Location-Based Memory Fences

Edya Ladan-Mozes I-Ting Angelina Lee
MIT CSAIL
32 Vassar Street, Cambridge, MA 02139
{edya, angelee}@csail.mit.edu

Dmitry Vyukov*
OOO Google
7 Balchug Street, Moscow, 115035, Russia
dvyukov@google.com

ABSTRACT

Traditional memory fences are program-based. That is, a memory fence enforces a serialization point in the program instruction stream — it ensures that all memory references before the fence in the program order have taken effect before the execution continues onto instructions after the fence. Such program-based memory fences always cause the processor to stall, even when the synchronization is unnecessary during a particular execution. We propose the concept of ***location-based memory fences***, in which a processor executing a concurrent algorithm incurs synchronization cost due to latency of memory fences only when the synchronization is needed.

Unlike a program-based memory fence, a location-based memory fence serializes the instruction stream of the executing thread T_1 only when a different thread T_2 attempts to read the memory location which is guarded by the location-based memory fence. In this work, we describe a hardware mechanism for location-based memory fences, prove its correctness, and evaluate its potential performance benefit. Our experiments show that applications perform better even when using an inefficient software implementation of location-based memory fences than when using program-based memory fences. The inefficiency of the software implementation stems from the cost of software communication between T_1 and T_2, which is greatly reduced in the hardware mechanism proposed in this paper. Furthermore, the proposed hardware mechanism is lightweight, and requires only a small modification to existing architectures. Thus, we believe that the proposed hardware design for location-based memory fences is a viable and appealing alternative to traditional program-based memory fences.

Categories and Subject Descriptors: C.1.m [Processor Architectures]: Miscellaneous; D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel programming*

General Terms: Design, Performance, Theory

*This work was conducted by the author outside of Google.

This research was supported in part by the National Science Foundation under Grant CNS-1017058 and in part by the Angstrom Project funded by the Defense Advanced Research Projects Agency UHPC program under Agreement Number HR0011-10-9-0009.

SPAA'11, June 4–6, 2011, San Jose, California, USA.


```
Initially L1 = L2 = 0;

        Thread 1                         Thread 2
T1.1  L1 = 1;                    T2.1  L2 = 1;
T1.2  if(L2 == 0) {              T2.2  if(L1 == 0) {
T1.3    /* critical              T2.3    /* critical
T1.4       section */            T2.4       section */
T1.5  }                          T2.5  }
T1.6  L1 = 0;                    T2.6  L2 = 0;
```

Figure 1: A simplified version of the Dekker protocol (omitting the mechanism to allow the threads to take turns), assuming sequential consistency.

Keywords: location-based memory fences, memory fences, asymmetric synchronization, the Dekker duality, the Dekker protocol, biased locks

1. INTRODUCTION

On many modern multicore architectures, threads[1] typically communicate and synchronize via shared memory. Classic synchronization algorithms such as Dekker [10], Dijkstra [9], Lamport (Bakery) [18], and Peterson [22] use simple load-store operations on shared variables to achieve mutual exclusion among threads. All these algorithms employ an idiom, referred as the ***Dekker duality*** [6], in which every thread writes to a shared variable to indicate its intent to enter the critical section and reads the other's variable to coordinate access to the critical section.

Crucially, the correctness of such idiom rely on that the memory model exhibits ***sequential consistency*** (SC) [19], where all processors observe the same sequence of memory accesses, and within this sequence, the accesses made by each processor appear in its program order. While the SC memory model is the most intuitive to the programmer, existing architectures typically implement weaker memory models that relax the memory ordering to achieve higher performance. The reordering affect the correctness of the software execution in some cases such as the Dekker duality, where it is crucial that the execution follows the program order, and the processors observe the relevant accesses in the same relative order.

Consider the following code segment shown in Figure 1, which is a simplified version of the Dekker protocol [10][2] using the idiom to synchronize access to the critical section among two threads. With "Total-Store-Order" and "Processor-Order" memory models,

[1]Throughout the paper, we use the terms thread and processor interchangeably. In particular, we use thread in the context of describing an algorithm and processor in the context of describing hardware features.

[2]This simplified version is vulnerable to livelock, where both threads simultaneous try to enter the critical section — each thread sets its own flag, reads the other thread's flag, retreats, and retries. Without some way of breaking the tie, the two threads can repeatedly conflict with each other and retry perpetually. The full version is augmented with mechanism to allow the threads to take turns and thus guarantees progress. For the sake of clarity, we present the simplified version here.

which are the memory models we consider in this paper, the read in line T1.2 may get reordered with the write in line T1.1 (and similarly for Thread 2), such that Thread 2 observes the read of Thread 1 (line T1.2) before it observes the write of Thread 1 (line T1.1). Thus, Thread 1 and Thread 2 observe different ordering of the reads and writes, resulting in an incorrect execution and causing the two threads to enter the critical section concurrently.

To ensure a correct execution in such cases, architectures that implement weak memory models provide serializing instructions and memory fences to force a specific memory ordering when necessary. Thus, a correct implementation of the Dekker protocol for such systems would require a pair of memory fences between the write and the read (between lines T1.1 and T1.2, and lines T2.1 and T2.2 in Figure 1), ensuring that the write becomes globally visible to all processors before the read is executed.

These memory fences are program-based — they are part of the code the processor is executing, and they cannot be avoided even when the program is executed serially, or when the synchronization is unnecessary because no other threads are reading the updated memory location. Furthermore, when a memory fence is executed, the processor stalls until all outstanding writes before the fence in the instruction stream become globally visible. Thus, memory fences are costly, taking many more cycles to complete than regular reads and writes. For example, on AMD Opteron with 4 quad-core and 2 GHz CPUs, a thread running alone and executing the Dekker protocol with a memory fence, accessing only a few memory locations in the critical section, runs $4-7$ times slower than when it is executing the same code without a memory fence.

In this work we propose a ***location-based memory fence***, which causes the executing thread T_1 to "serialize" only when another thread T_2 tries to access the memory location associated with the memory fence. Location-based memory fences aim to reduce the latency in program execution incurred by memory fences. Unlike a program-based memory fence, a location-based memory fence is *conditional* and *remotely enforced* by T_2 onto T_1; whether T_1 serializes or not depends on whether there exists T_2 that attempts to access the memory location associated with the memory fence. In essence, location-based memory fences allow T_1 to avoid the latency of memory fences and instead have T_2 born the overhead of communication to trigger T_1 to serialize. Performance benefit is obtained if the latency avoided by T_1 is greater than the communication overhead born by T_2.

The concept of location-based memory fences is particularly well suited for applications that employ the Dekker duality. It turns out that this idiom is commonly used to optimize applications that exhibit ***asymmetric synchronization patterns***, where one thread, the ***primary thread***, enters a particular critical section much more frequently than other threads running in the same process, referred as the ***secondary threads***. Such applications typically employ an augmented version of the Dekker protocol: the secondary threads first compete for the right to synchronize with the primary thread (by grabbing a lock); once obtaining the right, the winning secondary thread synchronizes with the primary thread using the Dekker protocol. The augmented Dekker protocol intends to speedup the execution path of the primary thread, even at the expanse of the secondary threads. In such applications, it is also desirable to optimize away the overhead of fences on the primary thread's execution path when the application executes serially or when there is no contention.

