when it has retired, and memory operations after the fence have to check the watchlist to bypass it. But the fence cannot simply complete even when all pending stores before it have completed, because memory operations after the fence may still have to check the watchlist to avoid cycles. Let us consider the example in Fig. 5.4(b) again, where $a2$ completes past $a1$ and `FENCE2` has obtained the watchlist containing x. But this watchlist cannot be cleared even when $b1$ has completed, as $b2$ still has to check the watchlist to avoid the cycle by ordering $a1 \rightarrow b2$. The watchlist can be cleared only when $a1$ has completed. That is, the watchlist can be cleared and hence the fence can complete *only when the pending stores whose addresses are contained in the*

watchlist have all completed. If we make each complete store notify the processors which may contain its address, there will be complicated communication of tracking information between processors, complicating the hardware design.

To address this problem, we will delay $b1$ until $a1$ completes. Hence, we introduce a buffer for each processor, called *active buffer*. We consider a memory operation or a fence as *active* if (1) it has retired; and (2) the memory operation is a pending store or there is a pending store before the memory operation/fence. In other words, after retirement, a memory operation or a fence becomes inactive when there is no pending store before it. The *active buffer* records the addresses of all active memory operations, as well as active fences, in the order they are retired. Now, each external coherence transaction will also check the active buffer. If there is a conflict in the active buffer (i.e., the target memory address is found in the active buffer) and there is a fence prior to the conflicting entry, then this coherence transaction is delayed until the target address has been removed from the buffer (i.e., the corresponding memory operation is no longer active). Let us recall Fig. 5.1(b). By using active buffer, if there is a conflict order $c1$ from $A2$ to $B1$, the completion of $B1$ will indicate the completion of $A1$; otherwise, $B1$ cannot complete as coherence transactions sent to `Proc 1` will be delayed. Thus, the watchlist can be safely cleared as soon as $B1$ has completed, because it indicates all pending stores in $A1$ have completed. This simplifies the implementation for clearing watchlist, as the processor can decide when to clear watchlist based on the local information. Note that, to guarantee that any potential conflict is detected, the cache block whose address is in the active buffer is not allowed to be evicted from local cache.

We illustrate the algorithm in Fig. 5.4(c). Suppose $a1$ is a cache miss and $a2$ has completed past $a1$ (so $a1$ will be present in the directory if it has not completed). Now both $a1$ and $a2$ are active, so their addresses are recorded in the active buffer; and `FENCE1` is recorded as well. In `Proc 2`, $b1$ is then executed, which will be a miss. It obtains the watchlist containing x from the directory and also sends a coherence transaction for y to `Proc 1`. Since y is in the active buffer and there is a fence prior to y, the transaction is delayed, and hence the watchlist is not cleared. If $b2$ tries to retire, it has to check the watchlist and hence it stalls. When $a1$ has completed, the active buffer will be empty, and the delayed coherence transaction for y from `Proc 2` can be satisfied. Thus, the completion of $b1$ indicates memory operations prior to `FENCE1` have completed; otherwise, $a2$ is active and hence $b1$ cannot complete. Now the watchlist containing x can be safely cleared as soon as memory operations prior to `FENCE2` have completed, without the risk of forming cycles.

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

Distributed directory

To make address-aware fence scalable, we now consider the implementation with a distributed directory. With a centralized directory, a cache miss transaction, that needs

to update the watchlist, is sent to the directory and obtains all pending stores in other processors. However, when the directory is distributed, a cache miss transaction is only sent to its *home directory*, and only obtains the pending stores in that directory. Hence, the memory operations after the fence can use the watchlist for checking only if they are mapped to the same home directory where the watchlist is collected.

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

(Handling multiple fences) In the above discussion, we only consider one executing fence. However, it is also possible that multiple fences are executing in the pipeline. Since a watchlist has to be removed when the corresponding fence can complete, we have to know which watchlist is associated with which fence. To do this, each issued fence in the

pipeline is associated with a unique tag, and the watchlist collected for the fence will also be associated with the same tag. When a fence completes, only the watchlists with the same tag as the fence are removed. Hence, the watchlists for uncompleted fences continue to remain recorded in the buffer. In particular, the watchlist for a fence can be replaced by the watchlist for a second fence if they are collected from the same home directory, because the latter is the updated watchlist from that directory.

5.3.2 Hardware summary

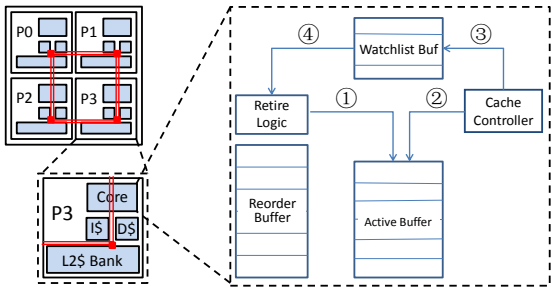

Figure 5.5: Overview of the architecture.

Fig. 5.5 shows the overview of hardware support for address-aware fence. For simplicity, only main logic blocks related to address-aware fence are presented. Each processor core is augmented with two new logic blocks to implement the technique: *active buffer* and *watchlist buffer* as discussed in the previous section. If a retired memory access or fence is active, it is added to the active buffer (①). Each external cache transaction has to check the active buffer, and is delayed if there is a hit (②). Each watchlist is recorded in the watchlist buffer when it is received from the directory (③). Each memory access has to check the watchlist buffer to see if it can retire (④). Although we leverage cache coherence to transfer metadata, the coherence protocol and its transactions are left unchanged. Next, we summarize the operations in the active buffer and watchlist buffer.

