Efficient Sequential Consistency Using Conditional Fences

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

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

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

ABSTRACT

Among the 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 the most intuitive and enables programmers to reason about their parallel programs the best. Nevertheless, processor designers often choose to support relaxed memory consistency models because the weaker ordering constraints imposed by such models allow for more instructions to be reordered and enable higher performance. Programs running on machines supporting weaker consistency models, can be transformed into ones in which SC is enforced. The compiler does this by computing a *minimal* set of memory access pairs whose ordering automatically guarantees SC. To ensure that these memory access pairs are not reordered, memory fences are inserted. Unfortunately, insertion of such memory fences can significantly slowdown the program.

We observe that the ordering of the minimal set of memory accesses that the compiler strives to enforce, is typically already enforced in the normal course of program execution. A study we conducted on programs with compiler inserted memory fences shows that only 8% of the executed instances of the memory fences are really necessary to ensure SC. Motivated by this study we propose the conditional fence mechanism (C-Fence) that utilizes compiler information to decide dynamically if there is a need to stall at each fence. Our experiments with SPLASH-2 benchmarks show that, with C-Fences, programs can be transformed to enforce SC incurring only 12% slowdown, as opposed to 43% slowdown using normal fence instructions. Our approach requires very little hardware support (<300 bytes of on-chip-storage) and it avoids the use of speculation and its associated costs.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Design, Performance, Experimentation

Keywords

Memory Consistency, Sequential Consistency, Interprocessor Delay, Associates, Conditional Fences, Active Table

1. INTRODUCTION

With the advent of multicores, the onus is on programmers to write parallel programs for increased performance. However, it is a well recognized fact that writing and debugging parallel programs is no easy task. Consequently, there has been significant research on developing programming models, memory models, and tools for making programmers' job easier. In particular, memory consistency models which specify guarantees on values returned by data accesses, allow programmers to reason about their parallel program. Out of the 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 the most intuitive.

```
          Initially   flag1=flag2=0
       P1                        P2

flag1=1;                 flag2=1;
if (flag2==0)            if (flag1==0)
   critical section         critical section
```

Figure 1: Effect of memory model on the Dekker's algorithm.

Consider the code segment in Fig. 1, which is taken from an implementation of the Dekker's algorithm [7, 1]. This algorithm is designed to allow only one processor to enter the critical section at a time, using two shared memory variables flag1 and flag2. The two flag variables indicate an intention on the part of each processor to enter the critical section. Initially, flag1 and flag2 are initialized to 0. When P1 attempts to enter the critical section, it first updates flag1 to 1 and checks the value of flag2. If flag2 is 0, under the most intuitive interpretation P2 has not tried to enter the critical section, and it is safe for P1 to enter. However, this reasoning is valid, only if the machine that executes the program enforces a strong consistency model like SC. If, on the other hand, the machine enforces a relaxed memory consistency model, it is possible for P1 to first read flag2 and then update flag1. Likewise, P2 may first read flag1 and then update flag2. As a result, both reads may potentially return the value of 0, allowing both P1 and P2 to enter the critical section, clearly not what the programmer intended. As we can see from the above example, the SC model enables programmers to understand and reason about their parallel programs the best. In addition to this, most of the prior work on the semantics and verification of parallel programs assumes SC. For programmers to effectively use such tools, it is necessary that SC is enforced.

(HW/SW approaches for high performance SC) In spite of the advantages of SC model, processor designers typically choose to support relaxed consistency models, as shown in Fig. 2. This is because a straightforward implementation of SC forces the processor to execute memory accesses in order, which precludes a number of optimizations. For example, *write buffering* cannot be directly utilized, if SC needs to be guaranteed. However, in recent work researchers have shown that it might be possible for architectures to support SC at a high performance using sophisticated speculative techniques [10, 11, 12, 22]. More recently, researchers have proposed *chunk based* techniques [3, 4, 26], that efficiently enforce SC at the granularity of coarse-grained chunks of instructions rather than individual instructions. While the above approaches are promising, the significant hardware changes inherent in each of the above approaches hinder widespread adoption.

Processor family	SC enforced?
IA 32/Intel 64/AMD 64	No
ARM	No
Itanium	No
SPARC	No
PowerPC	No
MIPS R10000	Yes

Figure 2: Most processors do not support sequential consistency.

Programs running on machines supporting weaker consistency models, can be transformed into ones in which SC is enforced [9, 14, 15, 16, 18, 20, 21, 24, 25]. The compiler does this by computing a *minimal* set of *memory access pairs* whose ordering automatically guarantees SC based on the delay set analysis [24]; to ensure that these memory access pairs are not reordered, memory fences are inserted at appropriate points. As shown in Fig. 3(a) a memory fence added after the update of flag1 ensures that processor $P1$ stalls until the update of flag1 completes and is visible to all processors. Likewise the memory fence added after the update of flag2 ensures that processor $P2$ stalls until this update completes. While this ensures SC, the insertion of such memory fences can significantly slowdown the program [8, 9, 14].

