End-To-End Sequential Consistency

Abhayendra Singh[*] Satish Narayanasamy[*] Daniel Marino[†] Todd Millstein[∓] Madanlal Musuvathi[‡]

[*]University of Michigan, Ann Arbor [†]Symantec [∓]University of California, Los Angeles [‡]Microsoft Research, Redmond

Abstract

Sequential consistency (SC) is arguably the most intuitive behavior for a shared-memory multithreaded program. It is widely accepted that language-level SC could significantly improve programmability of a multiprocessor system. However, efficiently supporting end-to-end SC remains a challenge as it requires that both compiler and hardware optimizations preserve SC semantics. While a recent study has shown that a compiler can preserve SC semantics for a small performance cost, an efficient and complexity-effective SC hardware remains elusive. Past hardware solutions relied on aggressive speculation techniques, which has not yet been realized in a practical implementation.

This paper exploits the observation that hardware need not enforce any memory model constraints on accesses to thread-local and shared read-only locations. A processor can easily determine a large fraction of these safe accesses with assistance from static compiler analysis and the hardware memory management unit. We discuss a low-complexity hardware design that exploits this information to reduce the overhead in ensuring SC. Our design employs an additional unordered store buffer for fast-tracking thread-local stores and allowing later memory accesses to proceed without a memory ordering related stall.

Our experimental study shows that the cost of guaranteeing end-to-end SC is only 6.2% on average when compared to a system with TSO hardware executing a stock compiler's output.

1. Introduction

A memory consistency model (or simply *memory model*) of a concurrent programming language specifies the order in which memory accesses performed by one thread become visible to other threads in the program. It is a language-level contract that programmers can assume and the system (compiler and hardware) must honor. Designing a memory model involves a balance between two, often conflicting goals: improving programmer productivity with a strong memory model that matches programmer intuition, and maximizing system performance with a weak memory model that enables hardware and compiler optimizations.

Sequential consistency (SC) [36] is one of the strongest memory models discussed in the literature. SC guarantees that the memory accesses of a program appear to have executed in a global sequential order consistent with the per-thread program order. This guarantee matches the natural expectation of a programmer that a program behaves as an interleaving of the memory accesses from its constituent threads.

Researchers widely agree that providing SC could significantly simplify the concurrency semantics of languages, but they also believe that it is an unaffordable luxury [28]. Accordingly, modern languages such as C++ and Java provide SC only for data-race-free programs [4, 10, 41]. For racy programs, on the other hand, the languages either provide no semantics or a weak semantics that is difficult for a programmer to understand.

This paper seeks to provide SC for all programs while achieving performance close to that of today's compiler and hardware implementations. In a recent study [43], we showed that the cost of preserving SC during compiler optimizations is quite acceptable — less than 3.8% performance overhead on average. To provide end-to-end SC, however, the output of an SC-preserving compiler must be executed on SC hardware. Thus, the remaining need in realizing end-to-end SC is an efficient *and* complexity-effective SC hardware design.

Past work has produced efficient SC hardware designs [6, 9, 13, 22, 23, 30, 49, 54] by introducing novel techniques for speculatively reordering memory accesses and recovering when there is a possible SC violation. In-window speculation [22] is relatively simple as it only reorders memory instructions before they are are committed from the reorder buffer (ROB). Commercial processors already implement this optimization to efficiently support x86's Total Store Order (TSO) consistency model [32]. However, in-window speculation alone is insufficient to attain high performance in SC hardware, as loads still cannot be committed until the store buffer is drained. To reap the benefits of a store buffer in SC hardware, researchers have proposed a more aggressive out-of-window speculation technique that reorders even committed memory instructions [9, 13, 23, 49, 54]. But out-of-window speculation and the accompanying recovery mechanisms are arguably quite complex and have not yet been realized in any practical processor implementation.

In this paper, we propose an SC hardware design that is more complexity-effective than past out-of-window speculation techniques, but still results in an efficient design. We leverage the simple observation that memory model constraints need not be enforced for private locations and shared read-only locations [3, 44, 51]. Since most memory accesses are to private or read-only data [15, 27], this observation provides an opportunity to design an efficient SC hardware by simply relaxing the ordering constraints on many memory accesses, obviating the need for complex speculation techniques.

We propose simple extensions to a modern TSO processor design (which already supports in-window speculation [22]) that exploit the above idea to support SC efficiently. We divide the store buffer into two structures: one is the regular

FIFO store buffer that orders stores to shared locations, and the other is a private, unordered store buffer to fast-track stores to private locations. Our design allows private and shared, read-only loads to commit from the ROB without a store buffer drain. It also allows a load to a shared read-write location to commit from the ROB without waiting for the private store buffer to drain. Therefore, when compared to the TSO design implemented in today's processors, the only additional memory ordering restriction that our SC design imposes is that loads to shared read-write locations are stalled until the FIFO store buffer containing shared stores is drained.

We discuss two complementary techniques to enable a processor to identify private and shared read-only accesses. The first technique is based on static compiler analysis. We implemented an SC-preserving version [43] of the LLVM compiler that conservatively identifies all memory accesses to function locals whose references do not escape their functions. These memory accesses are guaranteed to be private to a thread. The compiler communicates this information to the processor by setting a bit in a memory instruction's machine code.

The compiler analysis necessarily needs to be conservative in classifying a memory access as private. We employ a complementary dynamic technique that extends the hardware memory management unit and operating system's page tables to keep track of private and shared, read-only pages. During address translation, a processor determines the type of a memory access and decides whether or not to enforce memory model constraints for that access. Past work employed a similar dynamic technique to track private pages, but used it to optimize cache performance [27, 35] and directory-based coherence [15] rather than to reduce the overhead due to memory model constraints.

Our experimental study on the PARSEC [8], SPLASH [55] and Apache benchmarks shows that the overhead of our SC hardware over TSO is less than 2.0% on average. We also find that the overhead of providing end-to-end SC (running the SC-preserving compiler's output on our SC hardware) when compared to running the stock LLVM compiler's output on a TSO hardware is 6.2% on average. The overhead due to the SC-preserving compiler could be further reduced using hardware-assisted interference checks [43] which we did not use in our design.

Although we focus on designing an efficient SC hardware in this paper, the observation that memory model constraints need not be enforced for private and shared, read-only accesses could be similarly exploited to improve the performance of any memory model implementation.

2. Background

Our goal is to provide language-level SC for all programs. This section motivates the need for SC and the challenges in ensuring end-to-end SC using an SC-preserving compiler and SC hardware.

2.1. Why SC for all programs?

The benefits of an easy-to-understand memory model are well known [4]. Current languages provide intuitive behavior only for data-race-free programs. While we certainly would like programmers to write data-race-free programs, the unfortunate reality is that most programs contain data-races. Some of them are even intentional [46]. Without having clear guarantees for all programs, a programmer must assume the worst (complicated, unintuitive semantics, or potentially arbitrary behavior) while reasoning about a program's execution. We believe this situation significantly compromises the programmability of today's multiprocessor systems.

2.2. Compilers Can Preserve SC

One potential argument for relaxed hardware memory models (weaker than SC) is that commonly used compiler optimizations already violate SC, so the hardware makes the problem no worse for programmers than it already is. For instance, optimizations such as common subexpression elimination (CSE), loop-invariant code motion (LICM), and dead-store elimination can all have the effect of reordering accesses to shared memory, thereby potentially violating SC even if the resulting binary is executed on SC hardware. Indeed, it is precisely to support aggressive compiler (and hardware) optimizations that today's mainstream programming languages like Java [41] and C++ [10] employ relaxed memory models based on the *data-race-free* (DRF0) model, which only guarantee SC for data-race-free programs [4].