(Active buffer) The active buffer records the addresses of all active memory accesses and active fences. It also incorporates the function of store buffer. Each entry of the buffer consists of the following fields: *valid*, *type*, *done*, and *data*. Descriptions of fields are shown in Table 5.2. The *data* field depends on the type. For a store, it includes the destination memory address and a pointer to its data; for a load, it only includes the memory address; and for a fence, it includes a unique tag used to mark its watchlist.

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

Table 5.2: Fields in active buffer.

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

checks the counting bloom filter before checking entries in the active buffer, which greatly reduces useless checks as conflicts are rare.

(Watchlist buffer) The watchlist buffer records all collected watchlists. Each watchlist is a bit vector which contains memory addresses compressed using bloom filter. This is to minimize the network bandwidth, because watchlists are exchanged along with cache transactions. Each entry consists of the following fields: *valid*, *fence tag*, *tile ID*, and *watchlist*. Descriptions of each field are shown in Table 5.3.

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

Table 5.3: Fields in watchlist buffer.

The following are the operations for the watchlist buffer. (1) *Add*. Each watchlist is recorded when it is replied from the directory, with the corresponding fence tag and tile ID. (2) *Delete*. When all memory operations before a fence have completed, the entries with the same fence tag is invalidated. Moreover, when a cache miss before the fence brings back an updated watchlist, other entries are invalidated as they might contain stale watchlists. (3) *Retirement check*. Each retiring memory operation checks against the watchlist with the tile ID to which its memory address is mapped. If it does not find such watchlist, it is first forced to fetch the watchlist from its home directory. If the block address of the memory operation is not contained in the watchlist, it can retire even if there is a pending store prior to the fence. Otherwise, it indicates the retirement of the memory access may result in a violation of fence order, and hence it is delayed. It is worth noting that watchlist buffer will eventually become empty if the processor is stalled, which indicates that the forward progress is guaranteed.

5.4 Experimental Evaluation

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

(Simulation) We have developed a hardware simulation infrastructure using the Pin tool [71] that simulates a directory-based shared memory multiprocessor system. Each processor has a private 4-way 32KB L1 cache and all processors share a L2 cache (16-way 1MB/core). All L1 caches are kept coherent using a directory-based MESI protocol. All cores are connected via a mesh network, which has a link latency of 2 cycles and router latency of 3 cycles. Each instruction takes 1 cycle to execute, and it takes 2, 10, and 300 cycles to access the L1 cache, L2 cache, and main memory, respectively.

Benchmarks	**Description**
dekker	Dekker algorithm [34]
lamport	Lamport Queue [65]
msn	Non-blocking Queue [77]
wsq	Chase-Lev's Work Stealing Queue [27]
bst	Binary search tree
SPLASH-2	8 programs from SPLASH-2 [108]

Table 5.4: Benchmark description.

(Benchmarks) We evaluate the technique using benchmarks shown in Table 5.4. There are two groups, concurrent lock-free algorithms and SPLASH-2 benchmark programs. In the first group, concurrent lock-free algorithms are implemented using fences and atomic compare-and-swap (CAS) instructions. Dekker algorithm (*dekker*) [34] is a classic solution to mutual exclusion problems using only shared variables for communication. Lamport Queue (*lamport*) [65] is a single-producer and single-consumer queue.

Non-blocking concurrent queue (*msn*) is a multiple-producer and multiple-consumer queue. Chase-Lev work-stealing queue (*wsq*) [27] is a lock-free work-stealing deque implemented with a growable cyclic array. *bst* is a concurrent search structure implemented using atomic CAS instructions. Since these lock-free data structures are not closed programs, we constructed harnesses to use them to assess the performance of address-aware fences. The second group of benchmarks are from SPLASH-2 [108]. In these benchmarks, fences are inserted to enforce sequential consistency. We identified the fence insertion points based on Shasha and Snir's delay set analysis, where we employed dynamic analysis to find conflicting accesses as in [35]. We also use these benchmarks to compare address-aware fence and C-Fence (Chapter 4).

5.4.1 Performance

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

Effectiveness of address-aware fence

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

Table 5.5: Effectiveness of address-aware fence.

In this evaluation, we use SPLASH-2 benchmark programs where fences have been inserted for enforcing sequential consistency. All programs are run with 8 threads. Table 5.5 characterizes address-aware fences in terms of how often fences have to take effect. Column 2 in the table shows the number of dynamic fence instructions executed in each program. Although they only account for a small part of all instructions (∼1%), they induce relatively larger execution time overhead, as shown later in Fig. 5.6. With address-aware fence, when there is any active fence, memory accesses have to check with watchlists before retirement. Column 3 shows the number of such memory accesses. Column 4 shows the number of memory accesses whose block addresses are found to be present in the watchlists. We can see that, compared with the number in Column 3, very few memory accesses are stalled by fences. In fact, the number of fences that need to stall (Column 3) is negligible compared with the total number of fences (Column 2). For example, *volrend* has the largest number of detected conflicts, but this is still very small compared the total number of fences – 436 *vs.* 39 Million. On average, only 218 memory accesses are found in watchlists; thus, avoiding nearly all fences from taking effect.

