A Case for an SC-Preserving Compiler

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

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

Abstract

The most intuitive memory consistency model for shared-memory multi-threaded programming is *sequential consistency* (SC). However, current concurrent programming languages support a relaxed model, as such relaxations are deemed necessary for enabling important optimizations. This paper demonstrates that an SC-preserving compiler, one that ensures that every SC behavior of a compiler-generated binary is an SC behavior of the source program, retains most of the performance benefits of an optimizing compiler. The key observation is 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. An SC-preserving compiler, obtained by restricting the optimization phases in LLVM, a state-of-the-art C/C++ compiler, incurs an average slowdown of 3.8% and a maximum slowdown of 34% on a set of 30 programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites.

While the performance overhead of preserving SC in the compiler is much less than previously assumed, it might still be unacceptable for certain applications. We believe there are several avenues for improving performance without giving up SC-preservation. In this vein, we observe that the overhead of our SC-preserving compiler arises mainly from its inability to aggressively perform a class of optimizations we identify as *eager-load* optimizations. This class includes common-subexpression elimination, constant propagation, global value numbering, and common cases of loop-invariant code motion. We propose a notion of *interference checks* in order to enable eager-load optimizations while preserving SC. Interference checks expose to the compiler a commonly used hardware speculation mechanism that can efficiently detect whether a particular variable has changed its value since last read.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms Languages, Performance

Keywords memory consistency models, sequential consistency, SC preservation, interference checks

1. Introduction

A memory consistency model (or simply *memory model*) defines the semantics of a concurrent programming language by specifying the order in which memory operations performed by one thread become visible to other threads in the program. The most natural memory model is sequential consistency (SC) [31]. Under SC, the individual operations of the program appear to have executed in a global sequential order consistent with the per-thread program order. This semantics matches the intuition of a concurrent program's behavior as a set of possible thread interleavings.

It is commonly accepted that programming languages must relax the SC semantics of programs in order to allow effective compiler optimizations. This paper challenges that assumption by demonstrating an optimizing compiler that retains most of the performance of the generated code while preserving the SC semantics. A compiler is said to be *SC-preserving* if every SC behavior of a generated binary is guaranteed to be an SC behavior of the source program.

Starting from LLVM [32], a state-of-the-art C/C++ compiler, we obtain an SC-preserving compiler by modifying each of the optimization passes to conservatively disallow transformations that might violate SC.[1] Our experimental evaluation (Section 3) indicates that the resulting SC-preserving compiler incurs only 3.8% performance overhead on average over the original LLVM compiler with all optimizations enabled on a set of 30 programs from the SPLASH-2 [49], PARSEC [7], and SPEC CINT2006 (integer component of SPEC CPU2006 [26]) benchmark suites. Moreover, the maximum overhead incurred by any of these benchmarks is just over 34%.

1.1 An Optimizing SC-Preserving Compiler

The empirical observation of this paper is 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. Several common optimizations, including procedure inlining, loop unrolling, and control-flow simplification, do not change the order of memory operations and are therefore naturally SC-preserving. Other common optimizations, such as common subexpression elimination (CSE) and loop-invariant code motion, can have the effect of reordering memory operations. However, these optimizations can still be performed on accesses to thread-local variables and compiler-generated temporary variables. The analysis required to distinguish such variables is simple, modular, and is already implemented by modern compilers such as LLVM. Furthermore, transformations involving a single shared variable are also SC-preserving under special cases (Section 2).

Consider the instance of CSE in Figure 1, where the compiler eliminates the subexpression `X*2`. By reusing the value of `X` read at `L1` in `L3`, this transformation effectively reorders the second access to `X` with the access to `Y` at `L2`. While invisible to sequential

[1] The SC-preserving version of LLVM is available at `http://www.cs.ucla.edu/~todd/research/memmodels.html`.

```
    Original            Transformed           Concurrent
                                                Context
L1: t = X*2;         L1: t = X*2;      |  N1: X = 1;
L2: u = Y;      ⇒    L2: u = Y;        |  N2: Y = 1;
L3: v = X*2;         M3: v = t;        |

     (a)                  (b)                 (c)
```

Figure 1: A compiler transformation from program (a) into (b) that eliminates the common subexpression `X*2`. In the presence of a concurrently running thread (c) and an initial state where all variables are zero, (b) can observe a state `u == 1 && v == 0`, which is not visible in (a). Lowercase variables denote local temporaries, while uppercase variables are potentially shared.

programs, this reordering can introduce non-SC behaviors in a concurrent program, as shown in Figure 1. However, an SC-preserving compiler can still perform this transformation as long as at least one of `X` and `Y` is known to be thread-local. If `X` is thread-local, then its value does not change between `L1` and `L3` and so the transformation is SC-preserving. On the other hand, if `Y` is thread-local then any SC execution of the transformed program can be shown to be equivalent to an SC execution of the original program in which instructions `L1` to `L3` execute without being interleaved with instructions from other threads. By carefully enabling transformations only when they are SC-preserving, our compiler is able to achieve performance comparable to a traditional optimizing compiler while retaining the strong SC semantics.