However, in a recent study [43], we showed that an optimizing compiler can be modified to be *SC-preserving* — ensuring that every SC behavior of the generated binary is an SC behavior of the source program — while retaining most of the performance of the generated code. The empirical observation was that, a large class of optimizations crucial for performance are either already SC-preserving or can be modified to preserve SC while retaining much of their effectiveness by restricting the optimizations to thread-local variables. The study demonstrated how to convert LLVM [37], a state of the art C/C++ compiler, into an SC-preserving compiler by modifying each of LLVM's optimization passes to conservatively disallow transformations that might violate SC. The modified compiler was shown to generate binaries that were only slightly slower than a traditional, SC-violating compiler. Executing binaries produced by this SC-preserving compiler on SC hardware would guarantee end-to-end SC semantics to programmers for all programs, race-free or otherwise.

2.3. Efficient *and* Complexity-Effective SC Hardware Remains a Challenge

Before we discuss the challenges of designing SC hardware, we clarify a few commonly used terms which we also use in this paper. In a modern out-of-order processor, instructions can *execute* out-of-order but must *commit* from the reorder buffer in program order. If allowed by the memory model, a store may commit from the reorder buffer and be placed in a store buffer before its value has been written to cache or memory. The stored value is made *visible* to other threads only when a store *retires* from the store buffer, which is when its value is written to the appropriate memory location in the cache. Two memory accesses in different threads are said to *conflict* if they access the same memory location and at least one of them is a write.

SC hardware needs to guarantee that the memory accesses of a program appear to have executed in a global sequential order that is consistent with the per-thread program order. A naive SC hardware design would force loads and stores to be executed and committed in the program order. Also, a store's value needs to be made visible to all threads atomically when it is committed. This naive design disallows most hardware optimizations such as out-of-order execution and store buffers.

Even x86 processors' TSO memory model disallows loads from executing out-of-order. Fortunately, modern x86 processor implementations support a speculative optimization called in-window speculation [22] to reduce the overhead due to this load-load memory ordering constraint of TSO [32]. Loads are allowed to be speculatively executed out-of-order. The processor still commits them in-order and recovers when a possible memory ordering violation is detected between the execution and commit of a load. A violation is detected when a processor core receives a cache coherence invalidation request for a location accessed by a load that has already executed but has not yet committed. The logic that supports recovery from branch misprediction is mostly sufficient to recover from in-window memory ordering violations as well.

The primary performance overhead in TSO, when compared to weaker relaxed consistency models [4], is the cost of enforcing store-store ordering. TSO requires a global total order for all stores, which is guaranteed by committing stores to a FIFO store buffer and retiring them to memory atomically in the program order. As a result, a processor core may have to stall commit of a store from ROB if the store buffer is full. However, this overhead tends to be small for most programs.

In-window speculation is also useful for optimizing SC hardware since it allows many loads to execute out of order, eliminating much of the overhead in ensuring SC. However, unlike TSO which permits loads to be reordered before stores, SC can not take full advantage of store-buffer optimization. While SC hardware can commit a store from the ROB and place it in the store buffer, any following load cannot be committed from the ROB until the store buffer is drained. That is, all preceding stores need to be retired and their values made visible to other threads before a later load can commit. In-window speculation does not help reduce this important overhead in an SC hardware design.

Past research has proposed aggressive speculation techniques to allow store-buffer optimization in SC hardware [9, 13, 23–25, 49, 54]. These designs extend the idea of in-window speculation to speculatively commit loads from the ROB even when the store buffer is not empty. This requires fairly complex hardware that keeps track of the register and memory state before each committed load, detects potential SC violations by comparing incoming coherence invalidation requests with the addresses of committed loads, and performs a rollback when a potential SC violation is detected. To avoid speculation, Lin et al. [39] proposed to check if there is any conflict with pending accesses in remote cores before committing a memory instruction from the ROB. While this design eliminates the need for out-of-window checkpoint and rollback support, it still requires significant changes to the coherence protocol to efficiently perform conflict detection before committing a memory instruction from the ROB.

In this paper we propose an alternative mechanism to reap the benefits store-buffer optimization for a certain class of memory accesses while preserving SC.

3. Relaxing Memory Model Constraints for Safe Accesses

Processors enforce memory ordering constraints in order to prevent other processors from being able to observe reordering not allowed by the memory model. Past SC hardware designs have uniformly enforced memory model constraints on all memory accesses, distinguishing only between stores and loads. This is overly conservative and unnecessary for a significant fraction of memory accesses.

If either the compiler or the runtime system can guarantee that there can be no conflicting memory access on another thread which could observe or alter the result of a particular memory access, then the processor can safely reorder that access in any manner that preserves intra-thread data dependencies. We refer to memory accesses with this property as *safe* accesses and the rest as *unsafe* accesses.

For instance, if a memory access is to a location that is *private* to the current thread, then clearly there can be no conflicting memory accesses, so the access is safe. A compiler can guarantee this property for all dynamic instances of a static memory instruction that accesses only thread-local data. A runtime system can guarantee this property for any access to a location that it knows has only been accessed by the current thread so far during execution. Once a memory location is accessed by a second thread, the runtime system must detect this situation and require that this and future accesses obey memory model constraints on the processor. A similar idea can be used to identify *shared read-only* memory locations accessed by multiple threads as safe.

We exploit the above observation to design an efficient and complexity-effective SC hardware. Our SC hardware design can be understood in relation to out-of-window speculation techniques proposed in the past for reducing the overhead of enforcing memory model constraints [9, 13, 23, 25, 49, 54]. The key insight of those past techniques was to speculatively relax memory ordering restrictions on memory accesses as they are rarely violated. Unfortunately, the required support for recovery is costly in terms of processor complexity. In contrast, we propose to relax memory ordering restrictions only for those memory accesses which are guaranteed to be safe by the compiler or runtime system. Since our relaxation is always correct, we no longer need hardware support for misspeculation recovery, which results in a low-complexity solution.

Over 81% of memory accesses are found to be safe for our benchmark programs (Section 7.2). We focus on relaxing SC memory ordering restrictions for these accesses. But our approach is generally applicable to any memory model. For example, TSO requires that stores be retired in program order from the store buffer, but that restriction need not be enforced for safe stores.

4. Design: Memory Access Type Driven SC Hardware

This section discusses our low-complexity, efficient, SC hardware design based on exploiting memory access type information. Figure 1 shows the extensions we propose to a baseline TSO processor and operating system used today. Before we discuss our SC hardware design, we briefly describe the two techniques we use to determine safe memory accesses and how that information is communicated to the hardware.

To simplify the discussion, we assume that a memory instruction accesses only one location in memory. Section 5.3 discusses how memory instructions in a CISC architecture that can read or write to multiple locations are handled.

4.1. Two Techniques to Determine Memory Access Type

The proposed processor design relies on two complementary techniques to determine safe accesses: a static compiler analysis and a dynamic analysis based on the page protection mechanism.

The static analysis determines safe memory instructions in a program that are guaranteed to access private or read-only locations (safe locations). It does this by a conservative inter-procedural analysis to identify function-local variables that do not escape the scope of their functions (safe variables). Dynamically the memory locations of such variables will be private to the thread that invokes the function, so all accesses to these variables are considered safe. Care must be taken to ensure correctness as two function-local variables in different functions may be allocated to the same stack location (Section 5). Our analysis also considers accesses to constant literals as safe. The Instruction Set Architecture (ISA) is extended to allow a compiler to flag safe memory instructions. When a processor core decodes a memory instruction and allocates an ROB entry, it sets a bit (`ss`) in the ROB (Figure 1) if that instruction is flagged as safe by the compiler, which is later used to relax memory model constraints. This static approach incurs little runtime complexity, but it has to be conservative and may classify accesses to locations (especially those on the heap) that are actually private as unsafe.