(Comparison with C-Fence) We also implemented and studied C-Fence mechanism, which is able to dynamically eliminate a fence as long as none of its associate fence (provided by compiler) is executing concurrently in other processors. We measured the number of fences which detect executing associates and have to stall. Numbers in Column 5 show that the number of fences that have to stall when C-Fence is used. On an average, around 15% of C-Fences have to stall. In contrast, address-aware fences require very few stalls (Column 4 *vs.* Column 5). This is because address-aware fence is able to exploit more optimization opportunities dynamically, e.g., scenarios in Fig. 1.9, which C-Fence is not able to optimize.

Execution time

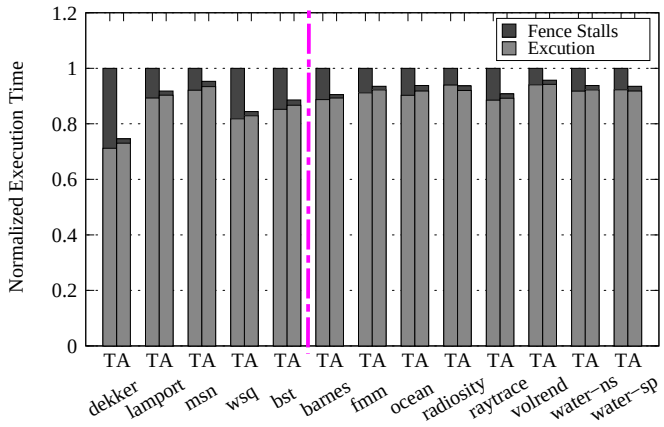

Figure 5.6: Execution time (T – traditional fence; A – address-aware fence).

We measured the execution time of programs with traditional fences and address-aware fences, respectively. Fig. 5.6 shows the results, which are normalized to the execution time achieved using traditional fences. The execution time is broken down into two parts: the stall time due to fences (*Fence stalls*) and the rest of the execution time (*Execution*). For the first group (concurrent lock-free algorithms), with traditional fences, we can see that fence stalls account for 8%-29% of the total execution time. Actually, the fence overhead can be larger depending on the programs using them. Concurrent algorithms have to guarantee correct data accesses by multiple threads, and fences are used to guarantee this goal under relaxed consistency models. However, if data is not frequently accessed by multiple threads concurrently, address-aware fences are able to utilize this opportunity to reduce the fence overhead. In the second group (SPLASH-2 programs), fences are inserted to ensure sequential consistency. Similarly, they experience significant overhead due to fences (about 10% on average). However, since fences are inserted conservatively, many dynamic fence instances are unnecessary, some of which were illustrated in Fig. 1.9. With address-aware fence, only very few fences have to take effect and delay the memory accesses that

follow them. On an average, address-aware fence improves the performance of all benchmark programs by 12.2%. More importantly, we can see that address-aware fence only induces negligible execution time overhead – a fence can retire as long as it has obtained its watchlist, which can be fetched from the directory efficiently. On the other hand, traditional fence has to stall the pipeline until all outstanding stores have completed, which takes a much longer time to access the memory or invalidate shared copies.

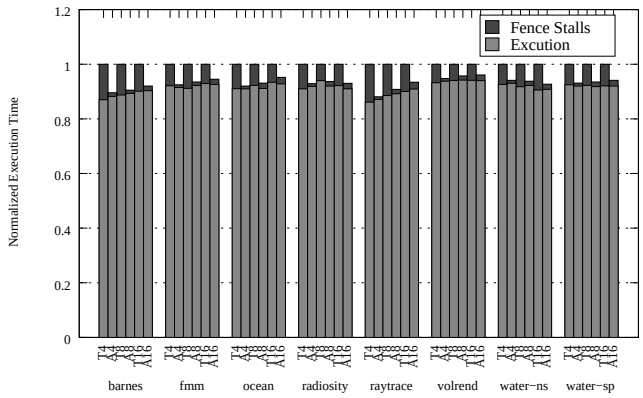

Figure 5.7: Scalability (Tn represents traditional fence with n processors and An represents address-aware fence with n processors).

(Scalability) We vary the number of processors with 4, 8 and 16 processors and measure the execution time of benchmark programs from SPLASH-2. The results are shown in Fig. 5.7, where data are presented in the same way as in Fig. 5.6. Each benchmark program has execution time of traditional fence and address-aware fence with 4, 8 and 16 processors, respectively. We can see that, with different numbers of processors, the execution time overhead induced by address-aware fence is not affected significantly. Therefore, the implementation appears scalable. In fact, the implementation leverages directory-based cache coherence protocol, piggybacking required information on coherence transactions. Without introducing centralized structures, the implementation maintains the level of scalability delivered by the directory-based cache coherence.

5.4.2 Space and traffic

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

Table 5.6: Characterization of space and traffic.

To support address-aware fence, we introduced active buffer and watchlist buffer to record information and leveraged cache coherence to avoid fence order violation. Table 5.6 characterizes space and traffic induced by address-aware fence. We conducted the experiments using SPLASH-2 benchmark programs with 8 threads.

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

of processors, with each entry recording the watchlist obtained from the corresponding processor.

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

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

5.5 Summary

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

Chapter 6

Related Work

This chapter discusses the research work related to enforcing and reducing memory ordering. They are categorized to hardware, compiler, programming language, verification and debugging, as well as some other work in the last section.

6.1 Hardware Support

6.1.1 Speculative techniques

Conventional hardware design can be augmented to accommodate the requirement of sequential consistency (SC), where speculation is typically used to achieve high performance despite ensuring SC. The key observation is that, for most memory accesses, the execution would still follow SC, even if they are reordered. A small fraction of memory accesses that require the delay for correctness are efficiently detected and corrected by reissuing the accesses to the memory system. Thus, the common case is handled with maximum speed without violating SC.