(Conditional Fences) We make the observation that the ordering of memory accesses that the compiler strives to enforce, is typically already enforced in the normal course of program executions; this obviates the need for memory fence instruction most of the time. For instance, let us consider the scenario shown in Fig. 3(b) in which the two processors' requests for critical section are staggered in time. In particular, by the time processor $P2$ tries to enter the critical section, the updates to flag1 by processor $P1$ has already completed. Clearly, under this scenario SC is already guaranteed without requiring the fence instructions. Furthermore, our study conducted on SPLASH-2 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 necessary to ensure SC. Motivated by this study, we propose a novel fence mechanism, known as the *C-Fence* (conditional fence) that utilizes compiler information to dynamically decide if there is a need to stall at each fence, only stalling when necessary. For each fence inserted by the compiler, let us call the corresponding fences inserted in other processors due to conflicting accesses, as its *associate* fences. Furthermore, let us call a fence to be *active*, when the memory accesses before it are yet to complete; consequently, a conventional fence that is active will stall the processor. However, the key insight here is that we can safely execute past a fence, provided *its associate fences are not active.* Thus the C-Fence, when it is going to be issued, checks to verify if its associate fences are active; if none of its associates are active, the processor is not made to stall at the C-Fence. If however, some of its associates are active, the C-Fence stalls until all of its associates become inactive. Fig. 3(c) illustrates the working of a C-Fence. The two fences in question are *associates*, since they are introduced due to the same conflicting variables (the flag variables). Initially, when *C-Fence 1* is issued at time t_1, it does not stall since its associate fence *C-Fence 2* is not active. However, when *C-Fence 2* is issued at time t_3, it is made to stall, since *C-Fence 1* is still active at that time. Consequently, *C-Fence 2* stalls the processor until time t_2, at which point *C-Fence 1* becomes inactive. However, if the processors' request for critical sections are staggered, as in Fig. 3(b), then by the time *C-Fence 2* is issued, the first fence would have already been inactive. This would mean that in this scenario, even *C-Fence 2* would not stall. Indeed, this situation being the common case, as our study shows, C-Fence is able to greatly minimize stalling. Our experiments conducted on the SPLASH-2 programs confirms that with C-Fences, programs can be transformed to enforce SC incurring only 12% slowdown, as opposed to 43% slowdown using conventional fence instructions.

The main contributions of this paper are as follows.

- We propose a novel fence mechanism known as the C-Fence (conditional fence), which can be used by the compiler to enforce SC, while incurring negligible performance overhead.

- Compared to previous **software** based approaches, our approach significantly reduces the slowdown. While the best known prior implementation causes an overhead of 45% [8] for the SPLASH-2 benchmarks, we reduce the overhead to only 12%.

- Compared to previous **hardware** approaches towards SC which require speculation and its associated hardware costs, our approach does not require any speculation. Our approach merely requires less than 300 bytes of on-chip storage for the implementation of the C-Fence mechanism.

2. BACKGROUND

2.1 Sequential Consistency

Memory consistency models [1] constrain memory behaviors with respect to read and write operations from multiple

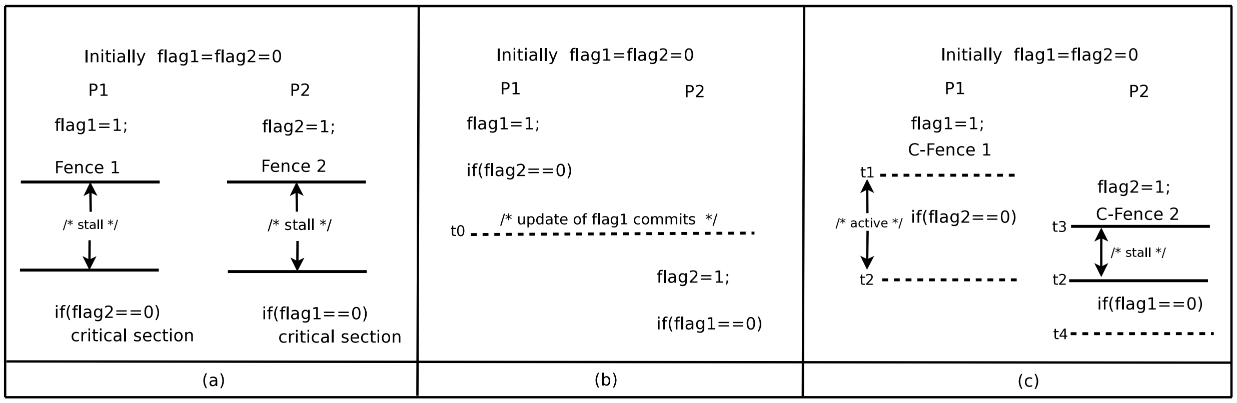


Figure 3: (a) SC using memory fences. (b) Scenario in which fences are not required. (c) SC using conditional fences.