Many examples of such applications exist. For example, Java Monitors are implemented with biased locking [7, 16, 21], which uses an augmented version of the Dekker protocol to coordinate between the bias-holding thread (primary) and a revoker thread (secondary). Java Virtual Machine (JVM) employs the Dekker duality to coordinate between mutator threads (primary) executing outside of JVM (via Java Native Interface) and the garbage collector (secondary) [7]. In a runtime scheduler that employs a work-stealing algorithm [2–5, 11, 12, 17], the "victim" (primary) and a given "thief" (secondary) coordinate the steal using an augmented Dekker-like protocol. Finally, in network package processing applications, each processing thread (primary) maintains its own data structures for its group of source addresses, but occasionally, a thread (secondary) might need to update data structures maintained by a different thread [23].

Such applications motivate our study of location-based memory fences. We propose a hardware mechanism to implement location-based memory fences, which aims to be lightweight and requires only modest modifications to existing hardware: two additional registers per processor and a new instruction, which implements a functionality that many modern architectures already support. With this hardware design for location-based memory fences, a thread running alone and executing the Dekker protocol will observe only negligible overhead when using location-based memory fences compared to executing the same code without fences at all.

We further evaluated the feasibility of location-based memory fences using a software prototype to simulate its effect and applied it in two applications that exhibit asymmetric synchronization patterns. While the software implementation incurs much higher communication overhead than the proposed hardware mechanism would, our experiments show that applications still benefit from the software implementation and would scale better if the communication overhead were smaller. Based on these results, we believe that the proposed hardware design for location-based memory fences is a viable and appealing alternative to traditional program-based memory fences.

The rest of the paper is organized as follows. Section 2 gives an abbreviated background on why reordering occurs in architectures that support a weaker memory model. Section 3 presents the proposed hardware mechanism for location-based memory fences. Section 4 formally defines the specification of location-based memory fences, proves that the proposed hardware mechanism implements the specification, and lastly shows that the Dekker protocol using location-based memory fences provides mutual exclusion. Section 5 evaluates the feasibility of location-based memory fences using a software prototype implementation with two applications. Section 6 gives a brief overview on related work. Finally, Section 7 draws concluding remarks.

2. STORE BUFFERS AND MEMORY ACCESSES REORDERING

In this section, we summarize features of modern architecture design, which are necessary for our proposed hardware mechanism for location-based memory fences. In particular, throughout the rest of the paper, we assume that the target architecture implements either the Total-Store-Order (TSO) model (implemented by SPARC-V9 [24]) or the Process-Ordering (PO) model (implemented by Intel 64, IA-32 [15], and AMD64 architectures [1]). We also describe how ***memory reordering*** can occur, i.e., how the observable order in which memory locations are accessed can differ from the program order. Memory reordering can be introduced either by the compiler or the underlying hardware. Compiler fences that prevent the compiler from reordering have relatively small overhead, whereas the memory fences that prevent the reordering at the hardware level are much more costly. In this section, we focus on the reordering at the hardware level.

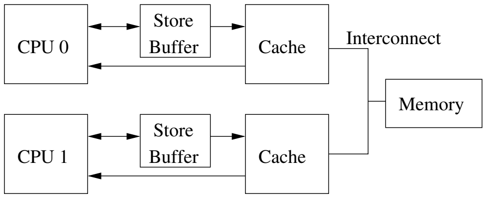


Figure 2: A simplified illustration of the relationship between the CPUs, the store buffers, and the memory hierarchy. Each CPU is connected with its own private cache. In addition, a store buffer is placed between the CPU and the cache, so that a write issued by the CPU is first stored in the store buffer and flushed out to the cache at later time. A read may be served by the cache, or by the store buffer if the store buffer contains a write to the same target address as the read.

The target architecture we consider supports out-of-order execution, but "commits" executed instructions in order. While the underlying hardware can freely reorder instructions, the result of committed instructions must still obey rules defined by the memory model implemented by the hardware. Specifically, the TSO and PO models conform to the following ordering principles for regular reads and writes issued by a given processor:[3]

1. Reads are not reordered with other reads;
2. Writes are not reordered with older reads;
3. Writes are not reordered with other writes; and
4. Reads may be reordered with older writes if they have different target locations (but they are not reordered if they have the same target location).

Principle 4 violates the Dekker duality, because it allows the read in line T1.2 of Figure 1 to appear to Thread 2 as if it has occurred before line T1.1, even though it appears as executed in order for Thread 1. The reason behind Principle 4 is to allow a typical optimization that modern architectures implement — writes performed by the processing unit are queued up in a private store buffer instead of being written out to the memory hierarchy.

A memory fence instruction, `mfence`, is used to prevent other processors from observing reordering of the executing processor's instruction stream, at the point of a `mfence` execution — `mfence` simply forces the processor to stall until its store buffer is drained, flushing all its entries out to the cache in FIFO order. We say that the executing processor "serializes" its instruction stream at the point of `mfence`, meaning that all the memory accesses of the executing processor prior to `mfence` become visible to all processors, before it can execute instructions after `mfence`.

Figure 2 is a simplified illustration of the relationship between the processors (CPUs), the store buffers, and the memory hierarchy. Though not explicitly shown in Figure 2, the memory hierarchy in modern architectures typically consists of several levels of private and shared caches and the main memory. The further away the memory hierarchy is from the processor, the higher the latency it incurs. The use of a store buffer improves the performance of the program, because writing to a store buffer avoids the latency incurred by writing out to the cache. A write in the store buffer is only visible to the executing processor but not to other processors, however. Thus, from other processors' perspective, it appears as if the read has taken place before the older write (in program order). On the contrary, assuming that a cache coherence protocol is employed, a write becomes globally visible once its written to the cache, since the coherence protocol mandates accesses to data and enforces sequentially consistent view of the accessed data among the caches of all processors. As required by the proposed hardware mechanism for location-based memory fences, we assume that the target architecture employs the MESI cache coherence protocol [15], although the mechanism can be adapted to other variants such as MSI [13] and MOESI [1].

Now we define more precisely what we mean by committing the executed instructions in order. A read instruction is considered to be ***committed*** once the data is available (in at least Shared state) in the processor's private cache. A read may be speculatively executed out of order, but it must be committed in order. That is, the processor may perform a speculative read and fetch the cache line early, but if the cache line gets invalidated between the speculative read and when the read should commit in program order, the processor must reissue the read and fetch the cache line again. Once a read is committed successfully, the read value can be used in subsequent instructions.

A write instruction, on the other hand, has two phases: "committed" and "completed." A write is considered to be ***committed*** once it is written to the store buffer, although its effect is not yet visible by other processors. A write is considered to be ***completed***, when it is flushed from the store buffer and written to the processor's cache. Once a write is completed, its effect becomes globally visible, since the cache coherence protocol ensures that all processors have a consistent view (the processor must gain Exclusive state on the flushed location before it update the value in the cache).