We also employ a dynamic technique that leverages operating system (OS) support for classifying accesses at the page granularity [27]. The OS protects pages at the process-level, which we extend to support thread-level page protection by adding a few fields to the page table entry (Figure 1). The first read and/or the first write from a thread will trigger an exception to the OS, which allows the OS to keep track of the state of the page (private, shared read-only, or shared read-write). The TLB entry for a page is also extended with an additional safe bit, which is used to determine if it is a safe page or not. During address translation for a memory access in the execution stage, a processor core determines if the access is to a safe page, and sets the `ds` bit in the ROB, which is later used to relax memory model constraints. Care must be taken to preserve memory ordering constraints between memory accesses when the state of the page changes (Section 6).

Even if a page contains only one shared read-write byte, accesses to any part of the page will be treated as unsafe by the dynamic scheme described above. Thus, a static analysis that classifies locations at finer granularity complements our dynamic analysis. In the proposed design, we use a *hybrid* scheme. Since both static and dynamic classification schemes are conservative, it is correct for the hybrid scheme to consider a memory access to be safe if either one of the two methods classifies that access as safe (i.e. either `ss` or `ds` is set in the ROB entry).

4.2. SC Architecture Design

As we pointed out in Section 2.3, TSO allows loads to be reordered before stores, which enables store buffer optimization. SC, however, disallows this optimization, which is the only performance cost of SC hardware when compared to TSO hardware (assuming in-window speculation [22] for both designs).

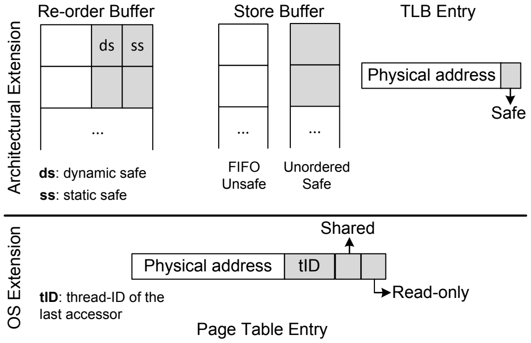


Figure 1: Memory Access Type Driven SC Processor and OS

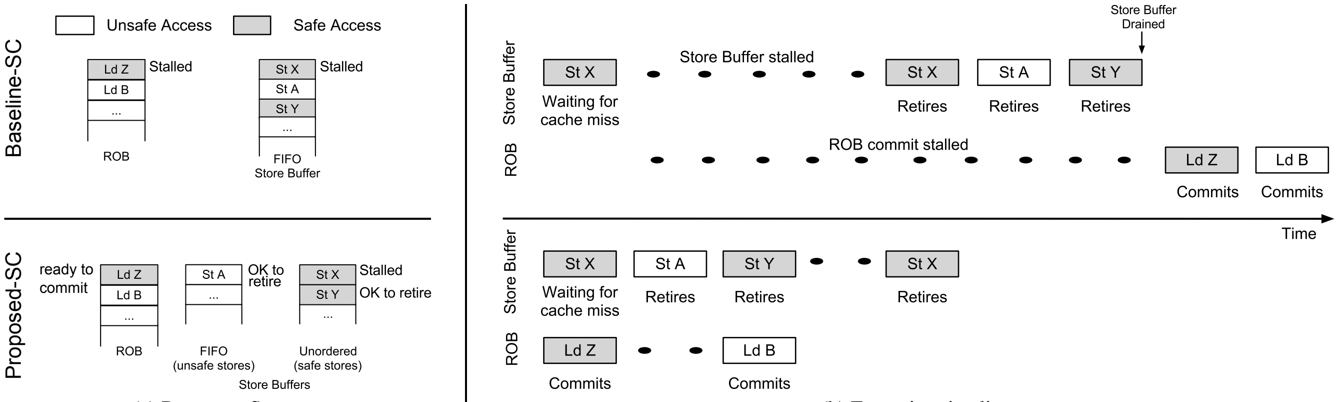
We propose a simple extension to reduce this cost significantly: divide the store buffer into two parts as shown in Figure 1. One part is the traditional FIFO store buffer for handling unsafe stores. The second is an unordered store buffer for fast-tracking safe stores. A processor core can determine whether a load/store is safe or not by examining the `ss` and `ds` bits in its ROB entry. This design has the following three main performance advantages when compared to the baseline SC design.

1. A safe load can commit from the ROB even when there are pending stores in either or both of the two store buffers (perhaps waiting for their cache misses to be serviced). Thus, we provide TSO performance for safe loads.
2. An unsafe load can commit from the ROB even when there are pending stores in the unordered store buffer containing safe stores. Thus, if a safe store is waiting for a cache miss, following unsafe loads need not wait to commit.
3. Stores in the unordered store buffer can be coalesced if they access the same cache line. Also, they can be retired out of order. As a result, a safe store need not wait for a pending (safe or unsafe) store to retire. This decreases pressure on store buffer capacity. This property could also be exploited to improve a TSO hardware's performance.

4.3. Store-to-Load Forwarding with Two Store Buffers

Having two store buffers could potentially complicate store-to-load forwarding logic. We avoid this complication by ensuring that all bytes accessed by a memory instruction are of the same type (safe or unsafe). We refer to this as the *memory-type* guarantee. Furthermore, we ensure that for any valid read-after-write dependency the two memory accesses are of the same type. Therefore, to detect store-to-load forwarding for a safe load, only the unordered store buffer needs to be searched. Similarly, an unsafe load needs to search FIFO store buffer only.

Our static analysis ensures that all the variables accessed by a memory instruction are of the same type as follows. If any memory instruction could access both safe and unsafe variables, then our analysis conservatively marks that instruction as unsafe. In addition, any safe variable accessed by that instruction is reclassified as unsafe, as are all other instructions



(a) Processor States

(b) Execution timeline

Figure 2: Comparison of a program's execution in baseline SC (top) and proposed SC hardware (bottom) designs.

that access those reclassified variables. The read-after-write dependency guarantee is ensured since the compiler uniquely classifies each variable as either safe or unsafe, so both stores and loads to the variable will use the same access type. A discussion of an interesting corner case arising from distinct logical variables being mapped to the same physical address can be found in Section 5.2.

Our dynamic analysis could violate the memory-type guarantee only when an instruction accesses memory locations that span multiple pages. Fortunately, current architectures produce multiple micro-operations to execute such unaligned load accesses [33]. As a result, we preserve the memory-type guarantee for each load/store micro-operation, which is sufficient to ensure correct store-to-load forwarding. The read-after-write dependency guarantee could only be violated when a page transitions from private or shared-read-only to shared-read-write. But such a transition entails flushing the store buffers (see Section 6), thus the guarantee is maintained.

4.4. Illustration

Figure 2 depicts an example to illustrate the performance advantages of our SC hardware design. The top half of the picture illustrates a baseline SC hardware design and the bottom half illustrates the workings of our design. Figure 2a represents the initial states of the ROB and the store buffers for a program, and Figure 2b shows the events that take place in the store buffer and in the ROB along a timeline. Shaded cells represent safe accesses. Assume that only `St(X)` incurred a cache miss and the rest are cache hits. Finally, for simplicity, assume that the cache has one read and one write port.