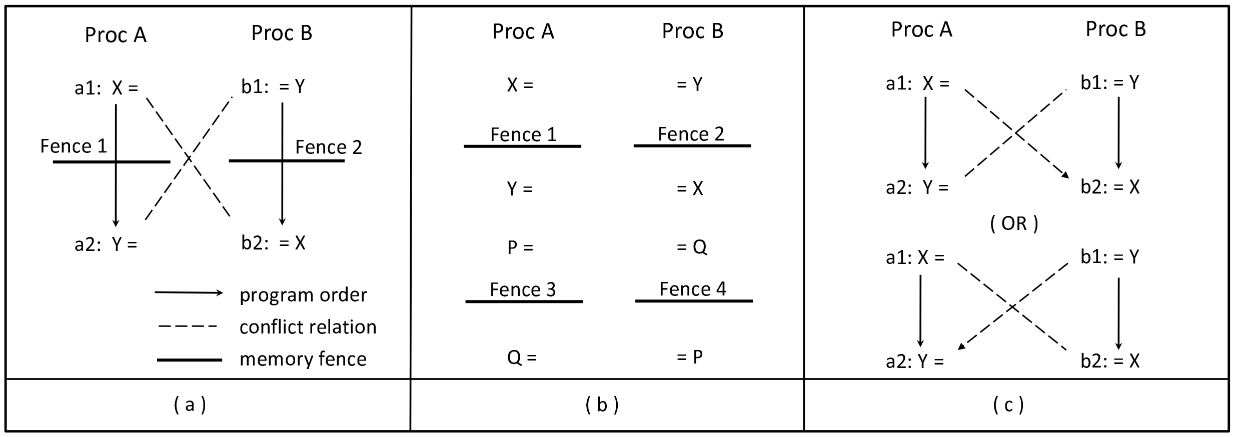


Figure 4: (a) Memory fences added to ensure SC. (b) $Fence_1$ and $Fence_2$, added to ensure SC of conflicting accesses to X and Y are associates; likewise, $Fence_3$ and $Fence_4$ are associates. (c) Interprocessor delay is able to ensure SC.

processors, providing formal specifications of how the memory system will appear to the programmer. Each of these models offers a trade-off between programming simplicity and high performance. Sequential Consistency (SC) [17] 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. SC is easy to understand, but it can potentially affect the performance of programs. The strict memory ordering imposed by SC can restrict many hardware and compiler optimizations that are possible in uniprocessors. Indeed, most of the processor families do not support SC as shown in Fig. 2.

2.2 Software Based Enforcement of SC

(Naive insertion of fences) While most processors do not enforce SC automatically, they provide support in the form of *memory fences*, which the compiler can utilize to enforce SC. A fence instruction ensures that all memory operations prior to it have completed, before the processor issues memory instructions that follow it. In the most naive scheme, inserting a fence after every memory instruction can guarantee SC. However, the memory fence being an expensive instruction requiring tens of cycles to execute, the excessive use of fences can significantly hurt performance.

(Delay set analysis) Various compiler based approaches [8, 9, 14, 18, 25] have been proposed to minimize the number of fences required to enforce SC. These approaches are generally based on a technique called delay set analysis, developed by Shasha and Snir [24], which finds a minimal set of execution orderings that must be enforced to guarantee SC. Delay set analysis identifies the points where the execution order may potentially violate the program order and inserts memory fences to prevent this violation. To identify these fence insertion points, the analysis utilizes information about conflicting memory accesses. 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. Fig. 4(a) shows a conflict relation between variables X and Y.

Let $\mathbf{C}$ denote the conflict relation and $\mathbf{P}$ the program order, then $\mathbf{P}\cup\mathbf{C}$ is known as the *conflict graph.* An *execution order*($\mathbf{E}$) is an orientation of the conflict relation, for a particular execution. *It is worth noting that a given execution order* $\mathbf{E}$ *violates SC, if there is a cycle in* $\mathbf{P}\cup\mathbf{E}$. A cycle in the conflict graph indicates that there is an execution order that is not consistent with the program order. Among such cycles, those cycles that do not have chords in $\mathbf{P}$ are known as *critical cycles.* Fig. 4(a) shows a critical cycle in the conflict graph $(a_1, a_2, b_1, b_2, a_1)$. Note that this cycle indicates a possible execution sequence (a_2, b_1, b_2, a_1) which violates