Gharachorloo *et al.* [46] propose two techniques to enhance the performance of SC – *hardware-controlled non-binding prefetch* and *speculative execution for load accesses*. Prefetching makes it more likely that a memory access will find its data in cache; and speculation allows the processor to proceed with load accesses that would otherwise be delayed due to earlier pending accesses. However, such reordering is within the instruction window, which still requires write buffer be drained before the subsequent memory operations can complete. The MIPS R10000 processor supports SC and includes this technique [109]. Ranganathan *et al.* [87] propose speculative retirement, where loads are speculatively retired while there is an outstanding store, and stores are not allowed to get reordered with respect to each other. The above two techniques allow only loads to bypass pending loads and stores speculatively; stores are not allowed to bypass other memory accesses.

In [49], Gniady *et al.* propose *SC++*, which allows both load and store to speculatively bypass each other. By supplementing the reorder buffer with the Speculative History Queue (SHiQ), *SC++* maintains the speculative states of memory accesses. This enables the proceeding of retiring instructions, which otherwise need to be stalled due to the long access latency of a store. This work shows that SC implementations can perform as well as RC implementations if the hardware provides enough support for speculation. Furthermore, since SHiQ may need to be very large to tolerate long latencies, in [48], they propose *SC++lite*, which places the SHiQ in the memory hierarchy, providing a scalable path for speculative SC systems across a wide range of applications and system latencies.

The above works track speculative state at a per-instruction or per-store granularity, which requires the history buffers must grow proportionally to the duration of speculation. Another set of techniques [16, 25, 107] propose the idea of enforcing consistency at

the granularity of coarse-grained chunks of instructions rather than individual instructions. These approaches execute programs in continuous speculative chunks, buffering register state via checkpoints and buffering speculative memory state in the L1 cache. At the end of a chunk, they send their write set to other processors, which use this write set to detect violations. If there is a violation, the recovery mechanism invalidates speculatively-written lines in L1 and refetches data from lower levels. [50, 26] first introduces this concept in transactional memory, although they do not target SC. Ceze *et al.* [25] propose BulkSC, which enforces SC at chunk granularity. A chunk that is going to commit sends its signatures to the arbiters and other processors to determine whether the chunk can be committed. Ahn *et al.* [6] propose a compiler algorithm called *BulkCompiler* to drive the group-formation operation and adapt code transformations to existing chunk-based implementations of SC. They select chunk boundaries so that they can maximize the potential for compiler optimization, and minimize the chance of reexecution. Blundell *et al.* [16] propose InvisiFence, which does not require either fine-grained buffers to hold speculative state or require global arbitration for commit in speculation state.

Speculative techniques can achieve good SC performance comparable to release consistency (RC). Although the above techniques are designed for efficient SC implementation, they can also be adjusted to reduce fence overhead. However, the main issue with these speculative techniques is that they employ extensive post-retirement speculation, which requires high hardware complexity. On the contrary, this dissertation aims at good performance without aggressive hardware speculation.

6.1.2 Non-speculative techniques

There are also works on reducing memory ordering overhead non-speculatively. In [102], Praun *et al.* propose *conditional memory ordering* (CMO) based on the observation that memory ordering instructions used on acquire and release of a lock are often unnecessary. The goal of CMO is to optimize and reduce the cost of the acquire-release memory synchronization protocol using a purely runtime technique. Ladan-Mozes *et al.* [63] propose *location-based memory fences* (`l-mfence`) to reduce fence overhead. It is a lightweight solution, but it is limited to the Dekker-like algorithms. *Address-aware fence* presented in the dissertation does not limit to specific applications. It handles all encountered fences without being aware of how they are used. Concurrently with this work, Duan *et al.* propose *WeeFence* [36], which shares similarities with *address-aware fence*. They use dedicated on-chip structures to record addresses of pending stores, while address-aware fence utilizes cache coherence directory to obtain pending store information.

Non-speculative techniques have also been proposed to enforce SC efficiently. Singh *et al.* [97] propose to identify thread-local and shared read-only data, and enforce SC by only ordering the accesses to remaining data. With assistance from static compiler analysis and hardware memory management, a processor can easily determine these safe accesses; and an additional unordered store buffer is employed to allow later memory accesses to proceed without a memory ordering related stall. In contrast, *conflict ordering* described in the dissertation is a pure hardware solution, which checks if there is any conflict with pending accesses in remote cores before committing a memory operation from the ROB, resulting in non-speculative completion.

6.2 Compiler Support

6.2.1 Fence insertion for enforcing sequential consistency

Programs running on machines supporting relaxed consistency models can be transformed into ones in which SC is enforced. Shasha and Snir, in their seminal work [95], observe that not all pairs of memory operations need to be ordered for SC; only those pairs which conflict with others run the risk of SC violation and consequently need to be ordered. To ensure memory operation pairs are not reordered, memory fences are inserted. They then propose *delay set analysis*, a compiler based algorithm for minimizing the number of fences inserted for ensuring SC. This work leads to other compiler based algorithms that implement various fence insertion and optimization algorithms.