Since the guarantee is only that instructions must be committed in order, once the write is committed, the processor is free to continue executing subsequent instructions. A subsequent read instruction (with a different target address) may freely commit, even though an older write may still be in the store buffer. Thus, the resulting behavior observable by the other processors is that the read appears to have taken place before the older write.

The executing processor does not observe this reordering and always sees its own write, however, since the hardware employs ***store-buffer forwarding***, so that a read with a target address that appears in the store buffer is serviced by the store buffer instead of by the cache. The store-buffer forwarding also enforces the ordering principle that a read is not reordered with an older write if they have the same target address. Furthermore, due to store-buffer forwarding, when two writes from two processors, say P_1 and P_2, interleave, the write ordering observed by P_1 may differ from the write ordering observed by P_2, because each processor always sees its own write as soon as it commits, but not the write performed by the other processor until the write reaches the cache.[4]

Whenever the system bus is available, the store buffer flushes the oldest entry to memory, so that each write becomes complete in FIFO order, ensuring that a write is never reordered with other writes (Principle 3). In the event that a context switch, an interrupt, or a serializing instruction (e.g., a memory fence) is encountered, the entire store buffer is drained as well, stalling the processor until all writes in the store buffer become globally visible.

3. LOCATION-BASED MEMORY FENCES

In this section, we describe location-based memory fences, or `l-mfence` in detail, including its informal specification, usage, and a proposed hardware implementation. The formal specification, as well as a correctness proof, is presented in Section 4. The proposed hardware mechanism that implements the `l-mfence` assumes an underlying architecture as described in Section 2.

[3]There are more ordering principles when one considers the interleaving of memory accesses issued by multiple processors and when one accounts for serializing instructions and memory fences; for the purpose of explaining the hardware mechanism, we only include a relevant subset. We refer interested readers to [1, 15, 24] for full details.

[4]While P_1 and P_2 may observe different orders, the other processors in the system will observe a consistent ordering of the two writes.

Primary Thread

```
K1  l-mfence(&L1,1);
K2  if(L2 == 0) {
K3    /* critical
K4       section */
K5  }
K6  L1 = 0;
```

Secondary Thread

```
J1  L2 = 1;
J2  mfence();
J3  if(L1 == 0) {
J4    /* critical
J5       section */
J6  }
J7  L2 = 0;
```

(a)

Instruction translation for l-mfence(L1,1) (line K1 in Thread 1)

```
K1.1  MOV LEBit <- 1     //set LEBit
K1.2  MOV LEAddr <- &L1  //LEAddr gets addr of l
K1.3  LE  &L1            //load l in E mode
K1.4  ST  [&L1] <- 1     //store l=v
K1.5  BNQ LEBit, 0, DONE //Go to DONE if LEBit != 0
K1.6  MFENCE             //else execute mfence
K1.7  DONE:
K1.8  //the rest of the program (line K2)
```

(b)

Figure 3: (a) The asymmetric Dekker protocol using location-based memory fences. The code for the primary thread is shown in lines K1–K6, and the code the secondary thread is shown in lines J1–J7. **(b)** The instructions generated for `l-mfence` shown in line K1 in **(a)**.

Informal Specification and Usage of l-mfence

An `l-mfence`, unlike an ordinary memory fence, executes a memory fence "on demand." It takes in two inputs: a location l guarded by the fence and a value v to store in l (see Figure 3(a)), and it serializes the instruction stream of the executing processor only when another processor attempts to access the guarded memory location.

The serialization of P's instruction stream enforces a relative ordering between the store S associated with the execution of `l-mfence` and the other accesses performed by P. The ordering between S and an access A are observed consistently across processors, including the processor executed the `l-mfence`. That is, from P's perspective, if P executed S before (after) A, all processors observe that S "happened" before (after) A. The serialization does not enforce any relative ordering between accesses that happen before (after) S, however, meaning that the `l-mfence` ensures that all processors (including P) consistently observe that A_1 and A_2 happened before (after) S, but they may not have a consistent view of the relative ordering between A_1 and A_2. The relative ordering between these accesses is still defined by the TSO / PO memory model.

The use of `l-mfence` is very similar to the use of `mfence`, except that the `l-mfence` is associated with a specific store. A in `mfence`, when `l-mfence` is used in the program, an implicit compiler fence should be inserted in place to prevent reordering at the compiler level. Threads synchronizing via `l-mfence` need to coordinate with each other and be careful as to where to place the `l-mfence` and which memory location to guard / read after. Since a `l-mfence` does not guarantee atomic read-modify-write operation, its correct usage typically involves single writer only. Note that `l-mfence` prevents other processors from observing the reordering of the executing processor's instruction stream, but it does not prevent the executing processor from observing reordering of other processors' instruction streams. Therefore, correct usages of `l-mfence` typically consist of a pair of memory fences. For instance, to ensure correct execution in the case of the Dekker protocol, it is crucial that *both* processors insert memory fences between the write and the read, to prevent the other processor from observing reordering. For `l-mfence`, the pairing can be with either another `l-mfence` or an ordinary `mfence`.

Hardware Implementation of l-mfence

Our proposed implementation of `l-mfence` requires a new hardware mechanism, called ***load-exclusive / store***, or ***LE/ST***. Conceptually, the LE/ST mechanism allows the processor to setup a "link" to keep track of the status of the store associated with the `l-mfence` (i.e., whether the store to guarded location is committed or completed as defined in Section 2). It also allows the processor to coordinate with the cache controller to monitor attempts to access the guarded location. Another processor's attempt to access the guarded location causes the processor to clear the link and triggers actions necessary to serialize the instruction stream. On the other hand, if the store becomes complete before another processor attempts to access the guarded location, the processor clears the link and thus stops guarding the location.

LE/ST requires one new instruction and two additional hardware registers. The new instruction, `LE`, takes one operand — the location of the variable to load, and obtains Exclusive state on that location. Therefore, once `LE` is ***committed***, the processor has the location in its cache in Exclusive state, and no other processors have a valid copy of the location in their cache. Since `LE` is very similar to a regular load, except the requirement for Exclusive state on the location, it can be easily implemented by modern architectures using the MESI coherency protocol. The two additional hardware registers are `LEBit` and `LEAddr`, both readable and writable by the processor, and readable by the cache controller. The processor must update these register to enable the link and guard the memory location specified by the `l- mfence`. First we describe how the processor updates these registers to setup the link, and then we describe how the processor and the cache controller coordinate with each other to guard the memory location.

Figure 3(b) presents the assembly-like translation for the `l-mfence(l,v)` where $l == \&L1$ and $v == 1$.[5] Initially, `LEBit` and `LEAddr` are cleared. As part of the `l-mfence(&L1,1)`, the processor starts to create the link to the guarded location by setting the `LEBit` with 1 and `LEAddr` with $L1$ (lines K1.1 and K1.2 in Figure 3(b)). Next, the `LE` instruction in line K1.3 loads $L1$ into the cache in Exclusive state, so that no other processor holds a copy of `L1` in its cache. At this point we say that the link is set. The `ST` instruction in line K1.4 stores the value 1 to `L1`, committing it into the store buffer. If for any reason the link is broken, implied by the zero value in `LEBit` (line K1.5), the processor executes a `MFENCE` (line K1.6). The `MFENCE` causes the processor to serialize its execution — it flushes the store buffer, and by that it completes the store of the guarded location, making it globally observable by other processors. If the link is not broken when the `ST` in line K1.4 commits, the processor may continue without flushing the store buffer.