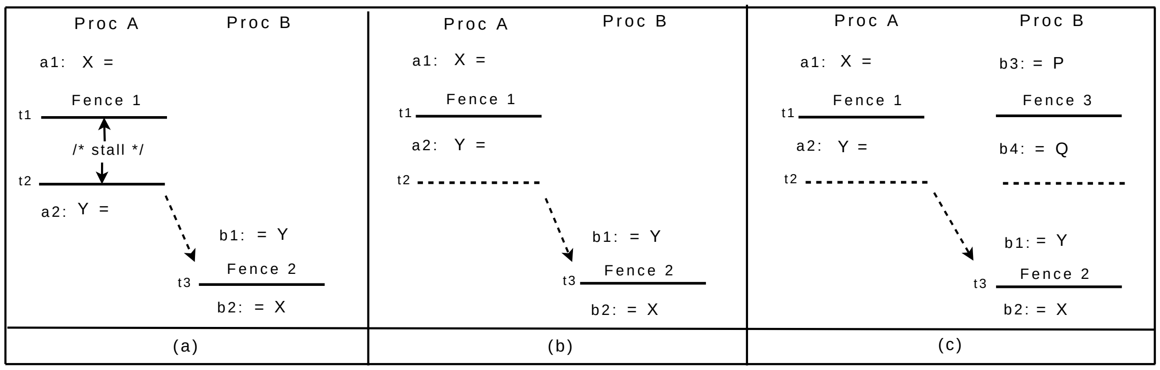


Figure 5: (a) If $Fence_1$ has already finished stalling by the time $Fence_2$ is issued, then there is no need for $Fence_2$ to stall. (b) In fact, there is no need for even $Fence_1$ to stall, if all memory instructions before it complete by the time $Fence_2$ is issued. (c) No need for either $Fence_1$ or $Fence_3$ to stall as they are not associates, even if they are executed concurrently.

SC, in which b_1 reads the new value of Y, but b_2 still reads the old value of X.

The compiler-based approaches break such critical cycles by enforcing *delays* between memory operations. For example, if the issuing of second instructions in pairs (a_1, a_2) and (b_1, b_2) is delayed until the first completes, SC will be enforced. Hence, a delay relation $\mathbf{D}$ is introduced, in which $u\mathbf{D}v$ indicates that u must complete before v is issued. The delays are satisfied by inserting memory fences into the program as shown in Fig. 4(a). We call those fences as *associate fences* or simply *associates*, if they are inserted to ensure the delays that appear in the same cycle in the conflict graph $\mathbf{P} \cup \mathbf{C}$. As we can see from Fig. 4(b), $Fence_1$ and $Fence_2$ which are added to ensure SC of conflicting accesses to X and Y are associates; similarly, $Fence_3$ and $Fence_4$ are associates. It is worth noting that a critical cycle can involve more than two processors [24], in which case more than two fences are associates.

(Program slowdown) Lee and Padua [18] 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. [9] 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 [8, 9, 14].*

3. OUR APPROACH

Next we describe our approach for simultaneously achieving SC and high performance. We first motivate our approach by showing that fences introduced statically may be superfluous (dynamically) for the purpose of enforcing SC. We then describe our empirical study which shows that most fences introduced statically are indeed superfluous. We then take advantage of this property by proposing and utilizing C-Fence (conditional fence) for enforcing SC.

3.1 Interprocessor Delays to Ensure SC

In spite of several optimizations proposed, statically introduced fences still cause significant performance overhead. While fences inserted by the compiler are sufficient to guarantee SC, it might be possible to remove the fences dynamically and still ensure that SC is guaranteed. More specifically, we observe that if two fences are staggered during the course of program execution, they may not need to stall.

Let us consider the critical cycle shown earlier in Fig. 4(a). Recall that the compiler inserts delays (through memory fences) to ensure SC. According to the previous definition, $Fence_1$ and $Fence_2$ are associates. $Fence_1$, 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 ensure that there are no cycles in the conflict graph, thereby ensuring SC. While each of the above two *intraprocessor* delays are needed to enforce SC, we observe that SC 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, SC is ensured. In other words, either $a_1\mathbf{D}b_2$ or $b_1\mathbf{D}a_2$ is sufficient to guarantee SC. To see why let us consider Fig. 4(c) that shows the execution ordering resulting from $a_1\mathbf{D}b_2$. With this ordering, we can now see that $\mathbf{P} \cup \mathbf{E}$ becomes acyclic and thus SC is ensured. Likewise, $b_1\mathbf{D}a_2$ makes the graph acyclic. *Thus, for every critical cycle, even if one of these interprocessor delays is ensured, then there is no need to stall in either of the compiler generated fences.* It is easy to prove the correctness by exploring that no critical cycles will exist when interprocessor delays are enforced.

We now describe execution scenarios in which one of these interprocessor delays is naturally ensured, which obviates the need for stalling at fences. Fig. 5(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 SC. 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. 5(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

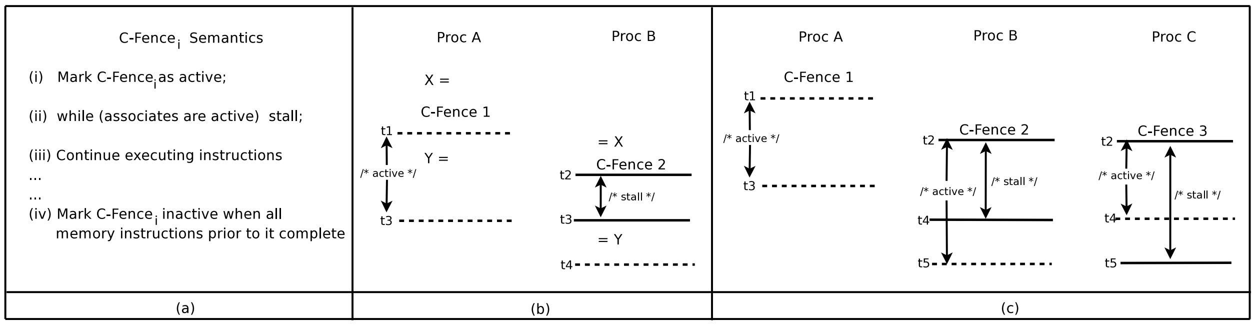


Figure 6: (a) The semantics of C-Fence. (b) Example with 2 C-Fences. (c) Example with 3 C-Fences.