The figure shows that, in the baseline design, `St(X)` is safe but is stalled at the head of the store buffer. This unnecessarily stalls the retirement of the following stores and also prevents the loads in the ROB from being committed. The loads in the ROB must wait to commit until after the cache miss is resolved and the store buffer is drained.

In our proposed SC design (bottom half of the picture), the long latency `St(X)` is sent to the unordered store buffer. This allows all the following safe (`St(A)`) and unsafe (`St(Y)`) stores to retire. Also, it allows safe (`Ld(Z)`) and unsafe (`Ld(B)`) loads to commit from the reorder buffer. Finally, observe that safe load `Ld(Z)` is allowed to commit even before the preceding unsafe store `St(A)` retires. The only memory ordering enforced is that unsafe load `Ld(B)` must wait to commit until the unsafe store `St(A)` retires, which results in a one cycle stall for the ROB commit. In contrast, in the SC baseline, ROB commit is stalled until the FIFO store buffer becomes empty. This stall can be significant depending on the number of pending stores that miss in the cache and the cache miss latency.

4.5. SC Memory Model Guarantees

The SC memory model requires that any program state that is made "externally" visible is *SC-reachable* in the sense that the state is reachable through an SC execution of the source program. We consider the program state read by a synchronous system call and the final program state to be externally visible. By construction, our SC-preserving hardware and compiler guarantee that the final program state is SC-reachable. To guarantee that any program state visible to a system call handler is SC-reachable, we only need to ensure that the store buffers of the processor core invoking the system call are drained before the system call handler is executed. This is already the case even with conventional processor designs that support precise context switches.

However, at an asynchronous interrupt (e.g., interrupt from an interactive debugger), we can only guarantee that the program state is SC-reachable for the shared variables but not for the private variables. For the private variables, we can only guarantee SC with respect to the compiled binary, because accesses to private variables may have been optimized and reordered by our SC-preserving compiler. But guaranteeing that the program state at an asynchronous interrupt is precise with respect to the source program is a more general problem that is known to be an issue even for sequential programs in the presence of compiler optimizations [2].

5. Static Classification of Memory Accesses

In this section we describe a static approach to classify memory instructions as either safe or unsafe. The compiler communicates this information to the hardware through dedicated bits in a memory instruction's machine code.

In the dynamic scheme described in Section 6, implementation efficiency requires that access patterns are tracked at the granularity of a memory page. This means that if a single byte on a page is accessed by multiple threads, then *all* locations on

```
void f(){
  int x;
  x=1;
  printf("%d\n",x);
}
```

(a)

```
void g(){
  int y;
  y=2;
  external_fn(&y);
  printf("%d\n",y);
}
```

(b)

```
void h(){
  int z;
  printf("%d\n",z);
  external_fn(&z);
}
```

(c)

```
void u(){
  union {
    int i;
    char * p;
  };
  p = (char *) ''SC'';
  printf(``%d\n'',i);
  external_fn(&i);
}
```

(d)

Figure 3: Local variables (x, y and z) in different functions in a program may map to the same physical location, complicating the unsafe versus safe distinction. A simple processor check can avoid violating program semantics. Union members (i and p in function u()) must have same memory access type.

that page must be treated as shared and suffer the performance consequences of strict ordering requirements when accessed. The static scheme described in this section has no runtime detection cost, and as such, nothing prevents us from treating an access to one byte as safe while treating an adjacent byte on the same page as unsafe.

5.1. Classification of Memory Accesses

Our static analysis runs at compile time and conservatively determines memory accesses that could potentially access mutable shared variables and marks them as unsafe. The remaining accesses to private and read-only shared variables are marked as safe.

The analysis first classifies program variables:

- **Safe variable:** A variable is classified as safe only if the compiler can statically guarantee that it is either a read-only variable or will be accessed by only a single thread during its lifetime.
- **Unsafe variable:** A variable that is not safe is classified as unsafe. It may be accessed by multiple threads during its lifetime.

Once program variables have been classified, the analysis can classify memory accesses:

- **Safe access:** A memory instruction that accesses one or more safe variables and does not access any unsafe variables is classified as safe.
- **Unsafe access:** A memory instruction that accesses one or more unsafe variables and does not access any safe variables is classified as unsafe.

It is possible that a memory instruction accesses both safe and unsafe variables (e.g., an instruction dereferencing a pointer that can map to variables of both types). We refer to such instructions as "mixed accesses". In order to ensure correct store-to-load forwarding on our specialized hardware, all accesses to a variable must be either safe or unsafe. To accomplish this, our compiler marks a mixed-access as unsafe *and* also demotes any mutable safe variable that it accesses to an unsafe variable. This step may now cause some safe accesses to become mixed or unsafe accesses. We iterate this step till all accesses are either classified as safe or unsafe.

Sophisticated sharing and thread escape analysis [16, 50] could be used to perform the initial classification of program variables. But rather than use a heavyweight, inter-procedural analysis, our compiler relies on simple modular information to conservatively determine if an access is safe. Global variables, dynamically allocated heap objects, and static variables are all considered unsafe. This leaves only function parameters and function locals as potentially safe variables.

Our compiler is built on top of LLVM which already performs a simple analysis to identify non-escaping, function-local variables (i.e. those variables whose address is not taken using the & operator). Our compiler takes advantage of this existing analysis and marks these non-escaping variables as safe. Stack locations used by the compiler for register spilling are also classified as safe. Finally, literals (shared or private) are classified as safe as well.

5.2. Ensuring Correctness for Hardware with Two Store Buffers

As mentioned in Section 4.3, store-to-load forwarding is only performed between loads and stores of the same memory access type. For instance, an unsafe store to a memory location L which is queued in the unsafe FIFO store buffer will not be forwarded to a safe load from L. Thus, maintaining correct program semantics requires that the compiler mark a load and a store that access the same memory location with the same access type.

The algorithm described above maintains this invariant for accesses to a location within a function: only non-escaping local variables and compiler temporaries, neither of which can have aliases, are marked as safe. Furthermore, demoting safe variables touched by mixed accesses to unsafe and reclassifying the variables' accesses guarantees that all accesses are either entirely to safe or unsafe variables.

However, this intra-procedural analysis does not account for location reuse across different functions. Consider the example functions shown in Figures 3a and 3b. Both function f and g contain a single local variable. In f, our compiler marks x as safe, while in g, it must mark y as unsafe since it escapes the function and may be accessed by another thread. Our compiler may store both x and y on the stack.[1] Now consider some code that calls function f and then calls g. Both x and y will be stored in the same physical location due to the runtime call stack growing and shrinking on function call and return. Furthermore, it is essential that the write to y complete (retire from the store buffer) after the write to x, otherwise we risk violating even sequential program semantics.

In order to ensure that such code executes correctly, we extend our hardware design to perform an additional check for every store. Before committing a safe store from the ROB, the processor checks the FIFO store buffer with unsafe stores for any conflicting store (a store with the same address), and vice versa. If a conflicting store is found in the other buffer, the commit is delayed until the conflicting store retires. This scenario is a rare occurrence, because it is unlikely that two

[1] In function f, the compiler might use a register for x and never assign it a physical memory location. Nevertheless, it is valid behavior to store x on the stack.

function-local variables mapped to the same physical location will be of different type *and* both have stores in-flight at the same time.