We now explain how the cache controller interacts with the processor to guard the location stored in `LEAddr`. Whenever both `LEBit` and `LEAddr` are set, the cache controller listens to cache coherency traffic, and notifies the processor if any request requires the controller to either (1) downgrade the cache line corresponding to the memory location stored in `LEAddr` from Exclusive state; or (2) evicts the cache line. The cache controller then waits for the processor's response before it takes any actions regarding the guarded location, since these events break the link to the guarded location.

When the processor receives the notification from the cache controller, it clears the `LEBit` and `LEAddr`, flushes the store buffer, and replies to the cache controller. At the time the processor replies the cache controller, the most up-to-date value of the guarded location

[5]The code shown is not strictly assembly. First, we are not using a particular instruction set. Second, for the sake of clarity, we choose to use the store instruction (line K1.4) instead of using the regular move instructions to specify instructions that write to memory (i.e., non registers).

is already in the cache. When the cache controller gets the processor's reply, it resumes the actions it needs to take regarding the guarded location. By clearing the `LEBit`, the processor remembers that the link to the guarded location is broken. In the event that the link is broken before `ST` (line K1.4) was committed, the code for `l-mfence` takes the branch that executes an `MFENCE`, causing the store buffer to flush (line K1.5) after the store commits.

The link remains set for as long as the primary processor still has the cache line, until the corresponding store to the guarded location is complete. When the corresponding store in the store buffer is flushed, possibly due to other internal reasons (for instance, the store is naturally flushed as the oldest entry in the buffer, the buffer is full, or a context switch occurs), upon completing the store, the processor also clears `LEBit` and `LEAddr`. The guarded location can still remain in the cache in Exclusive (or Modify) state if there is no request to evict or downgrade it.

The design of this hardware mechanism is intended to be lightweight and efficient. Since we assume only one pair of `LEBit` and `LEAddr` are allocated per processor, if a processor encounters a second `l-mfence` while the link from the first `l-mfence` is still in effect, the processor must clear the link and flush the store buffer before it can proceed with the second `l-mfence`, *unless* the second `l-mfence` has the same guarded location as the first one. Since a processor can handle two consecutive `l-mfence` instances without flushing the store buffer in between, in order to guarantee correctness, the cache controller must always cause the processor to flush the store buffer when it receives a downgrade request while the link is set. That means, in the rare case where a downgrade request arrives at the processor between the commit of `LE` (line K1.3) and `ST` (line K1.4) for the second `l-mfence`, the processor will flush the store buffer twice — the first flush is performed when the processor is notified, and the second flush is performed after the `ST` commits, via taking the branch (lines K1.5 and K1.6).

In the context of the Dekker protocol, since `LE` ensures that the primary processor has the cache line for `L1` in Exclusive state before the `ST` in line K1.4, its cache controller must receive a downgrade request from a secondary processor before the secondary processor can access `L1`. Furthermore, since the cache controller of the primary processor cannot respond to the downgrade request until the primary processor replies, the secondary processor will see the most up-to-date value of `L1`. Essentially, we piggyback on the cache coherence protocol to detect another processor's attempt to access the guarded location. We also rely on the coherency protocol to deliver the most up-to-date value to the other processor, since the store buffer is flushed before the cache controller replies to the secondary processor. It is necessary for the cache controller to notify the processor when it needs to evict the cache line, since the cache controller can no longer help guarding the memory location, if the given cache line is evicted.

4. FORMAL SPECIFICATION AND CORRECTNESS OF L-MFENCE

In this section, we formally define the specification of `l-mfence` and prove that the hardware mechanism described in Section 3 implements the specification.[6] Then, based on the specification of `l-mfence`, we prove that the asymmetric Dekker Protocol using `l-mfence` (as shown in Figure 3(a)) achieves mutual exclusion.

[6]The definitions we describe in this section in order to formally define the specification for `l-mfence` are similar to certain definitions described in [14], although we use different notations and terminology, and define only the terms we need.

Formal Specification of l-mfence

To formally define the specification of `l-mfence`, we first define the ***serialization order*** for a given memory location.

DEFINITION 1. *Given a memory location l, the **serialization order** of accesses to l performed by all processors is as follows.*

1. *A load L from location l is **serialized after** a store S of v to l if and only if L observes v.*
2. *A store S of v to location l performed by a processor P is **serialized after** a store S' of v' to l if at the time of **completion** of S, had P executed a load, the load would have observed v' from S'.*
3. *A load L from l is **serialized before** a store S of v to l if there exists a store S' to l such that L is serialized after S', and S is also serialized after S'.*

Note that the serialization order involving stores are defined by the time of completion, not commit. To complete a store of v to l, the executing processor P must gain Exclusive state on l, and thus it can be viewed as if the store was preceded by a load from l, since the value of l exists in P's cache in Exclusive state. Furthermore, since the serialization order on a location l is defined by the completion time of stores, all processors agree on a single serialization order.

Definition 1 defines the serialization order on a given memory location that is globally consistent. ***Program order***, on the other hand, is defined for a given processor, which is the order of memory accesses occurred in a processor P's instruction stream from P's perspective. If we consider all memory accesses from every processor to every memory location, there exists a global ***visibility order*** on these accesses (a posteriori), where the visibility order is consistent with the serialization order for each memory location and the ordering principles defined by the TSO / PO model relative to each processor's program order.

Given the visibility order of a particular execution, we say that a memory access A_1 ***happened before*** another access A_2 if A_1 precedes A_2 in the visibility order, or $A_1 < A_2$. From P's perspective, we say that a memory access A_1 ***occurred before*** another access A_2 if A_1 precedes A_2 in P's program order, or $A_1 \ll A_2$. Vice versa, we say that A_2 happened / occurred after A_1 if A_1 happened / occurred before A_2.

Now we define the specification of `l-mfence` formally.

DEFINITION 2. *Given a store S associated with `l-mfence` executed by a processor P, and an access A also performed by P, the `l-mfence` enforces that if $A \ll S$ then $A < S$, and vice versa, without breaking the TSO / PO ordering principles.*

An `l-mfence(l,v)` performed by processor P executes a store S of v to l, and enforces a happened-before (after) relation between S and any other access A performed by P that is consistent with S and A's relative ordering in P's program order. That is, if access A occurred before (after) `l-mfence` in P's instruction stream, A appears to all processors that it has happened before (after) S in the global visibility order.

Correctness Proof of the LE/ST Mechanism

We start by some definition and lemmas that will help us show that the LE/ST mechanism (which includes the code sequence shown in Figure 3(b)) implements the specification of `l-mfence`.