Although [95] provides some of the foundational ideas for enforcing SC from a compiler perspective, it is not designed as a practical static analysis. Midkiff and Padua [80] extend Shasha and Snir's characterization to work for programs with branches, alias and array accesses, but they do not provide a polynomial-time algorithm for performing the analysis. Krishnamurthy and Yelick [60, 59] provide some early implementation work on cycle detection. They show that computing the minimal delay set for an arbitrary parallel program is an NP-hard problem for MIMD programs and propose a polynomial-time algorithm for analyzing SPMD programs. Chen *et al.* [29] substantially improve both the speed and the accuracy of the SPMD cycle detection algorithm described in [60]. By utilizing the concept of strongly connected components, they improve the running time of the analysis asymptotically from $O(n^3)$ to $O(n^2)$.

Some compiler techniques aim to find the delay set that is sufficient to enforce SC, and minimize the number of inserted fences. Lee and Padua [66] develop a compiler tech-

nique that reduces the number of fence instructions for a given delay set, by exploiting the properties of fence and synchronization operations. Later, Fang *et al.* [39] also develop and implement several fence insertion and optimization algorithms in their Pensieve compiler project. [55] propose schemes for concurrent languages that employ simple synchronization and little aliasing, e.g., Titanium. However, their techniques are hard to extend to C/C++ programs because the extensive use of pointers in such programs seriously complicates the analysis. Sura *et al.* [98] describe co-operating escape, thread structure, and delay set analysis to enforce SC. In [67], Lee *et al.* use a perfect escape analysis to provide the best bound on the overhead incurred for a SC memory model. In addition, [78] takes a different approach to reducing the 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 the above works target reducing the number of fences, this dissertation aims to make fence cheap by eliminating unnecessary fence stalls, and hence it would help achieve SC more efficiently.

6.2.2 Compiler optimizations consistent with memory models

Compiler optimizations should also be consistent with the required memory models. Transformations that can have the effect of reordering memory operations should be carefully examined to avoid violations. [104] analyzed the validity of several common program transformations in multithreaded Java, as defined by the Java Memory Model (JMM) [73]. It identified several valid and invalid transformations with respect to Java memory model. In [105], Ševčík *et al.* describe a concurrency extension to a C-like programming

language that provides end-to-end TSO semantics. They prove that a set of optimizations are TSO-preserving, which provide an end-to-end guarantee when the generated binaries are executed on x86 hardware. Later, they present *CompCertTSO* in [106], a verified compiler that generates x86 assembly code, permitted by the source language x86-TSO semantics. [103] gives a rigorous study of a group of transformations that involve both reordering and elimination of memory accesses, and proves that any composition of these transformations is sound with respect to the DRF guarantee. [100] describes further work on verified fence elimination. [12, 89] prove the correctness of the proposed compilation scheme from C/C++11 concurrency primitives to POWER/ARM machines. Moreover, [75, 22] also show that a large class of optimizations crucial for performance are either already SC-preserving or can be modified to preserve SC while retaining much of their effectiveness.

These compiler optimizations cannot prevent hardware from reordering, and hence they are orthogonal to the techniques proposed in this dissertation. They are complementary in improving program performance.

6.3 Programming Support

Mainstream programming languages like C++ and Java use variants of the data-race-free memory model known as DRF0 [4, 5], which guarantee SC as long as the program is free of data races. Some works, such as [13, 15], give formal semantic descriptions for these programming languages. However, if the programs do have data races then these models provide much weaker semantics [73, 17]. To redress this, there have been works that employ dynamic race detection [38] in order to stop execution when semantics become undefined. Since dynamic data race detection can be slow, recent works [70, 74, 96] propose

raising an exception upon encountering an SC violation as this can be done more efficiently. They also require compiler support to partition a program into regions, and many valid compiler optimizations can be performed within a region but not across regions. During execution, hardware guarantees that there is no SC violation by detecting conflicting accesses in concurrently executing regions; otherwise, an exception will be raised. These efforts in programming language are rooted in the observation that good programming discipline requires programmers to protect shared accesses with appropriate synchronization.

On the contrary, the techniques in this dissertation allow data races, as data races can be intentional and harmless, such as in lock-free data structures. In particular, *conflict ordering*, *address-aware fence*, and *conditional fence* rely on the detection of data races which are overlapping in time and intertwined in a manner that can form a cycle.

6.4 Debugging and Verification

Most prior SC violation detection schemes have used data races as proxies for SC violation [44, 16, 25, 49, 70, 74], which is highly imprecise. In [35], Duan *et al.* use a race detector to construct a graph of races dynamically, and then traverse the graph off-line to find potential SC violations. However, it inherently suffers from the same limitations as data race detection techniques. False negatives resulting from data race detection could lead to undetected SC violations, and false positives would report SC violations that never occur. In [23], Burnim *et al.* develop a tool to execute programs with a biased random scheduler to create with high probability a predicted sequential consistency violation. Recently, [81, 85] propose hardware schemes to precisely detect SC violations in machines implementing relaxed models. When there is violation that is about to occur, an exception is triggered,

providing information to debug the SC violation. The above works focus on debugging, so they target precise detection; however, this dissertation aims to avoid unwanted ordering violations, and targets execution efficiency.

There are also works in verification community [19, 18, 20, 21, 28, 53, 61, 62, 69] to verify and enforce SC. Verification tools developed in [20, 21] aim at inserting fence instructions accurately. These tools take the concurrent program and a relaxed memory consistency model, e.g. TSO [21], as inputs, then enumerate all possible execution patterns and simulate them according to the memory consistency model. Fences can be inserted according to the executions that lead to non-SC results. While promising, these techniques are not designed for on-the-fly SC enforcement with negligible overhead, or reducing fence overhead. Moreover, [30, 32, 76] are works that can verify if a memory system hardware is correctly implemented, which is different from the goal of this dissertation.