Note that this additional processor check does not necessarily prevent a safe load from executing while a store to the same physical location (though different logical variable) is in the unsafe store buffer, or vice versa. Consider the functions in Figures 3a and 3c. If the compiler decides to store both safe variable `x` and unsafe variable `z` on the stack, and some code first calls `f` and then calls `h`, then `x` and `z` will both use the same physical location. On current hardware, this means that both `f` and `h` will print the number "1" to the console. In our design, it is possible that `h` will print a value that was stored at that physical location prior to execution of `f`. However, this does not violate program semantics, since the result of reading an uninitialized variable (as `h` does in the `printf` statement) is not well defined. In fact, nothing requires that the compiler store `x` on the stack at all, so there can be no expectation that `h` will print "1". If `z` was indeed initialized by a store, then any following read to `z` in the thread would correctly receive that store's value as both of those accesses would be guaranteed to be of the same type.

Additional care must also be taken with local variables of union type. For instance, notice that the address of `p` is never taken in Figure 3d. But, because it is essentially an alias for `i` which does have its address taken, our static analysis must classify both variables as unsafe.

5.3. CISC Architecture

We have so far assumed that a memory instruction in a program's binary can access only one variable. However, in the CISC architecture an instruction may access multiple variables. For such instructions, we propose to extend the ISA to provide one extra bit per memory operand in the instruction's machine code. This will allow our compiler to mark each memory access in a memory instruction as safe or unsafe. A processor can use this information to classify a micro-operation generated for each memory access in a CISC instruction as safe or unsafe.

6. Dynamic Classification of Memory Accesses

A static technique does not have the benefit of observing the actual runtime stream of memory accesses. It must conservatively classify accesses at compile time. Therefore, we discuss a complementary dynamic technique for determining if a memory access is safe or not. As we described in Section 3, an access to private or shared, read-only locations is safe. To determine safe accesses, we leverage the hardware memory management unit (MMU) and the OS page protection mechanism [15, 17, 27].

6.1. Background: Process-Level Page Protection

Current systems provide page protection at the process level. Each process has a page table that is shared among all the threads of the process. Each page table entry contains the read and write access permissions for a page. In the execute stage of a memory operation, after its effective address is resolved, this virtual address is translated by the processor to the corresponding physical address. To assist in fast translation, the processor uses a Translation Lookaside Buffer (TLB) in each core. Each TLB entry caches a page table entry for a thread executing on its processor core. It includes read and write permission bits, which are checked by the processor when it executes loads and stores respectively. A page-fault exception is raised to the OS on detecting a permission violation. On a TLB miss for an address, a TLB miss handler (hardware assisted page-table walker) is executed, which fetches the page entry from the main memory, and allocates and initializes a TLB entry for it.

6.2. Proposed Extension: Thread-Level Page Protection

A page table is shared by all the threads in the process. In order to detect page sharing among threads and determine safe accesses at runtime, we extend the page table entries to keep track of the sharing state for pages. Figure 4 shows the sharing states that a page can be in. We add the following fields to keep track of these states: (a) a thread identifier (`tID`), (b) a `Read-Only` bit, and (c) a `Shared` bit. Any access to a page in $\langle shared, rw \rangle$ state is considered unsafe, and all the others are considered to be safe.

We also extend the TLB entry with an additional `Safe` bit. A processor consults this bit during address translation to determine if an access to a page is safe, and if it is, it sets the `ds` bit for the access in the ROB. To support the static classification scheme, before committing a store, a processor needs to ensure that there is no conflicting store in the store buffer handling the opposite type (Section 5.2). However, this check is not needed if the store is classified as safe by the dynamic scheme (`ds` bit is set), irrespective of the static scheme's classification. This optimization is correct, because when the dynamic scheme classifies a store as safe, it is guaranteed that there cannot be any preceding unsafe store to the same address.

In the rest of the section we describe how the above states are maintained and how we guarantee memory ordering constraints when a page changes its state.

6.3. State Transitions and Guaranteeing Memory Ordering Constraints

When a page is allocated by a page fault handler, its state is set to $\langle untouched \rangle$ (Figure 4). Its `tID` is set to `INV` to indicate that no thread has executed a read or write to this page yet. Also, its `Read-Only` bit is set and `Shared` bit is reset.

The first thread to issue a read to a page will trigger a TLB miss. The TLB miss handler checks if the page has already been allocated. If so, it checks the `tID` of the page and determines that this read is the first access. It then sets the page state to $\langle private, ro \rangle$ by setting its `tID` field. It allocates a TLB entry, sets the safe bit, but resets the write permission in the TLB entry, irrespective of the write permission bit's value in the page table entry. This allows our system to detect when the same thread attempts a write to this page, as that would cause a page fault. The page fault handler can then check the write permission for the page in its corresponding page table entry. If the attempted write is legal, the page fault handler changes the state of the page to $\langle private, rw \rangle$. Also, the write permission for the page is enabled in the TLB entry to allow future writes from the same thread. The safe bit in the TLB entry would remain set.

When another thread issues a read to a page in the $\langle private, ro \rangle$ state, it would also incur a TLB miss. The TLB

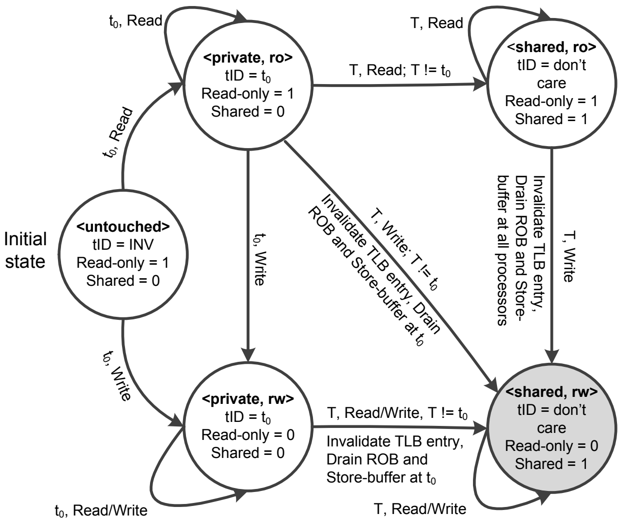


Figure 4: State transition of a page. Accesses to the shaded state are unsafe.

miss handler determines that the page has been read by another thread, and changes the state of the page to $\langle shared, ro\rangle$. An entry in the local TLB is allocated with the safe bit set, but the write permission is disabled.

The state of a page can transition to the unsafe $\langle shared, rw\rangle$ state from three different safe states as shown in Figure 4. Care must be taken during these state transitions to guarantee memory ordering constraints. Let us assume a thread `P` owns a page in $\langle private, ro\rangle$ state, and a remote thread `R` issues a store to that page. The TLB miss handler in the remote processor core running `R` determines that the page needs to be transitioned into the $\langle shared, rw\rangle$ state, because the `tID` in the page table entry would be different from `R`. Before modifying the state of the page table entry, the handler issues an inter-processor-interrupt (IPI) to the processor core running `P`.[2] When the processor core running `P` receives the IPI, the interrupt handler invalidates the corresponding entry in its local TLB, and sends an acknowledgment back to the core running `R`. Note that before any interrupt handler begins its execution, the processor flushes both the ROB and the store buffers in order to support a precise context switch. This behavior ensures correct memory ordering when a page transitions to the unsafe state. On receiving the acknowledgment, the TLB miss handler of `R` updates the state of the page to $\langle shared, rw\rangle$, and allocates a TLB entry for the page by initializing it to the permission bits in the page table entry, but resets the `Safe` bit. Thereafter, all accesses to the page will be treated as unsafe and ordered correctly.