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. 5(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 SC 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.*

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 1: Study: A significant percentage of statically inserted fences need not stall.

(Study) We showed examples of why fences that were inserted statically may not be necessary (i.e., they need not stall) dynamically. We wanted to conduct a study to check empirically how often the fences inserted statically are not required to stall. For this study, we used the SPLASH-2 benchmarks and inserted fences using Shasha and Snir's delay set analysis. For each benchmark, 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 from Table 1, about 92% of the total fences executed do not need to stall. This motivates our technique for taking advantage of this observation and reducing the time spent stalling at fences.

3.2 C-Fence: Conditional Fence

We propose an efficient fence mechanism known as *conditional fence* (C-Fence). As opposed to the conventional memory fence that is used to ensure SC through intraprocessor delays between memory instructions, a C-Fence ensures SC through interprocessor delays. 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. It is important to note that ensuring SC via C-Fences does not require any specialized compiler analysis. The same delay set analysis is used to insert C-Fences at the same points as conventional memory fences. The only difference is that we provide 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.

(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. 6(a)).

Fig. 6(b) 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. 6(c). 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,

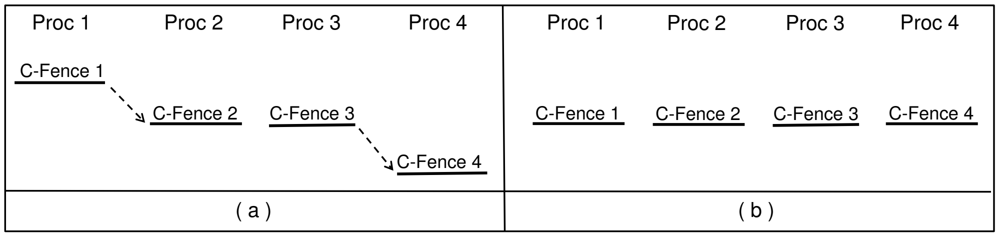


Figure 7: Scalability of C-Fence.

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.

(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 our 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. 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. 7(b). In fact, our experiments do confirm that even with 16 processors the C-Fence mechanism performs significantly better than the conventional fence.

4. C-FENCE HARDWARE

4.1 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.

4.2 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 is our 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, in our study we found that just 50 static fences account for 90% of dynamic execution count as shown in Fig. 8.

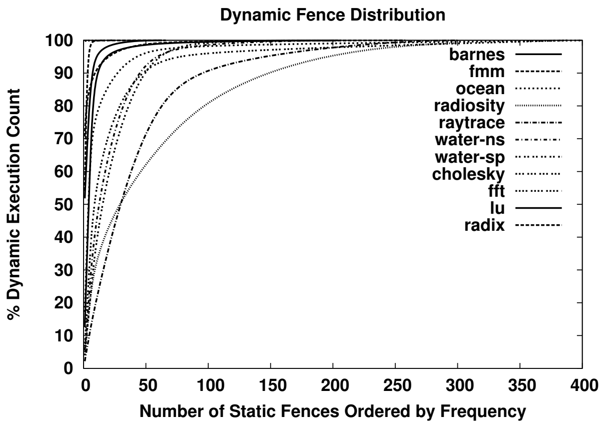
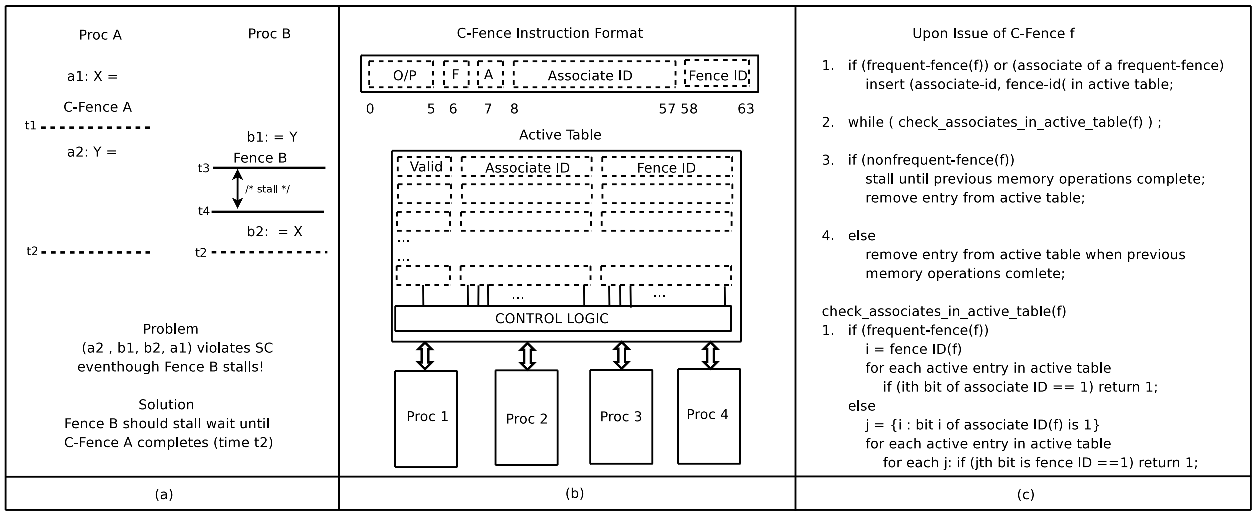