DEFINITION 3. *Given the LE/ST mechanism and a particular instance of `l-mfence(l,v)`, a link for the `l-mfence` is **set** if `LEBit` contains* 1*, `LEAddr` contain l, and the cache line for l is* *in the executing processor's private cache in Exclusive or Modified state. If any of these conditions is not met, the link is **clear**.*

LEMMA 1. *Given a particular instance of `l-mfence(l,v)`, if `LEBit` contains 1 when the store commits (line K1.4), the link must be set.*

PROOF. By committing instructions shown in lines K1.1–K1.3, the executing processor set up the link. Since `LEBit` is set as the **first** instruction of the `l-mfence` execution, if the link was broken at any point before the commit of `ST` in line K1.4, the LE/ST mechanism clears `LEBit` as part of breaking the link. Once the link is broken, `LEBit` is never set again until the next instance of `l-mfence`. □

LEMMA 2. *The LE/ST mechanism maintains the ordering principles defined by the TSO / PO memory model.*

PROOF. The LE/ST mechanism uses regular loads[7], stores, and memory fences, which maintains the FIFO ordering in the store buffer and the fact that instructions are committed in order. Thus, the TSO / PO principles are maintained. □

LEMMA 3. *The LE/ST mechanism ensures that, before P_1 commits the next instruction following `l-mfence(l,v)`, either the store S to l in line K1.4 is already complete, or any other access to l from another processor P_2 must happen after S.*

PROOF. There are two cases to consider — either the link is clear at the time when S commits (Case 1), or the link is still set (Case 2).

Case 1: By Lemma 1, we know that if the link is clear, the `LEBit` must be 0. Therefore, by the code for LE/ST mechanism (Figure 3(b)), the condition for the branch (line K1.5) is false, and thus P_1 must execute the `MFENCE` in line K1.6, causing S to complete before the next instruction (line K2 in Figure 3(a)) commits.

Case 2: If the link is set, by Definition 3, we know that P_1 has l in Exclusive / Modify state. Therefore, any processor P_2 will issue coherence traffic to P_1 before P_2 can commit a load from l or complete a store to l (a store must acquire Exclusive state on the location before it can complete). Since the link is set, P_1's cache controller will notify the processor when such request arrives. By the LE/ST mechanism upon notification, P_1 clears the link, flushes its store buffer to complete S, and replies to the cache controller. Only after that, the cache controller responds to P_2's request. Thus, P_2's access to l happened after S. □

THEOREM 4. *The LE/ST mechanism implements the specification of `l-mfence` as defined in Definition 2.*

PROOF. To show that the LE/ST mechanism implements the specification of `l-mfence`, we show that `l-mfence` enforces that if $A \ll S$ then $A < S$ and vice versa. By Lemma 2, we know that the LE/ST mechanism maintains the TSO / PO principles. Thus, the case where $A \ll S$ (for A being either a load or a store) is trivially true. Similarly, the case where $S \ll A$ where A is another store to a different location is also trivially true. Moreover, since the visibility order is always consistent with the serialization order to a given location, the case where $S \ll A$ where A loads from or stores to the same target location as S, is also trivially true. Thus, the only case we need to analyze is $S \ll A$, where A is a load with a different target location, and we show that $S < A$.

Let P_1 be the processor executing the `l-mfence(l1,v)`, and its program order dictates that the store S to l_1 (associated with the `l-mfence`) happened before a load A from location l_2. Assume for the purpose of contradiction, that some processor P_2 observes that S happened after A. The only way that P_2 can observe such happened-after relation is if P_2 performs some operations B accessing l_2 and C accessing l_1 in such way that S is forced to happen after A. That is, based on the TSO / PO principles and the serialization order observed by P_2 during execution, S cannot happen before A.

We consider possible candidates for B and C. In order to enforce an ordering on B and C in visibility order, we cannot have B being a store and C being load, because the TSO / PO principles does not enforce that $B < C$ if $B \ll C$. We will not consider B being a load, because we wish to enforce a visibility order between A (also a load) and B based on the serialization order of l_2, which is determined by stores completed on l_2. Involving an additional store on l_2 to force a serialization order between A and B is essentially the same effect as simply choosing B as a store. Thus, we only consider the case in which both B and C are store operations.

With B storing to l_2 and C storing to l_1, we construct a scenario to obtain the visibility order $A < B < C < S$. Since $B \ll C$ in P_2's program order, we have $B < C$ as dictated by the TSO / PO model (Principle 3 in Section 2). We can obtain $A < B$ via the serialization order, since they both operate on memory location l_2 (assuming B is serialized after A). Similarly, we can obtain $C < S$ via the serialization order, since they both operate on memory location l_1.

Given this visibility order, we know that A (a load) must commit before B completes, otherwise A would observe the value stored by B and therefore serialize after B. Similarly, C must complete before S completes, otherwise S would serialize before C. We also know that B must complete before C completes, by the TSO / PO principles. That means, A must commit before S completes. There are two cases to consider here.

Case 1: The link for the `l-mfence` that S is associated with is clear when A commits. By Lemma 3, since S must complete before the next instruction (following `l-mfence`) commits, we know that this visibility order cannot occur, and $S < A$.

Case 2: The link is set when A commits. In this case, S is committed but not yet complete when A commits. Let's name the next immediate access to l_1 that completes as D (possibly from any processor). By Lemma 3, D must happen after S, i.e., $S < D$. If D turns out be C, then $S < C$, and there is no reason why we cannot rearrange the visibility order to obtain $S < A < B < C$, since there is no ordering constraint that prevents S from moving upward. If D is not C, then C must complete and happen after D, so we still have $S < C$. With the same reasoning, we can rearrange the visibility order to obtain $S < A < B < C$.

In both cases, we have $S < A$, which agrees with P_1's program order, $S \ll A$. □

Theorem 4 proves that the LE/ST mechanism correctly implements the specification of `l-mfence`. In the following subsection, we prove that this specification is sufficient to guarantee mutual exclusion if it is used by the primary thread in the asymmetric Dekker protocol.

Correctness Proof of the Asymmetric Dekker Algorithm using l-mfence

We now prove that the `l-mfence(l,v)` specification is sufficient for achieving mutual exclusion when it is used in the asymmetric Dekker protocol, such as shown in Figure 3(a). The proof is based on two lemmas, each shows that if one thread is running in

[7] As explained in Section 2, the `LE` instruction is very similar to a regular load and can be implemented using the existing architecture and cache coherency protocol.

its critical section, the other one is prevented from entering it. For brevity, we name the primary thread executing the `l-mfence` $T1$ (lines K1–K6 in Figure 3(a)) and the secondary thread $T2$ (lines J1–J7 in Figure 3(a)).