The state transition from $\langle private, rw\rangle$ to $\langle shared, rw\rangle$ is handled similarly. The state transition from $\langle shared, ro\rangle$ to $\langle shared, rw\rangle$ is also similar, except that an IPI needs to be broadcast to all processor cores. To prevent races between page state updates, the TLB miss handler and the page fault handler always acquire a lock for a page before updating its page table entry.

TLB invalidations through inter-processor-interrupts could be expensive. Fortunately, this cost is incurred only once per page during an execution of a program. This allows us to provide a low-complexity hardware solution. Notice that other than the maintenance and use of the `Safe` bit, the changes required are restricted to the system software and TLB miss handler implementation.

6.4. Initialization Phase

Usually in a parallel program, the main thread initializes several data-structures before spawning threads. We do not want to classify a page as $\langle shared, rw\rangle$ just because it was modified by the main thread during initialization. Therefore, we reset the state of all pages to $\langle untouched\rangle$ just before the main thread creates the second thread for the process. This logic can be extended further to periodically reset the state of pages, but we leave this for future work.

6.5. Context Switches

We do not store the `tID` in a TLB entry. Therefore, when a thread is context switched out, the processor core cannot determine that the `Safe` bits in the TLB entries belong to the older thread. This problem of virtualizing the TLB across context switches is also a problem for supporting process-level page protection. Many processor implementations employ a simple solution that flushes the TLB entries on a context switch, which is sufficient to ensure correctness for our design as well. However, some newer implementations maintain additional tags in each TLB entry to efficiently support virtualization [1, 14, 47]. A similar hardware design could also allow us to support TLB virtualization while providing thread-level page protection.

6.6. Direct Memory Accesses (DMA)

Modern systems support Direct Memory Access (DMA) to efficiently transfer data from a slower physical device directly to main memory without involving the processor core's computational resources. However, this raises the question of what semantics the system should provide in case of a data race between the DMA transfer and concurrent accesses within the processor cores [31]. We leverage the observation made by Dunlap et al. [17] that the DMA transfer occurs between well-defined boundaries, and none of the processor cores should access the affected locations during that interval. This property can be explicitly enforced by the OS by acquiring access privileges to pages on behalf of the device and releasing them once the transaction is completed [17]. Another alternative is to temporarily change the state of pages that DMA can access to the unsafe state, and then restore the original state after the DMA transfer completes. Both of these alternatives would ensure SC even in the presence of DMA accesses. Another simpler option would be to assume that the system is properly synchronized with respect to DMA, and make no guarantee when races exist between DMA accesses and regular processor core accesses.

7. Results

In this section, we evaluate our low-complexity SC hardware's performance. Our evaluation answers the following questions:

- What is the performance overhead of our SC hardware design when compared to TSO? What is the advantage

[2] The TLB miss handler can determine the processor core running `P` by checking `P`'s Thread-Control-Block (TCB) maintained by the OS.

Table 1: Processor Configuration

Processor	16 cores operating at 4 GHz
Fetch/Exec/Commit	4 instructions (maximum 2 loads or 1 store) per cycle in each core
FIFO Store Buffer	64 8-byte entries
Unordered Store Buffer	8 64-byte entries; coalescing
L1 Cache	64 KB per-core private, 4-way set associative, 64 byte block size, 2-cycle hit latency, write-back
L2 Cache	512 KB private, 4-way set associative, 64 byte block size, 10-cycle hit latency.
Coherence	MOESI directory protocol
Interconnection	Torus-2D topology, 512-bit link width, 8-cycle link latency.
Memory	160 cycles (40 ns) DRAM lookup latency.

over baseline SC?

- What is the accuracy of our static and dynamic classification schemes when compared to a byte-level dynamic classification scheme?
- What is the performance overhead of guaranteeing end-to-end SC when compared to executing stock compiler's binary on TSO hardware?

7.1. Methodology

We modeled our hardware designs using a cycle-accurate, execution-driven, Simics-based, full-system simulator called FeS2 [19]. We modeled a 64-bit 16-core processor with an on-chip network. Details of the processor configuration are listed in Table 1. For our baseline SC and TSO processor, we assumed a 64-entry FIFO store buffer with 8-byte (one word) entries. For the proposed SC design, in addition to the 64-entry FIFO store buffer, we modeled another 8-entry unordered store buffer with 64-byte (one L1 cache block) entries. The unordered store buffer allows out-of-order retirement of stores and coalesces multiple stores to the same cache block. In Section 7.4 we evaluate the sensitivity of our design to various store buffer sizes.

For all of the SC and TSO designs, we implemented in-window speculative load execution as described in [22]. We also model exclusive prefetch [22] for stores which can reduce the latency of a store by obtaining the necessary write permission before the store is able to retire from the store buffer and write the cache block. Our TSO and SC simulations are functionally equivalent, because our front-end Simics functional simulator is SC. Our back-end timing simulator enforces the appropriate set of memory ordering constraints depending on the simulated memory model.

To implement the static classification scheme, we extended the LLVM [37] compiler to classify private accesses and communicate this information to the hardware through an ISA extension. To evaluate the cost of supporting end-to-end SC, we built an SC-preserving compiler as described in [43]. Currently, static classification is performed only for application code, because we were not able to recompile the Linux kernel and glibc using our compiler. Therefore, our evaluation underestimates the potential benefits of the static and hybrid classification schemes.

We evaluated three variants of the proposed SC design based on the memory access classification scheme: static only (SC-staticOnly), dynamic only (SC-dynamicOnly), and hybrid (SC-hybrid). Our schemes are conservative in classifying a memory access as safe. Therefore, we may misclassify a safe memory access as unsafe. To understand the accuracy of our classification schemes, we evaluated a hypothetical system that dynamically tracked the type of a memory location at the byte granularity (SC-ideal), which solves the false sharing problem in our page-level dynamic scheme. This fourth variant would be too expensive to realize in an actual hardware, but it is useful as a limit study.

Our benchmarks include the Apache web server and applications from the PARSEC [8] and SPLASH-2 [55] benchmark suites. We used the "simlarge" input set for PARSEC benchmarks. For barnes, we used a 65536 nbody simulation. For Apache, we used the SURGE [7] benchmark. For the SPLASH-2 benchmarks, we simulated the complete parallel section. For Apache, we warmed up the caches and micro-architectural structures for 20000 transactions, and then simulated the execution for the next 20000 transactions. It was not feasible to simulate the entire parallel section for the PARSEC benchmarks due to their long execution times. Therefore, for these programs, we sampled five checkpoints that span across the entire parallel section of the program. For each checkpoint, after the warmup phase (100K stores per core), we simulated at least 10 million stores for each processor core. We employed this sampling approach for comparing hardware designs running the same binary. However, measuring progress in terms of stores may not be accurate while comparing the performance of binaries produced by two different compilers (SC-preserving and stock compiler). Therefore, for such comparisons, we simulated the entire execution of the parallel section. While simulating the dynamic and hybrid schemes, we started tracking the state of a page only after the parallel section starts executing. We evaluated the performance of both user-level and system execution in our full-system simulation using instructions-per-cycle (IPC) as the performance metric.

7.2. Performance of Memory Access Type Driven SC Hardware

Figure 5 compares the performance of the proposed SC hardware to a baseline SC hardware design. The performance overhead of all configurations is shown relative to a TSO hardware design that is similar to modern x86 processor implementations. All the configurations use our SC-preserving compiler implementation. Therefore, SC hardware provides end-to-end SC and TSO hardware provides end-to-end TSO.