Figure 8: Distribution of fences

Thus, in our 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, 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



```
Upon Issue of C-Fence f

1.  if (frequent-fence(f)) or (associate of a frequent-fence)
        insert (associate-id, fence-id( in active table;

2.  while ( check_associates_in_active_table(f) ) ;

3.  if (nonfrequent-fence(f))
        stall until previous memory operations complete;
        remove entry from active table;

4.  else
        remove entry from active table when previous
        memory operations comlete;

check_associates_in_active_table(f)
1.  if (frequent-fence(f))
        i = fence ID(f)
        for each active entry in active table
            if (ith bit of associate ID == 1) return 1;
    else
        j = {i : bit i of associate ID(f) is 1}
        for each active entry in active table
            for each j: if (jth bit is fence ID ==1) return 1;
```

Figure 9: (a) C-Fence + conventional Fence. (b) HW Support. (c) Action upon issue of a C-Fence.

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. 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 SC. 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 is used to determine the N most frequent fences; in our 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. 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 our 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.

(CFence: 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. 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. 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. 9(c).

(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

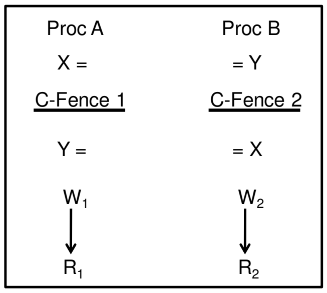


Figure 10: Coherence of active-table.

our 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. 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. 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 inorder. 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.

(CFence: 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 instruction addresses 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 did experiments and found that 5 entries are enough, which results in a hit rate of 94.22%.

5. EXPERIMENTAL EVALUATION

We performed our 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 SC is concerned. Second, we also wanted to evaluate as to how close the performance of our 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 our 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. However, before we present the results of our evaluation, we briefly describe our implementation.

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 2: Architectural Parameters.

We implemented our C-Fence mechanism in the SESC [23] 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 our implementation are presented in Table 2. The default architectural parameters were used in all experiments unless explicitly stated otherwise. The simulator uses release consistency (RC) as its memory consistency model. To enforce SC we inserted fences by identifying fence insertion points using Shasha and Snir's delay set 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 [8].

5.2 Benchmark Characteristics

We used the SPLASH-2 [27], a standard multithreaded suite of benchmarks for our evaluation. We could not get the program *volrend* to compile using the compiler infrastructure that targets the simulator and hence we omitted *volrend* from our experiments. We used the input data sets prescribed in [27] and ran the benchmarks to completion. Table 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 our 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 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.

Benchmark	# of static fences	Avg # of associates	# of dynamic fences($\times$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 3: Benchmark Characteristics.

5.3 Conventional Fence vs C-Fence: Execution Time Overhead

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 our actual hardware implementation with 50 frequent fences and 20 active-table entries. Fig. 11 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.

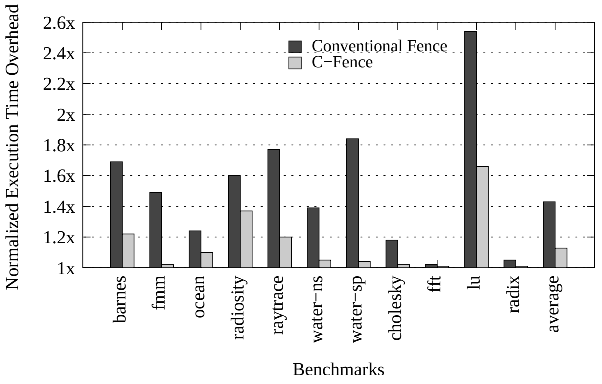


Figure 11: Execution time overhead of ensuring SC: Conventional fence vs C-Fence.

5.4 Sensitivity Studies

(Sensitivity towards number of frequent fences) Recall that our 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. 12 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.