LEMMA 5. *Assuming both $T1$ and $T2$ are concurrently attempting to enter the critical section. If $T1$ reads that $L2 == 0$ in line K2 and is therefore entering the critical section, $T2$ will not enter the critical section.*

PROOF. If $T1$ reads that $L2 == 0$ in line K2, we know that the load in line K2 must have committed before the store in line J1 completed. That is, the load in line K2 happened before the store in line J1. Since $T2$ uses an `mfence` (line J2) between lines J1 and J3, the load in line J3 cannot execute until the store in line J1 completes. Thus, the load in line K2 must also have happened before the load in line J3. By the specification of `l-mfence` (Definition 2), since the store to $L1$ associated with the `l-mfence` in line K1 must appear to happen before the load in line K2 to all processors, $T2$ must observe that the store in line K1 happened before the load in line K2, which happened before the load in line J3. Therefore, when $T2$ executes the load in line J3, it must observe the store performed in line K1, read $L1 == 1$ (assuming $T1$ has not left the critical section), and refrain from entering the critical section. □

LEMMA 6. *Assuming both $T1$ and $T2$ are concurrently attempting to enter the critical section. If $T2$ reads that $L1 == 0$ in line J3 and is entering the critical section, $T1$ will not enter the critical section.*

PROOF. If $T2$ reads that $L1 == 0$ in line J3, we know that the load in line J3 must have been serialized before the store that is associated with the `l-mfence(L1, 1)`. Thus, the load in line J3 happened before the the store for `l-mfence(&L1, 1)` in line K1. By the specification of `l-mfence` (Definition 2), $T2$ must observe that the store associated with line K1 happened before the load in line K2, so the load in line J3 must also have happened before the load in line K2. Since $T2$ uses an `mfence` (line J2) between lines J1 and J3, the load in line J3 cannot execute until the store in line J1 completes, and thus the store in line J1 must also have happened before the load in line K2. Thus, when $T1$ executes the load in line K2, $T1$ must observe $T2$'s store to $L2$ and read $L2 == 1$ (assuming $T2$ has not left the critical section), and refrain from entering the critical section. □

THEOREM 7. *The asymmetric Dekker protocol using `l-mfence` allows at most one thread to execute in the critical section at any given time.*

PROOF. Follows form Lemmas 5 and 6. □

As we have shown, the asymmetric Dekker protocol shown in Figure 3(a) guarantees mutual exclusion. Nonetheless, it requires additional tie-breaking code (similar to the original Dekker protocol) to avoid live lock situations in which both threads are kept outside their critical sections.

The asymmetric Dekker protocol is designed to optimize away the overhead incurred onto the primary thread at the expanse of additional overhead on the secondary thread, which is advantageous for applications that exhibit asymmetric synchronization patterns. Hence, we use an `mfence` in the secondary thread instead of `l-mfence` to avoid incurring additional overhead on the primary thread. If the secondary thread was using an `l-mfence`, the primary thread may need to wait for the secondary thread to flush its store buffer when it attempts to read `L2` in line K2. Nevertheless, the secondary thread has the option of executing the mirrored code (using `l-mfence(&L2,1)` in line J2), and the protocol still provides mutual exclusion in such case.

5. EVALUATION

The purpose of our evaluation is two folds. First, we demonstrate that performance benefit is gained using location-based memory fences instead of program-based memory fences. Second, we evaluate and analyze the expected performance of our proposed hardware mechanism, LE/ST, based on performance results of a software implementation of location-based memory fences.

We implemented a software prototype of `l-mfence` in which the threads communicate by signaling. We applied this implementation in two applications that exhibit asymmetric synchronization patterns, and examined their behavior under different benchmarks when executed serially and when executed in parallel. We ran all experiments on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache and a 512-KByte private L2-cache, and all cores on a chip share a 2-MByte L3-cache.

When executed serially, the benchmarks perform better using the software implementation of `l-mfence` than their counterparts using `mfence`. The reason for these results is that our software prototype incur effectively no overhead on the executing thread when it runs serially. When executed in parallel, even though the communication overhead of the software prototype is high, some benchmarks still gain performance benefit from using the software implementation of `l-mfence`. While the software implementation is feasible, we argue that the LE/ST mechanism would significantly enhance the performance of the benchmarks in parallel executions (without changing the results in the serial executions), and enable a larger class of programs to benefit from `l-mfence`.

In this section we briefly summarize the software prototype, compare the overhead between the software prototype and the LE/ST mechanism, describe the experimental results based on the software prototype, and discuss how the outcomes would differ with the LE/ST mechanism.

Software Prototype of l-mfence

We implemented the location-based memory fence using signals, similar to the approach proposed in [6]. The software prototype must correctly capture two main effects. First, the primary thread must not reorder the write and the read at the compiler level. We achieve this simply by inserting a compiler fence at the appropriate place. Second, before the secondary thread attempts to read the variable written by the primary thread, it must cause the primary thread to serialize, and only proceed with the read *after* the primary thread has performed the serialization. We achieve this via signals — a software signal generates an interrupt on the processor receiving the signal, and the processor flushes its store buffer before calling the signal handling routine. Thus, the secondary thread sends a signal to the primary thread and waits for an acknowledgment by spinning on a shared variable. Upon receiving the signal (which implicitly flushes the store buffer), the primary thread executes a user-defined signal handler, which sets the shared variable as an acknowledgment, thereby allowing the secondary thread to resume execution.

Overhead Comparisons between the Software Prototype and the LE/ST Mechanism

We compare the overhead between the software prototype and the LE/ST mechanism in two cases: when the primary thread executes alone, and when other secondary threads exist in the context.

When the primary thread executes alone, the software prototype incurs negligible overhead from the compiler fence, while the LE/ST mechanism would incur small additional overhead from setting the link, performing the load-exclusive, and taking the branch.

Nevertheless, this additional overhead should be negligible as well, since the target cache line of the load stays in the primary processor's cache, and the branch can be predicted successfully.

During parallel execution, the software implementation using signals would incur much higher communication overhead compared to the LE/ST mechanism. In the software implementation, the communication overhead includes the secondary thread sending the signal and waiting for the primary thread to flush its store buffer and handle the signal. Furthermore, this software implementation also slows down the primary thread whenever communication occurs, because the primary thread must handle the signal (which entails crossing between kernel and user modes four times to execute a user-defined signal[8]) while the secondary thread waits. The estimated cost of a single round trip communication is in the order of $10,000$ cycles on the system in which we ran the experiments. On the other hand, the round trip time in the LE/ST mechanism involves waiting for the cache controllers of the two processors to send and handle messages (akin to a L1 cache miss / L2 cache hit), and for the primary processor to flush its store buffer. We ran a synthetic benchmark to simulate this round trip time, which costs about 150 cycles on our system. Moreover, the performance impact on the primary processor is negligible: it just needs to flush the store buffer and regain the cache line the next time around; the processor performance is not affected by the cache controller listening to cache traffic and handling messages.

As stated in Section 1, performance benefit can be gained using `l-mfence` if the latency avoided by the primary thread is greater than the communication overhead born by the secondary thread. Putting the overhead comparison into the context of benchmarks execution, the software implementation requires significantly more asymmetry in the benchmarks in order to gain performance than the LE/ST mechanism.

Applications Overview

We incorporate a software implementation of `l-mfence` into two applications — the asymmetric Cilk-5 runtime system and an asymmetric multiple-readers single-writer lock.

For the first application, we have modified the open-source Cilk-5 runtime system [11][9] to incorporate `l-mfence` into the Dekker-like protocol employed by its work stealing scheduler, referred as the ***ACilk-5 runtime system***. In a work-stealing scheduler, when a thief (the secondary thread) needs to find more work to do, it engages in an augmented Dekker-like protocol with a given victim (the primary thread) in order to "steal" work from the victim's "deque." Assuming the benchmarks contains ample parallelism, a victim would access its own deque much more frequently than a thief, because steals occur infrequently.