6.5 Other Works

6.5.1 Fence instructions in commercial architectures

Commercial architectures provide various fence instructions [4]. Some of them also enforce ordering of a subset of memory operations. 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 (i.e., with mask values indicating #LoadLoad, #StoreLoad, #LoadStore, and #StoreSore); 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 this dissertation also provide fences for enforcing ordering of a subset of memory operations, its improvements will be orthogonal to the above fences. While the above fences explore the ordering of a combination of previous load and store operations with respect to future load and store operations, this dissertation explores the ordering of a certain set of memory accesses whose scopes (*scoped fence*) or addresses (*address-aware fence*) are considered.

6.5.2 Optimizing lock implementations

Thin locks [10] are an influential work on lock optimization for Java. They are developed based on the observation that sequential programs often needlessly use locks indirectly by calling thread-safe libraries. By overlaying CAS lock on top of Java's more costly monitor mechanism, thin locks avoid all monitor accesses for a single threaded program, and also guarantee the correctness if additional threads are introduced. There are also other refinements [56, 83, 82] of thin locks. The basic idea is to allow a particular thread to reserve a lock, and hence acquisitions of the lock by the reserving thread can be performed efficiently. [101] propose a fast biased lock, which simplifies and generalizes prior implementations of biased lock. The above work focuses on reducing the frequency of atomic RMW operations for implementing locks, while this dissertation aims to reduce the memory ordering overhead induced by fence instructions. Speculative lock elision (SLE) [86] eliminates unnecessary serialization of threads due to critical sections to achieve high performance in multithreaded programs, using speculation. This is because, dynamically,

these critical sections could have safely executed concurrently without locks. Besides, lock elision with transactional memory [99, 33] also use speculative technique to dynamically eliminate lock operations. In contrast, this dissertation does not simply focus on lock operations but also fences which are also used for lock implementation, and the proposed techniques do not resort to aggressive speculation.

6.5.3 Formalization of memory models for commercial architectures

Commercial architectures with relaxed memory models are often described in ambiguous informal prose, leading to widespread confusion. In [94], Sewell *et al.* describe how the recent Intel and AMD memory model specifications do not match with the actual behaviors observed in real machines. Thus, it is necessary to give formal description for these architectures. [91] formalizes the Intel and AMD architecture specifications of the time, but those turn out to be unsound with respect to actual hardware later. Then, x86-TSO model is described in [94, 84], which is sound with respect to actual processor behavior, matches the current vendor intentions, and is a good model to program above. For POWER/ARM machine, Alglave *et al.* [7, 8] give preliminary axiomatic models for them. Later, in [90], Sarkar *et al.* provide an abstract-machine model for POWER that can accurately capture the architectural intent and observable processor behavior for a wide range of subtle examples. Moreover, [72] describes an axiomatic model that is provably equivalent to the operational model in [90]. These formalization works help us better understand the behaviors of commercial architectures.

Chapter 7

Conclusions

7.1 Contributions

This dissertation makes contributions in eliminating unnecessary memory ordering on multiprocessors. It presents techniques to implement sequential consistency (SC) efficiently and reduce fence instruction overhead. The techniques are based on the observation that a subset of memory orderings that are in general required, may be unnecessary during the current execution. This dissertation explores programming, compiler, and hardware support to efficiently detect such unnecessary memory orderings and eliminate them to improve performance. In particular, this dissertation makes the following contributions.

I. Efficient hardware SC implementation without aggressive speculation. It is attractive to have a system in which programmers can treat the combined compiler and hardware as a SC system and still transparently profit from the performance advantage of the relaxed memory models, achieving both programming efficiency and high performance. Marino *et al.* [75] have recently shown that it is possible for compiler to preserve SC efficiently. However, it

is questioned whether the costs of enforcing SC in hardware justify its benefits, as prior SC proposals need to employ aggressive speculation, which has high hardware complexity. This dissertation presents *conflict ordering* for efficient hardware enforced SC, by detecting and preventing cycles in memory access orders across threads which may violate SC. This work demonstrates that the benefits of SC can indeed be realized using lightweight hardware resources, without requiring post-retirement speculation.

II. Making fence instructions cheap. Fence instructions are used by the compiler and hardware to prevent the reordering of memory accesses. However, they are expensive in today's processors, as enforcing memory ordering is expensive. Consequently, programmers strive for ways of avoiding excessive use of fences, or replacing heavyweight ones with lightweight ones. Unfortunately, this can lead to complicated code or sometimes, even bugs. This dissertation explores various ways to reduce fence overhead, making them cheap to use. Programmer and compiler can use fences conservatively without worrying about their overhead. In particular, in high-level languages such as C++ or Java, atomic or volatile is used to declare some shared variables. Their semantics implies the same effect as fence instructions. Lightweight fence would be able to implement accesses to such variables efficiently.

III. Exploring programming, compiler, hardware support for eliminating unnecessary memory ordering. Memory ordering requirements in memory models influence many aspects of system design, including the design of programming languages, compilers, and the underlying hardware. All these aspects work together to ensure memory ordering requirements. This dissertation discusses the trade-offs between them to design a system that can detect and eliminate unnecessary memory ordering. With programming and compiler support, a de-

sign with simple hardware support is possible; on the other hand, hardware should be finely designed, but it also leads to higher precision of detecting unnecessary memory ordering.