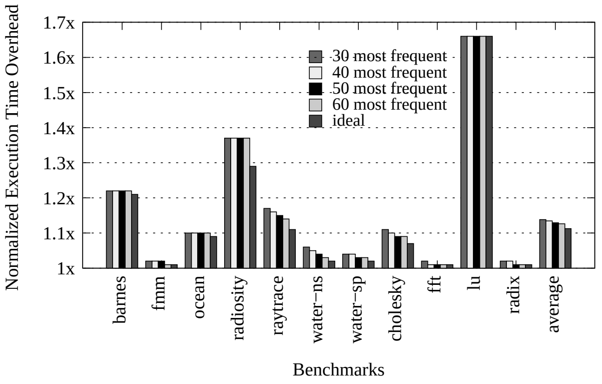


Figure 12: Varying the # of frequent fences.

(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

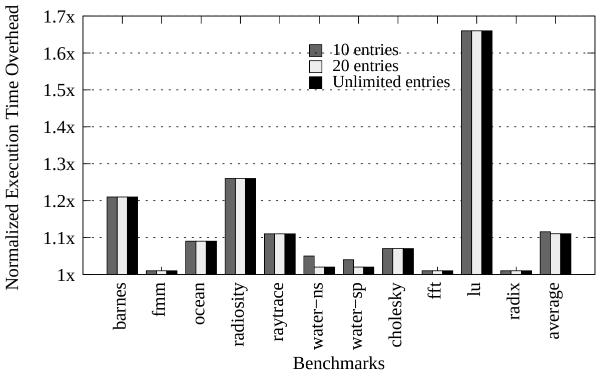


Figure 13: Varying the size of active table.

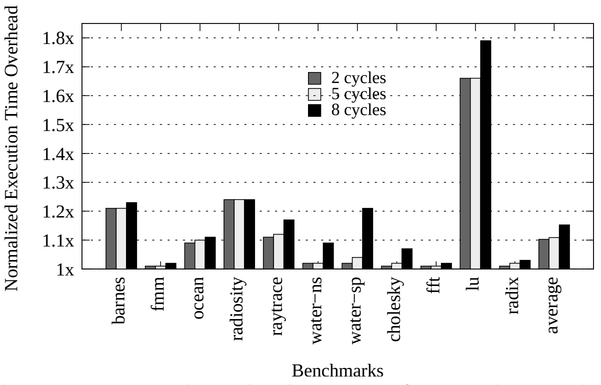


Figure 14: Varying the latency of accessing active-table.

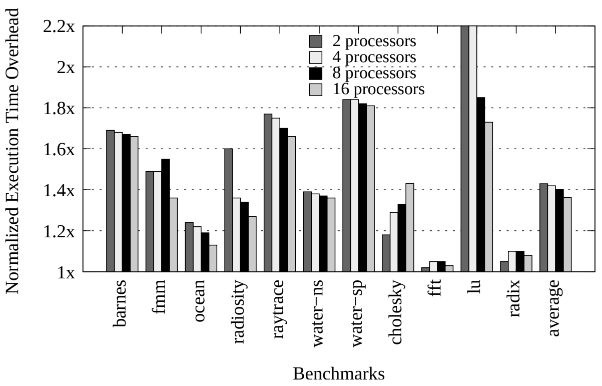


Figure 15: Conventional fence performance for 2, 4, 8 and 16 processors.

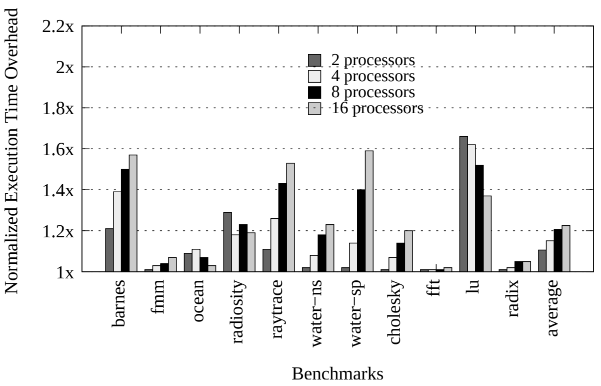


Figure 16: C-Fence performance for 2, 4, 8 and 16 processors.

of entries of the active-table that can achieve performance close to the ideal. 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. 13, 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.

(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. 14. 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.

(Sensitivity towards number of processors) Here, we wanted to study the performance variation as the number of processors is varied. Fig. 15 shows the execution time overhead of the conventional fence mechanism as the number of processors is varied across 2, 4, 8 and 16 processors. In general, we can see that the performance remains almost the same as the number of processors is varied. Fig. 16 shows the performance of the C-Fence mechanism as the number of processors is varied. 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, the execution time overhead increases from 1.12x for 2 processors, to 1.15x for 4 processors, to 1.2x for 8 processors and 1.22x for 16 processors. This is because, as the number of processors increases, there is a greater chance of associates executing concurrently, which in turn require each individual fences to stall. However, we can also observe that the execution time increases are modest. Consequently, even for 16 processors the C-Fence mechanism is able to reduce the execution time overhead significantly compared to the conventional fence (from 1.38x to 1.22x). Thus, while the relative performance gain reduces as the number of processors increase, the C-Fence mechanism still performs significantly better than conventional fence.