1.2 Providing End-to-End Programmer Guarantees

Providing end-to-end SC semantics to the programmer requires executing the output of an SC-preserving compiler on SC hardware. The empirical results in this paper complement recent architecture research [8, 15, 23, 24, 28, 44] that demonstrates the feasibility of efficient SC hardware. The basic idea behind these proposals is to speculatively reorder memory operations and recover in the rare case that these reorderings can become visible to other processors. While such speculation support necessarily increases hardware complexity, we hope that our work on an SC-preserving compiler increases the incentives for building SC hardware, since in combination they enable end-to-end SC semantics for programmers at a reasonable cost.

Even in the absence of SC hardware, the techniques described in this paper can be used to provide strong semantics to the programmer. For instance, when compiling to x86 hardware, which supports the relatively-strong total store order (TSO) memory model [40], a compiler that preserves TSO behavior (Section 7.2) provides TSO semantics at the programming language level. The result is a language-level memory model that is stronger and simpler than the current memory-model proposals for C++ [6, 11] and Java [36].

1.3 Speculative Optimization For SC-Preservation

While the cost of an SC-preserving compiler is much less than previously assumed, one possible concern is that some applications might be unwilling to pay this cost, however small. We argue that one should exhaust possible avenues for improving the performance of SC-preservation, such as more sophisticated static and dynamic analyses, before exposing a relaxed program semantics to the programmer.

In this vein, we observe that a number of disabled optimizations responsible for lost performance in our SC-preserving compiler involve an *eager load*. For instance, the elimination of the expression `X*2` in Figure 1 can be considered as performing the load of variable `X` eagerly at line `L1` instead of at `L3`. Other eager-load optimizations include constant propagation, copy propagation, partial-redundancy elimination, global value numbering, and common cases of loop-

```
                         L1: t = X*2;
L1: t = X*2;             L2: u = Y;
L2: u = Y;        ⇒      M3: v = t;
L3: v = X*2;             C3: if(X modified since L1)
                         L3:   v = X*2;

    (a)                           (b)
```

Figure 2: Performing common subexpression elimination while guaranteeing SC. The interference check at `C3` ensures that the value of `X` has not changed since last read at `L1`. This allows the compiler to reuse the value of `X*2` computed in `L1` without violating SC.

invariant code motion. Our experiments show that fully enabling these eager-load optimizations in our compiler reduces the maximum slowdown of any benchmark from 34% to 6.5%.

To enable eager-load optimizations without violating SC, we propose the use of compiler-inserted *interference checks* to dynamically ensure the correctness of optimizations that cannot be statically validated as SC-preserving (Section 4). Figure 2 demonstrates this idea. The figure shows the code from Figure 1(a) and its transformation with an interference check. For the CSE optimization to be sequentially valid, the compiler already ensures that the variable `X` is not modified by instructions between `L1` and `L3`. The interference check lifts this correctness requirement to concurrent programs by ensuring that no other thread has modified `X` since last read at `L1`. If the check succeeds, the program can safely reuse the earlier computation of `X*2`; if not, the program reverts to the unoptimized code.

Our interference checks are inspired by a common hardware speculation mechanism [23] that is used to safely strengthen hardware memory models. This mechanism allows a processor to track cache-coherence messages to conservatively detect when a particular memory location may have been modified by another processor. We observe that this speculation mechanism can be used to discharge our interference checks efficiently. We describe a simple interface for exposing this capability to the compiler, based on the Itanium architecture's design of a similar feature [29] (Section 5). We have built a hardware simulator supporting our speculation mechanism and have performed a simulation study on 15 parallel programs from the SPLASH-2 and PARSEC benchmarks. By incorporating interference checks into a single optimization pass, we reduce the average performance overhead of our SC-preserving compiler on simulated TSO hardware from 3.4% to 2.2% and reduce the maximum overhead from 23% to 17% (Section 6).

2. Compiler Optimizations as Memory Reorderings

In this section, we classify compiler optimizations based on how they affect the memory reorderings of the program [2, 47].

2.1 SC-Preserving Transformations

Informally, we represent the (SC) behaviors of a program as a set of interleavings of the individual memory operations of program threads that respect the per-thread program order. A compiler transformation is SC-preserving if every behavior of the transformed program is a behavior of the original program. Note that it is acceptable for a compiler transformation to reduce the set of behaviors.