IV. Programming directed fence. *Scoped fence* (S-Fence) introduces the concept *fence scope*, and enables programmers to express their ordering demands. S-Fence bridges the gap between programmers' intention and hardware execution with respect to the memory ordering enforced by fences. The idea of fence scoping is consistent with the principle of encapsulation and modularity of object-oriented programming languages. This makes it easy to incorporate fence scoping in current popular object-oriented programming languages. Moreover, S-Fence is easy for programmers to use and requires simple modifications to the hardware. In particular, S-Fence only makes changes locally in each processor core, without adding inter-processor communication, and hence scalability is not an issue for S-Fence.

V. Compiler directed fence. *Conditional fence* (C-Fence) is proposed to utilize compiler information to dynamically decide if there is a need to stall at each fence. It does not require programming effort, and the hardware cost is lightweight. C-Fence can be used to enforce SC efficiently. Compared to previous software based approaches, C-Fence significantly reduces the slowdown; compared to previous hardware based approaches, C-Fence does not require any speculation.

VI. Hardware directed fence. *Address-aware fence* is proposed to speedup fence execution, without the help from compiler or programmer. It is implemented in the microarchitecture without instruction set support and is transparent to programmers. Fence instructions in the executable are identified at runtime and processed by the hardware. Address-aware

fence is broadly applicable, and has the highest precision, eliminating nearly all possible unnecessary memory orderings due to fences.

7.2 Future Directions

I. Memory ordering in distributed systems. In distributed systems, software shared virtual memory [54, 68, 24, 57] is usually implemented to provide a virtual address space that is shared among all processors. Application programs can use the system as a physically shared memory machine. There are two design choices that greatly influence the implementation of a shared virtual memory: the granularity of the memory units and the strategy for maintaining coherence. The memory unit can be a page or another fine-grain unit, that influences the communication frequency between distributed memories. Furthermore, strategy for maintaining coherence involves deciding when an update should be seen by remote processors. These two factors greatly influence program performance in a distributed system, as communication cost between processors is nontrivial due to lack of physically shared memory. A strong memory consistency model (e.g., SC) can be expensive, leading to poor performance. Therefore, in distributed systems, some relaxed models are usually implemented, e.g., lazy release consistency [58], entry consistency [14], etc. As a result, it would be interesting to explore ways to improve performance of stronger memory models in distributed systems. The ideas described in this dissertation to eliminate unnecessary memory ordering can be considered in context of distributed systems, although there exist challenging problems to tackle. For example, in multicore processors, we can obtain information from directory (closer to processor than memory) to assist reordering decision quickly; however, in distributed systems, there is no such structure. It is also possible

to consider application characteristics to design application-specific protocols to improve performance.

II. Memory ordering in manycore processors. In manycore processors, a large number of cores are used to perform massively parallel computing. Researchers from Intel have announced that the architecture of their recent 48-core processor, called Single-Chip Cloud Computer (SCC) [1], can scale to 1,000 cores. In this kind of architecture, each core is designed as in-order processor core for power efficiency. Besides, it is based on a message passing architecture and does not provide any hardware cache coherence mechanism; hence software shared virtual memory is also considered in this case. With such massively parallel architecture, it is necessary to revisit the techniques described in this dissertation. For example, in-order processor core is different from out-of-order core. Besides, with a large number of cores in a chip, the traffic due to communication becomes significant, which should be considered when applying the proposed techniques.

III. Implementing Fences using RMWs. The design of concurrent lock-free applications requires comprehensive consideration of algorithmic concerns and architectural issues. In particular, enforcing $w \rightarrow r$ order is one common synchronization pattern [9]. For example, in Fig. 1.2, the purpose of fences (Lines 2 and 6) is to enforce the $w \rightarrow r$ order involving flag0 and flag1. However, when Dekker's algorithm is used inside an application, the fences also order other memory accesses. Since we already know the fences are used to enforce the $w \rightarrow r$ order involving flag0 and flag1, it is possible to use an atomic RMW instruction and a lightweight fence to implement the function of heavyweight fence, imposing less constraints on memory ordering.

	(a) fence		(b) RMW
1	Wr(flag0)	1	RMW(flag1)
2	**FENCE**	2	**FENCE**(load−load)
3	Rd(flag1)	3	Rd(flag1)

Figure 7.1: Implementing fence using RMW.

Fig. 7.1 shows how we can enforce $w \rightarrow r$ order using an atomic RMW instruction. Fig. 7.1(a) is the original implementation using a fence, where the processor first writes to flag0 and then reads from flag1. A fence is inserted between them to enforce $w \rightarrow r$ order. Instead, as shown in Fig. 7.1(b), we can use a RMW instruction to write to flag0, followed by a lightweight fence instruction only enforcing the load-load order. Let us see why we can enforce the $w \rightarrow r$ by using a RMW instruction. An atomic RMW instruction can be considered as a read immediately followed by a write, i.e., RMW(flag0) in Line 1 can be considered as Rd(flag0);Wr(flag0). Since there is a load-load fence in Line 2, it will order Rd(flag0) (read part of RMW) and Rd(flag1) (Line 3). Moreover, Rd(flag0);Wr(flag0) is atomic, so Wr(flag0) (write part of RMW) will also be ordered before Rd(flag1) (Line 3); otherwise, the atomicity will be violated. Therefore, the write to flag0 and the read from flag1 are ordered. The benefit of using a RMW instruction (plus a load-load fence) instead of a heavyweight fence is that, writes other than the target write before the load-load fence are not required to be drained from the write buffer.

Bibliography

[1] Intel Single-Chip Cloud Computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.

[2] Telera Tile-Gx processor. `http://www.tilera.com/products/processors`.