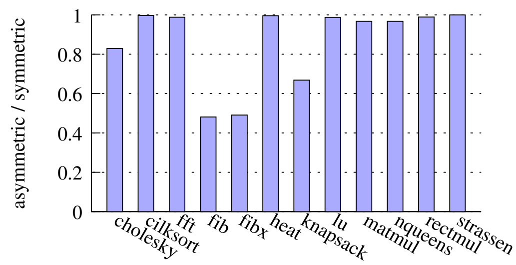
For the second application, we have designed an ***asymmetric multiple-readers single-writer lock***, where the lock is biased towards the readers, henceforth referred as the ***ARW lock***. From time to time, a reader (the primary thread) turns into a writer (the secondary thread), and attempts to acquire the ARW lock in the write mode by engaging in an augmented Dekker protocol with each of the registered readers.

Evaluation Using ACilk-5

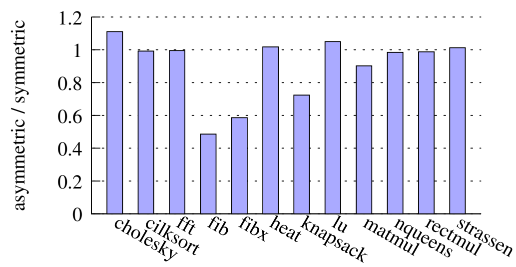
To evaluate the effect of location-based memory fences, we compare the execution time of 12 benchmarks running on ACilk-5 versus running on Cilk-5. Figure 4 provides a brief description of each benchmark.

Benchmark	*Input*	*Description*
`cholesky`	4000/40000	Cholesky factorization
`cilksort`	10^8	Parallel merge sort
`fft`	2^{26}	Fast Fourier transform
`fib`	42	Recursive Fibonacci
`fibx`	280	Alternate between fib(n-1) and fib(n-40)
`heat`	2048×500	Jacobi heat diffusion
`knapsack`	32	Recursive knapsack
`lu`	4096	LU-decomposition
`matmul`	2048	Matrix multiply
`nqueens`	14	Count ways to place *N* queens
`rectmul`	4096	Rectangular matrix multiply
`strassen`	4096	Strassen matrix multiply

Figure 4: The 12 benchmark applications.



relative execution time on single processor



relative execution time on 16 processors

Figure 5: (a) The relative serial execution time of the ACilk-5 runtime system compared to the original Cilk-5 runtime system for 12 Cilk benchmarks. **(b)** The relative execution time of the ACilk-5 runtime system compared to the original Cilk-5 runtime system for 12 Cilk benchmarks on 16 cores. A value below 1 means that the application runs faster on ACilk-5 than on Cilk-5; a value above 1 means the other way around. Each value is calculated by normalizing the execution time of the benchmarks on ACilk-5 with that on Cilk-5.

Figure 5(a) compares the performance of the benchmarks run on ACilk-5 and Cilk-5 when executed serially. Figure 5(b) shows a similar performance comparison when executed on 16 cores. For each measurement, we took the mean of 10 runs (with standard deviation of less than 3%). A value below 1 means that the benchmark runs faster on ACilk-5 than on Cilk-5.

Not surprisingly, when executed serially, benchmarks on ACilk-5 run faster, because the victim executes on the fast path with virtually no overhead from memory fences. The improvement that ACilk-5 exhibits over Cilk-5 when running a given benchmark is directly related to the ratio between the overall work in a given benchmark and the number of fences avoided in the benchmark (which corresponds to the the number and the granularity of parallel tasks that the benchmark generates). The fewer the number of memory accesses is performed under a given fence, the more saving one obtains from avoiding the fence. All these benchmarks except for `fib`, `fibx`, and `knapsack` have their base case coarsened (so as to generate enough parallel tasks and avoid parallel overhead

[8] One could modify the operating system to cut the signal handling overhead down by half (crossing between kernel and user modes two times instead of four), but that would still be in the order of thousands of cycles.

[9] The open-source Cilk-5 system is available at `http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz`.

when there is enough parallelism), so the ratio of work per fence is high. On the other hand, `fib` is specifically designed to measure the "spawn" (for generating parallel tasks) overhead, and the number suggests that the spawn overhead is cut by half if one could avoid the fence. We believe the numbers will be comparable if `l-mfence` were implemented using the LE/ST mechanism.

Figure 5(b) shows the same performance comparison when executed on 16 cores. When executed in parallel, the software implementation of `l-mfence` incurs an additional communication overhead for every steal attempt (which impacts both the victim and the thief). Despite the communication overhead, many benchmarks still exhibit saving or stay even (meaning that saving and overhead even out). The three exceptions are `cholesky`, `heat`, and `lu`. There are two factors at play here. First, while the analysis of the work-stealing algorithm dictates that one should put the scheduling overhead onto the steal (thief's) path instead of onto the work (victim's) path (referred as the "work-first" principle [11]), one must be able to amortize the overhead against successful steals in order to obtain good scalability. In the case of `cholesky` and `lu`, much of the communication overhead did not translate into successful steals — only 53.6% of signals sent in `cholesky` turns into successful steals, and only 72.8% for `lu` (while other benchmarks have over 90%). As a result, the benchmarks do not scale as well. Second, while over 90% of the signals sent in `heat` translates to successful steals, the number of fences avoided per signals sent is much smaller compared to other benchmarks, so the communication overhead incurred by `l-mfence` outweighs the benefit. Given that the LE/ST mechanism has much smaller communication overhead and impacts only the thief, we believe both problems would be avoided.

Evaluation Using ARW Lock

We evaluate the effect of location-based memory fences by comparing the read throughput of a microbenchmark using the ARW lock to the read throughput using its symmetric counterpart: the same design but uses the `mfence` for the primary thread in the Dekker protocol instead of using the `l-mfence`, henceforth referred as the ***SRW lock***. Each thread performs read operations most of the time, and only occasionally it performs a write. In the tests, the threads read from and write to an array with 4 elements. The read-to-write ratio is an input parameter to the microbenchmark: assuming the ratio is $N:1$, and there are P threads executing, then for every N/P reads, a thread performs a write. With each configuration, we run the microbenchmark for 10 seconds, calculate the read throughput, and compare the throughput using the ARW lock against the throughput using the SRW lock.

Figure 6(a) shows the throughput comparison between the ARW lock and the SRW lock. In the software implementation of `l-mfence`, since a request for serialization translates to a signal, the writer ends up signaling a list of readers and waiting for their responses one by one, which becomes a serializing bottleneck. This is particularly inefficient when the thread counts is high, and the read-to-write ratio is low (less asynchronous), since the communication overhead outweighs the benefit from avoiding fences.

We believe that the lack of scalability is again due to the high communication overhead in the software implementation. To confirm this, we devised an ARW lock that implements a *waiting heuristic*: when a writer wants to write, instead of sending signals to the readers immediately, it first indicates intent to write and spin-waits to see if any reader responds, acknowledging the writer's intent to write. Only after spin-waiting for awhile, the writer sends signals to readers who have not acknowledged. We refer to the ARW lock with this heuristic as the ***ARW+ lock***.