5.5 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. 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 $2 \times 50 + 1 = 101$ bits. Hence, 20 entries amount to 252 bytes. In addition, we use a 5 entry buffer (requiring 40 bytes) which caches the fence instruction addresses to speedup the requests to the active-table. Therefore, we would require an on-chip storage of less than 300 bytes.

6. RELATED WORK

There have been several techniques proposed by researchers to speed up the implementation of SC. These techniques can be categorized into two classes: *hardware-based* and *software-based.*

(Hardware based implementations) In hardware-based implementations, conventional hardware design is modified to accommodate the requirement of SC. Speculation is the technique that is used in the hardware-based implementations to achieve high performance despite ensuring SC. Gharachorloo et al. [10] proposed two techniques to enhance the performance of SC: *hardware-controlled non-binding prefetch* and *speculative execution for load accesses.* Ranganathan et al. [22] proposed speculative retirement. 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 [12], Gniady et al. proposed *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. The above works showed that SC implementations can perform as well as RC implementations if the hardware provides enough support for speculation. Another set of techniques [3, 4, 26] proposed the idea of enforcing consistency at the granularity of coarse-grained chunks of instructions rather than individual instructions. [13, 5] first introduced this concept in transactional memory, although they do not target SC. Ceze et al. [4] proposed 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. [2] proposed a compiler algorithm called *BulkCompiler* to drive the group-formation operation and adapt code transformations to existing chunk-based implementations of SC. Blundell et al. [3] proposed InvisiFence, which does not require either fine-grained buffers to hold speculative state or require global arbitration for commit in speculation state compared with the two classes mentioned above. Although hardware-based implementations of SC can achieve good performance comparable to RC, they need extra hardware support that is not widely available in current processors. Furthermore, each of the HW techniques requires extensive speculation. *The C-Fence mechanism, in comparison with the hardware based implementations, utilizes the compiler inserted fence instructions to achieve SC. Furthermore, it requires no speculation and the associated hardware costs. The only additional HW required is the active-table and a small buffer which amounts to less than 300 bytes of on-chip storage, which is significantly lesser than the above HW techniques.*

(Software based Implementations) Shasha and Snir's work [24] originally proposed delay set analysis which finds a minimal set of delays that enforces sequential consistency. Midkiff and Padua [21] extended Shasha and Snir's characterization to work for programs with array accesses. Krishnamurthy and Yelick [16, 15] provided some early implementation work on cycle detection, which was improved later by Chen et al [6]. Lee and Padua [18] 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. [9] also developed and implemented several fence insertion and optimization algorithms in their Pensieve compiler project. Sura et al. [25] described co-operating escape, thread structure, and delay set analysis to enforce SC. In [19], Lee et al. used a perfect escape analysis to provide the best bound on the overhead incurred for a SC memory model. The delays considered in all the above works are intraprocessor delays, while in this work we introduce interprocessor delays for ensuring SC. This is because interprocessor delays also known in prior work as orientation of a conflict edge [20] was considered too expensive [24]. *The main issue with the software based implementation is performance overhead. The C-Fence mechanism can be used in conjunction with the software based techniques to reduce the performance overhead.* In [8], Duan et al. detected data races by running the program instead of finding conflict accesses by static program analysis. The above technique is more a testing based solution for SC and is orthogonal to this work. In fact, the above work can utilize C-Fence to further improve their performance.

7. CONCLUSION

Memory fences can be used by the compiler to enforce sequential consistency (SC) of a given program. However, the inserted memory fences can significantly reduce the program performance. In this paper, we proposed a novel fence mechanism known as C-Fence that can be used by the compiler to ensure SC at a significantly lesser performance overhead. While conventional fence enforces *intraprocessor* delays to prevent memory reordering, the C-Fence enforces *interprocessor* delays to ensure SC. However, we observe that most of the interprocessor delays that are required for the enforcement of SC, 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 does not require any novel compiler analysis, it works with currently used fence insertion techniques. 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%.

ACKNOWLEDGEMENTS

This work is supported by NSF grants CCF-0963996, CCF-0905509, CNS-0751961, and CNS-0810906 to the University of California, Riverside.

8. REFERENCES

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

[2] 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 MICRO-42*, pages 133–144, New York, NY, USA, 2009. ACM.

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

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

[5] 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 *HPCA-13*, pages 97–108, Washington, DC, USA, 2007. IEEE Computer Society.

[6] W.-Y. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *LCPC*, pages 2–4. Springer-Verlag, 2003.

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

[8] 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 *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 25–34, Washington, DC, USA, 2009. IEEE Computer Society.

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

[10] 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.

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

[12] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of ISCA-26*, pages 162–171, Washington, DC, USA, 1999. IEEE Computer Society.

[13] 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. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.

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

[15] A. Krishnamurthy and K. Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

[16] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38, 1996.

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

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

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

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

[21] 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.

[22] P. Ranganathan, V. Pai, and S. 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*, page pages, 1997.

[23] 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.

[24] 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.

[25] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–13, New York, NY, USA, 2005. ACM.

[26] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of ISCA-34*, pages 266–277, New York, NY, USA, 2007. ACM.

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