[3] S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, 2010.

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

[5] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 2–14, 1990.

[6] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. Bulkcompiler: high-performance sequential consistency through cooperative compiler and hardware support. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, pages 133–144, 2009.

[7] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, DAMP '09, pages 13–24, 2008.

[8] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 258–272, 2010.

[9] 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.

[10] 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.

[11] 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.

[12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 509–520, 2012.

[13] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 55–66, 2011.

[14] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical report, 1991.

[15] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking c++ concurrency. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP '11, pages 113–124, 2011.

[16] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 233–244, 2009.

[17] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 68–78, 2008.

[18] T. Braun, A. Condon, A. J. Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving sequential consistency by model checking. Technical report, Vancouver, BC, Canada, Canada, 2001.

[19] S. Burckhardt, R. Alur, and M. M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *CAV. Volume 4144 of LNCS*, pages 489–502. Springer, 2006.

[20] 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.

[21] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 107–120, 2008.

[22] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 104–123, 2010.

[23] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 122–132, 2011.

[24] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 152–164, 1991.

[25] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 278–289, 2007.

[26] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 97–108, 2007.

[27] 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.

[28] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. In *in Computer Aided Verification, Lecture Notes in Computer Science*, page 02. Springer-Verlag, 2002.

[29] W.-Y. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency for spmd programs with arrays. Technical Report UCB/CSD-03-1272, EECS Department, University of California, Berkeley, Sep 2003.

[30] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan. Fast complete memory consistency verification. In *HPCA '09*, pages 381–392, 2009.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2nd edition, 2001.

[32] A. DeOrio, I. Wagner, and V. Bertacco. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In *HPCA '09*, pages 405–416, 2009.

[33] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 157–168, 2009.

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

[35] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew. Detecting and eliminating potential violations of sequential consistency for concurrent C/C++ programs. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 25–34, 2009.

[36] 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.

[37] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 320–328, 1998.

[38] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 245–255, 2007.

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

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

[41] 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.

[42] M. Galluzzi, E. Vallejo, A. Cristal, F. Vallejo, R. Beivide, P. Stenström, J. E. Smith, and M. Valero. Implicit transactional memory in kilo-instruction multiprocessors. In *Proceedings of the 12th Asia-Pacific conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 339–353, 2007.

[43] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Specifying system requirements for memory consistency models. *Technical Report CSL-TR-93-594, Stanford University*, 1993.

[44] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 316–326, 1991.

[45] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS '91, pages 245–257, 1991.

[46] 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*, pages 355–364, 1991.

[47] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, 1990.

[48] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 179–188, 2002.

[49] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, pages 162–171, 1999.

[50] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–113, 2004.

[51] 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.

[52] M. D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, 1998.

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

[54] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. In *Proceedings of the IEEE*, pages 498–507, 1999.

[55] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in titanium. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, page 15, 2005.

[56] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 130–141, 2002.

[57] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, 1994.

[58] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 13–21, 1992.

[59] A. Krishnamurthy and K. Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 196–204, 1995.

[60] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *J. Parallel Distrib. Comput.*, 38(2):130–144, Nov. 1996.

[61] 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.

[62] 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.

[63] 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.

[64] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

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

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

[67] K. Lee, X. Fang, and S. P. Midkiff. Practical escape analyses: how good are they? In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 180–190, 2007.

[68] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[69] 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.

[70] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 210–221, 2010.

[71] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[72] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 495–512, 2012.

[73] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 378–391, 2005.

[74] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: a simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 351–362, 2010.

[75] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an SC-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 199–210, 2011.

[76] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 73–82, 2006.

[77] 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.

[78] 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.

[79] S. P. Midkiff. Dependence analysis in parallel loops with i±k subscripts. In *LCPC*, pages 331–345, 1995.

[80] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing, Volume 2: Software*, pages 105–113, Urbana-Champaign, IL, USA, 1990.

[81] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware support for detecting sequential consistency violations dynamically. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 363–375, 2012.

[82] T. Ogasawara, H. Komatsu, and T. Nakatani. TO-Lock: Removing lock overhead using the owners' temporal locality. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 255–266, 2004.

[83] T. Onodera, K. Kawachiya, and A. Koseki. Lock reservation for Java reconsidered. In *ECOOP '04*, pages 559–583, 2004.

[84] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, 2009.

[85] X. Qian, J. Torrellas, B. Sahelices, and D. Qian. Volition: scalable and precise sequential consistency violation detection. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 535–548, 2013.

[86] 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 '01, pages 294–305, 2001.

[87] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 199–210, 1997.

[88] 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.

[89] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 311–322, 2012.

[90] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 175–186, 2011.

[91] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 379–391, 2009.

[92] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th annual international symposium on Computer architecture*, ISCA '87, pages 234–243, 1987.

[93] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, 2008.

[94] 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*, 53(7):89–97, July 2010.

[95] 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.

[96] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient processor support for DRFx, a memory model with exceptions. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 53–66, 2011.

[97] 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.

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

[99] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 3–14, 2009.

[100] V. Vafeiadis and F. Z. Nardelli. Verifying fence elimination optimisations. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 146–162, 2011.

[101] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 65–74, 2010.

[102] 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.

[103] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 306–316, 2011.

[104] J. Ševčík and D. Aspinall. On validity of program transformations in the java memory model. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 27–51, 2008.

[105] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 43–54, 2011.

[106] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.

[107] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 266–277, 2007.

[108] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, 1995.

[109] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.