While our optimizations may not have an effect on programs that already provide good SC performance, they significantly reduce the overhead for those programs that do suffer a high performance penalty due to SC constraints. On average, SC-baseline has a performance overhead of about 9.1%. The maximum overhead for SC-baseline, however, is much higher: 28.8% (facesim). SC-staticOnly reduces the overhead to 5.1% on average, with a maximum of 13.5%. SC-dynamicOnly incurs only 2.9% overhead on average. The proposed SC design, SC-hybrid, which uses both static and dynamic classification schemes, has an average overhead of 2.0%. Worst case overhead for SC-hybrid is 5.4% (facesim) which is a significant reduction from the 28.8% (facesim) that we observe for SC-baseline. The proposed design's performance is close to that of SC-ideal , which uses a byte-level classification scheme. We conclude that our optimizations are effective in reducing the SC memory ordering overhead when it is present.

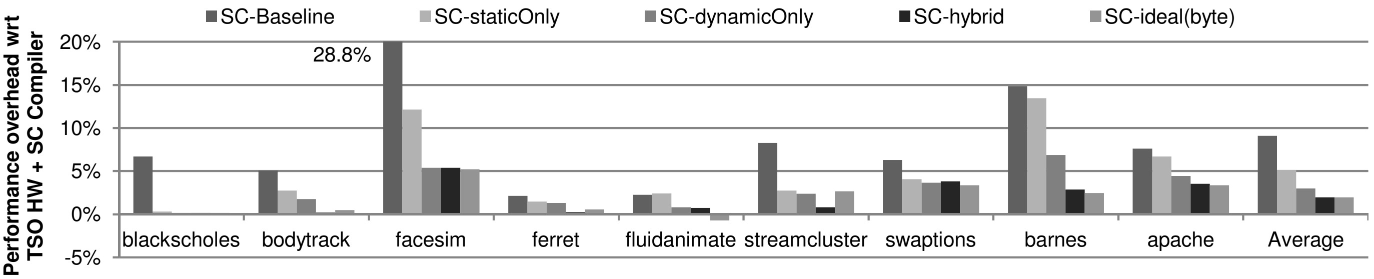


Figure 5: Performance of the baseline SC and variants of the proposed SC designs compared to TSO.

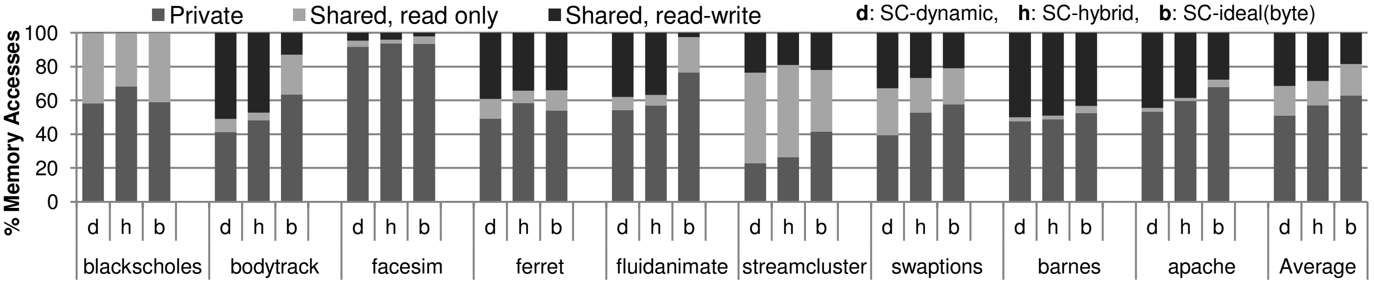


Figure 6: Classification of memory accesses by various methods.

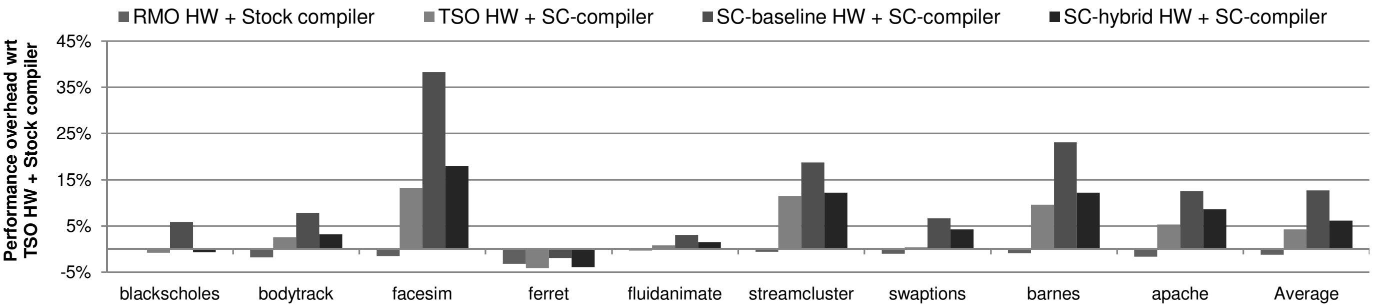


Figure 7: Comparison of our proposed system with baseline SC, TSO, and RMO hardware.

Figure 6 compares the accuracy of our classification schemes, which determines the effectiveness of our SC hardware optimizations. On average, our page-based dynamic scheme (`SC-dynamicOnly`) classifies 68.5% of memory accesses as safe. Combining this with the static scheme improves the accuracy further to 71.5%, which is close to the accuracy that one can achieve with byte-level tracking (81.5%).

Applications with a higher proportion of safe accesses (90% for `facesim`) benefit significantly from our optimization as we discussed earlier. However, proportion of unsafe accesses is not the only factor that determine the final SC hardware overhead. Cache miss rate of unsafe accesses have a more direct influence. For example, `SC-hybrid`'s overhead for `bodytrack` with more than 40% unsafe accesses is lower than that of `swaptions` with only about 25% unsafe accesses. This is because `bodytrack`'s cache miss rate is lower than `swaptions`.

Figure 6 shows that for some applications (`blackscholes, ferret`), `SC-hybrid` classifies more accesses as private than `SC-ideal`. This is because we classified all compiler specified safe accesses as private. However, some of these safe accesses could be to shared read-only locations (e.g, literals), which `SC-ideal` would more accurately classify as shared read-only accesses. While distinguishing between private and shared read-only accesses is useful to understand our classification schemes, it has no bearing on our performance studies, because they both are treated the same way in our SC hardware designs.

7.3. Cost of End-to-End SC

The cost of end-to-end SC is shown in Figure 7. We consider a TSO processor running the binary produced by the stock LLVM compiler as our baseline as it represents the most commonly used systems today. End-to-end SC has two sources of overhead: 1) the cost of preserving SC in the compiler, and 2) the cost of enforcing SC in the hardware. In Figure 7, we observe that the cost of preserving SC in the compiler is on average about 4.3% (`TSO HW + SC compiler`). This overhead could be further reduced using the interference checks described in [43]. If we use baseline SC hardware, the total end-to-end SC cost is about 12.7% on average. However, by using the hybrid classification scheme, our SC design reduces the cost to 6.2% on average. This overhead is only slightly higher (7.4%) when we compare to a relaxed memory model (RMO) hardware (which is sufficient to support C++ or Java memory model) executing the stock LLVM compiler's output.

7.4. Sensitivity to Store Buffer Sizes

Our `SC-hybrid` design assumed an additional unordered store buffer when compared to the `SC-baseline`. When we halved the size of the two store buffers in `SC-hybrid`, which made them area neutral with the store buffer in `SC-baseline`, the increase in performance overhead was negligible (less than 1%).

It is important to note that for a store buffer, the dominant cost is not area, but rather the latency and power cost of associative lookups. An associative lookup of the store buffer is necessary for each load to support store-to-load forwarding. In our design, a load has to search only one of the two store buffers. Thus, the additional unordered store buffer in our SC design does not aggravate this dominant overhead.