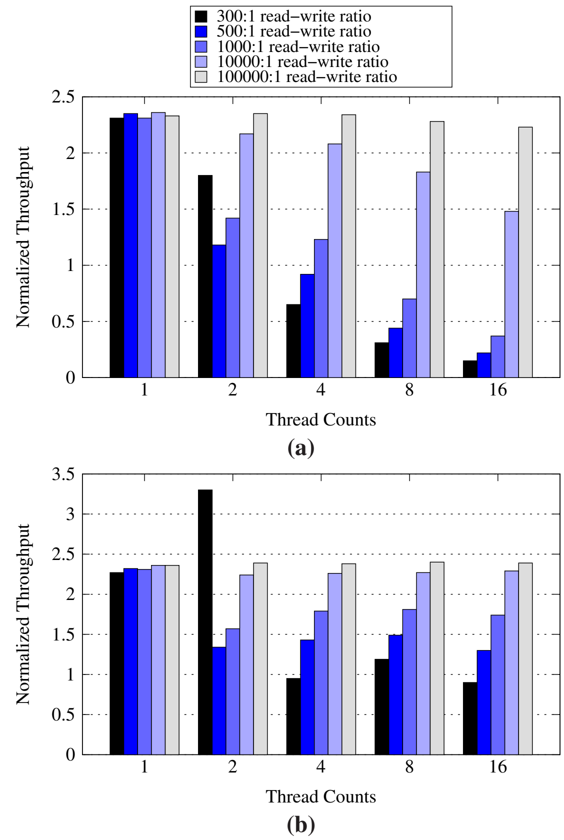


Figure 6: **(a)** The relative read throughput of execution using the ARW lock compared to that using the SRW lock. **(b)** The relative throughput of execution using the ARW+ lock (i.e., the ARW lock with the waiting heuristics) compared to that using the SRW lock. Since we are comparing the relative throughput, a value above 1 means that the ARW lock / ARW+ lock performs better; a value below 1 means that the SRW lock performs better. Each value is calculated by normalizing the read throughput from the execution using the ARW lock by that using the SRW lock.

Figure 6(b) shows the throughput comparison between the ARW+ lock and the SRW lock. A value above 1 means that the ARW+ lock performs better. There are two main trends to notice. Indeed, the ARW+ lock scales much better and consistently has higher throughput compared to the SRW lock, except for the $300:1$ read / write ratio (which is close to 1). One notable out-lier in Figure 6(b) is the data point for $300:1$ ratio with two threads, which has much higher throughput compared to other thread counts. This can be explained by the fact that when there are only two threads, the writer end up receiving the acknowledgment most of the times and does not need to send signals.

While the waiting heuristic seems to work well in the microbenchmarks, if the reader does not access the lock frequently, the heuristic would not help as much, because a thread would only check for pending intent during lock acquire and release. With that in mind, the results inspire confidence that the ARW lock should perform and scale well when the `l-mfence` is implemented with the LE/ST mechanism.

6. RELATED WORK

Our work is closely related to studies performed on biased locks and asymmetric synchronization, so we focus on these in the section. Several researchers studied this area, mainly in the context of improving performance for Java locks.

[23] describes a fast biased lock algorithm, which allows the primary thread to avoid executing memory fences, until a secondary thread attempts to enter the critical section. In which case, the secondary thread must wait for the primary thread to grant access in order to continue execution. While this request and grant protocol is performed via shared variables and therefore fairly efficient, this implementation can potentially deadlock if the biased lock is nested within another lock (or any resource that can block).

The studies in [7] and [21] describe similar biased lock implementations, where the owner of the lock is on the fast path for accessing the lock, and other threads need to revoke it and compete for ownership, and the lock ownership may transfer. Both algorithms use the "collocation" trick, where the status field and the lock field are allocated on the same word. They first write to one field and then the whole word is read. The correctness of the algorithm depends on the fact that hardware typically does not reorder read before older write when the addresses overlap. This collocation trick, while interesting, is not guaranteed to be safe, and on systems which this trick works correctly, it always forces a memory fence to be issued regardless of whether there is contention [8].

Serialization using signal and notify was proposed in [6], as well as other more heavy-weight serialization mechanisms. Their work focus on software means to cause serialization in another thread, while decreasing synchronization overhead on the primary thread in applications that exhibit asymmetric synchronization patterns.

Finally, in [20], Lin et al. propose a hardware mechanism for conditional memory fences, whose aim is also to reduce the overhead of memory fences when synchronization is unnecessary. In [20], however, the assumption is that the compiler would automatically insert memory fences in order to enforce sequential consistency everywhere, and there may be multiple outstanding memory fences for a given thread at a given moment. Thus, their hardware mechanism is much more heavyweight compared to ours, so as to handle multiple outstanding fences at a given moment. Our mechanism, on the other hand, requires the programmer to manually insert fences at the appropriate program points where the relative orderings of memory accesses matter; when a second `l-mfence` is encountered while the first one (with a different guarded location) is still in effect, our mechanism also causes the store buffer to drain before handling the second one. As a result, our mechanism is much more lightweight.

7. CONCLUSION

In this work, we propose location-based memory fences, which aim to reduce the overhead incurred by memory fences in parallel algorithms. Location-based memory fences are particularly well-suited for algorithms that exhibit asymmetric synchronization patterns. We describe a hardware mechanism to support location-based memory fences, proved its correctness and evaluate the feasibility of the fences using a software prototype. Our evaluation with the software prototype inspires confidence that the suggested LE/ST mechanism for supporting location-based memory fences in hardware is worth considering.

Finally, location-based memory fences lend itself to a different way of viewing programs compared to the traditional program-based memory fences. It would be interesting to investigate what other algorithms can benefit from location-based memory fences, as well as other mechanisms that exploit the location-based model.

8. ACKNOWLEDGMENTS

We like to thank Joel Emer of Intel Corporation and MIT, David Dice of Oracle Labs, and William Hasenplaugh, Charles Leiserson, Jim Sukha, and other members of the SuperTech Group at MIT CSAIL for helpful discussions.

9. REFERENCES

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2010.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*, pages 119–129, June 1998.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '05*, pages 207–216, July 1995.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, September 1999.

[5] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[6] Dave Dice, Hui Huang, and Mingyao Yang. Asymmetric dekker synchronization. Technical report, Sun Microsystems Inc., July 2001.

[7] Dave Dice, Mark Moir, and William Scherer III. Quickly reacquirable locks. Technical report, Sun Microsystems Inc., 2003.

[8] David Dice. David Dice's Weblog: `http://blogs.sun.com/dave/entry/biased_locking_in_hotspot#comments`, 2006.

[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[10] E. W. Dijkstra. Co-operating sequential processes. In *Programming Languages*. 1968.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[12] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, fourth edition, 2007.

[14] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, October 2011.

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, January 2011.

[16] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA '02*, pages 130–141, 2002.

[17] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI '89*, pages 81–90, June 1989.

[18] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[20] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient sequential consistency using conditional fences. In *PACT '10*, pages 295–306, 2010.

[21] Tamiya Onodera, Kiyokuni Kawachiya, and Akira Koseki. Lock reservation for java reconsidered. In *ECOOP '04*, pages 559–583, 2004.

[22] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[23] Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *PACT '10*, pages 65–74, 2010.

[24] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, 1994.