8. Related Work

To our knowledge, no past work has exploited memory access type to relax memory model constraints in hardware while still supporting SC. We have already discussed work on optimizing SC hardware in Section 2. Here we discuss a few other related works that provided end-to-end SC, and designs that exploited memory access type for improving system performance.

8.1. End-to-end Sequential Consistency

Hammond et al. [26] proposed transactional coherence and consistency (TCC). In TCC, a programmer ensures that every instruction is part of a transaction and a hardware transactional memory [29] ensures that execution of all transactions is serializable, which in turn guarantees SC at the language-level. BulkCompiler [6] and BulkSC [13] also provide end-to-end SC, but unlike programmer specified transactions, the BulkCompiler automatically partitions a program into regions called "chunks". These region-based solutions provide SC, but rely on fairly expensive speculation hardware (checkpointing, versioning, conflict detection, and recovery) to guarantee serializability of regions.

Researchers have also proposed to use static analysis for guaranteeing SC on hardware supporting weaker memory models. Shasha and Snir [51] proposed "delay set analysis", which finds the minimum number of fences required for an SC execution. A fence incurs significant performance penalty on current processors. To optimize this cost, Sura et al. [53] and Kamil et al. [34] used static analyses to identify shared accesses and insert fences only for these accesses. More recently, Lin et al. [38] proposed `conditional fence`. They employ hardware support to enforce a fence ordering only when there is a possibility that SC may be violated. However, all of these static approaches use whole program analyses, which are not scalable to real-world programs.

8.2. Enforcing Data-Race-Free Discipline

Current DRF0 [4] based memory models provide SC for data-race-free programs. Therefore, one option would be to use a sound static [11, 12, 21] data-race detector to reject racy programs at compile-time and enforce the data-race-free discipline assumed by the DRF0 memory model. However, static solutions need to be conservative in their analysis and report a number of false data-races. It would be unacceptable if a compiler rejects a valid race-free program. Instead of static analysis, researchers proposed to use a runtime mechanism [18, 40, 42] to dynamically detect SC violations due to a data-race and raise a memory model exception. However, runtime data-race detection in software incurs prohibitively high overhead [20], and custom hardwares [5, 40, 42, 45, 48, 52] are fairly complex. Furthermore, legacy software contains a number of data races that are deliberately used by programmers to achieve high performance [46]. A solution that raises an exception for these data races will face backwards compatibility issues.

8.3. Private and Shared Data Driven Architectures

Past work has leveraged the page-protection mechanism for improving data placement in a processor cache [27], reducing snoops in a token-based coherence protocol [35], detecting thread dependencies to support replay [17], and more recently to improve the efficiency of directory caches [15]. Unlike these solutions, our design goal is to relax memory model constraints, which requires us to carefully orchestrate the state transitions of a page to ensure that memory ordering constraints are not violated. We also employ a complementary static analysis technique to classify memory accesses.

9. Conclusions

The memory model of a concurrent language defines what values a load instruction can return. Semantics as fundamental as this should have a clean definition that matches the intuition of programmers. While the benefits of language-level sequential consistency are well known, an efficient *and* practically feasible solution for SC hardware has remained elusive.

We exploited an important opportunity that has been overlooked in the past while designing SC hardware: no memory model constraints need to be enforced on accesses to private locations and shared, read-only locations. By exploiting this observation, we derived a low-complexity SC hardware design that obviates the need for aggressive speculation to obtain high performance. It uses a combination of static analysis and the page protection mechanism to identify safe accesses and relax SC constraints on them. Apart from an additional unordered store buffer, there is very little hardware modification needed to support our design. Our end result is promising: SC hardware is only 2.0% slower than TSO, and end-to-end SC costs only about 6.2% when compared to the performance of a state-of-the-art compiler and TSO hardware.

For the SC memory model to be adopted at the language level, all the compilers and processors that support the language should be made SC-preserving. While our study considered one of the most widely used processor designs as baseline (an out-of-order TSO processor), further study is needed to understand the overhead due to end-to-end SC in other classes of systems (e.g., a low power in-order architecture may be important for embedded systems).

10 Acknowledgements

We would like to thank Sarita Adve and anonymous reviewers for their valuable suggestions. This work is supported by the National Science Foundation under awards CNS-0725354, CNS-0905149, and CCF-0916770 as well as by the Defense Advanced Research Projects Agency under award HR0011-09-1-0037.

References

[1] AMD Corporation. AMD-V Nested Paging. *White paper. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx*, 2008.

[2] A.-R. Adl-Tabatabai and T. Gross. Source-Level Debugging of Scalar Optimized Code. In *PLDI*, 1996.

[3] S. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin–Madison, 1993.

[4] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA*, 1990.

[5] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *ISCA*, 1991.

[6] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, 2009.

[7] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *SIGMETRICS*, 1998.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[9] C. Blundell, M. Martin, and T. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *ISCA*, 2009.

[10] H. J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[12] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[13] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, 2007.

[14] Compaq Computer Corporation. Alpha 21264 Microprocessor Hardware Reference Manual. *Order Number: EC-RJRZA-TE.*

[15] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *ISCA*, 2011.

[16] J. deok Choi, M. Gupta, M. J. Serrano, V. C, and S. P. Midkiff. Stack Allocation and Synchronization Optimizations for Java using Escape Analysis. *ACM TOPLAS*, 2003.

[17] G. W. Dunlap, D. G. Lucchetti, M. Fetterman, and P. M. Chen. Execution Replay on Multiprocessor Virtual Machines. In *VEE*, 2008.

[18] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, 2007.

[19] The FeS2 simulator. `http://fes2.cs.uiuc.edu/`.

[20] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[21] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, pages 219–232, 2000.

[22] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP*, 1991.

[23] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *PACT*, 2002.

[24] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP=RC? In *ISCA*, 1999.

[25] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *ASPLOS*, 2004.

[26] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, 2004.

[27] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *ISCA*, 2009.

[28] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. 2010.

[29] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.

[30] M. D. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer*, 31:28–34, 1998.

[31] M. D. Hill, A. E. Condon, M. Plakal, and D. J. Sorin. A System-Level Specification Framework for I/O Architectures. In *SPAA*, 1999.

[32] Intel® Corporation. Pentium® Pro Family Developer's Manual. 1996.

[33] Intel® Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. *Order Number: 248966-025*, 2011.

[34] A. Kamil, J. Su, and K. Yelick. Making Sequential Consistency Practical in Titanium. In *ICS*, 2005.

[35] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace Snooping: Filtering Snoops with Operating System Support. In *PACT*, 2010.

[36] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Computer*, 1979.

[37] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[38] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, 2010.

[39] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, 2012.

[40] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions. In *ISCA*, 2010.

[41] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, 2005.

[42] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, 2010.

[43] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, 2011.

[44] S. P. Midkiff, D. A. Padua, and R. Cytron. Compiling Programs with User Parallelism. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, 1990.

[45] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. SigRace: Signature-based Data Race Detection. In *ISCA*, 2009.

[46] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races using Replay Analysis. In *PLDI*, 2007.

[47] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), 2006.

[48] M. Prvulovic and J. Torrelas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded codes. In *ISCA*, June 2003.

[49] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, 1997.

[50] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP*, 2001.

[51] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS*, 10(2), 1988.

[52] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, 2011.

[53] Z. Sura, X. Fang, C. Wong, S. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, 2005.

[54] T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *ISCA*, 2007.

[55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.