Transformations involving thread-local variables and compiler-generated temporaries are always SC-preserving. Furthermore, some transformations involving a single shared variable are SC-preserving [47]. For example, if a program performs two consecutive loads of the same variable, as in Figure 3(a), the compiler can remove the second load. This transformation preserves SC as any execution of the transformed program can be emulated by an interleaving of the original program where no other thread executes

a) redundant load:	`t=X; u=X;`	⇒	`t=X; u=t;`
b) forwarded load:	`X=t; u=X;`	⇒	`X=t; u=t;`
c) dead store:	`X=t; X=u;`	⇒	`X=u;`
d) redundant store:	`t=X; X=t;`	⇒	`t=X;`

Figure 3: SC-preserving transformations

```
                                    |                     u = X*X;
                                    |   for(...){        for(...){
L1: X = 1;        L1: X = 1;        |     ...              ...
L2: P = Q;   ⇒   L2: P = Q;        |     P = Q;     ⇒     P = Q;
L3: t = X;        L3: t = 1;        |     t = X*X;         t = u;
                                    |     ...              ...
                                    |   }                }

              (a)                   |            (b)
```

Figure 4: Examples of eager-load optimizations include constant/copy propagation (a) and loop-invariant code motion (b). Both involve relaxing the $\mathtt{L} \longrightarrow \mathtt{L}$ and $\mathtt{S} \longrightarrow \mathtt{L}$ ordering constraints.

between the two loads. On the other hand, this transformation reduces the set of behaviors, as the behavior in which the two loads see different values is not possible after the transformation.

Similar reasoning can show that the other transformations shown in Figure 3 are also SC-preserving. Further, a compiler can perform these transformations even when the two accesses on the left-hand side in Figure 3 are separated by local accesses, since those accesses are invisible to other threads.

2.2 Ordering Relaxations

Optimizations that are not SC-preserving change the order of memory accesses performed by one thread in a manner that can become visible to other threads. We characterize these optimizations based on relaxations of the following ordering constraints among loads and stores that they induce: $\mathtt{L} \longrightarrow \mathtt{L}$, $\mathtt{S} \longrightarrow \mathtt{L}$, $\mathtt{S} \longrightarrow \mathtt{S}$, and $\mathtt{L} \longrightarrow \mathtt{S}$.

Consider the CSE example in Figure 1(a). This optimization involves relaxing the $\mathtt{L} \longrightarrow \mathtt{L}$ constraint between the loads at `L2` and `L3`, moving the latter to be performed right after the first load of `X` at `L1`, and eliminating it using the transformation in Figure 3(a). If the example contained a store, instead of a load, at `L2`, then performing CSE would have involved an $\mathtt{S} \longrightarrow \mathtt{L}$ relaxation. We classify an optimization as an *eager load* if it only involves $\mathtt{L} \longrightarrow \mathtt{L}$ and $\mathtt{S} \longrightarrow \mathtt{L}$ relaxations, as these optimizations involves performing a load earlier than it would have been performed before the transformation.

Another example of an eager load optimization is constant/copy propagation as shown in Figure 4(a). In this example, the transformation involves moving the load of `X` to immediately after the store of `X` (which requires $\mathtt{L} \longrightarrow \mathtt{L}$ and $\mathtt{S} \longrightarrow \mathtt{L}$ relaxation with respect to the `P` and `Q` accesses) and then applying the transformation in Figure 3(b). The loop-invariant code motion example in Figure 4(b) involves eagerly performing the (possibly unbounded number of) loads of `X` within the loop once before the loop. This also requires relaxing $\mathtt{L} \longrightarrow \mathtt{L}$ and $\mathtt{S} \longrightarrow \mathtt{L}$ ordering constraints due to the store and load to shared variables `P` and `Q` respectively.

Figure 5 shows examples of optimizations that are *not* eager loads. The dead-store elimination example in Figure 5(a) involves relaxing the $\mathtt{S} \longrightarrow \mathtt{S}$ and $\mathtt{S} \longrightarrow \mathtt{L}$ constraints by delaying the first store and then applying the SC-preserving step of combining the adjacent stores as in Figure 3(c). Figure 5(b) shows an example of a redundant store elimination that involves eagerly performing the store of X by relaxing the $\mathtt{L} \longrightarrow \mathtt{S}$ and $\mathtt{S} \longrightarrow \mathtt{S}$ ordering constraints and then applying the transformation in Figure 3(d).

```
X = 1;        ;          |  t = X;        t = X;
P = Q;   ⇒   P = Q;     |  P = Q;   ⇒   P = Q;
X = 2;        X = 2;     |  X = t;        ;

       (a)               |         (b)
```

Figure 5: (a) Dead store elimination involves relaxing the $\mathtt{S} \longrightarrow \mathtt{S}$ and $\mathtt{S} \longrightarrow \mathtt{L}$ constraints. (b) Redundant store elimination involves relaxing the $\mathtt{L} \longrightarrow \mathtt{S}$ and $\mathtt{S} \longrightarrow \mathtt{S}$ constraints.

3. An SC-Preserving Modification to LLVM

This section describes the design and implementation of our optimizing SC-preserving compiler on top of LLVM and evaluates the compiler's effectiveness in terms of performance of the generated code versus that of the baseline LLVM compiler.

3.1 Design

As described in the previous section, we can characterize each compiler optimization's potential for SC violations in terms of how it reorders memory accesses. In order to build our SC-preserving compiler, we examined each transformation pass performed by LLVM and determined whether or not it could potentially reorder accesses to shared memory. We further classified these passes based on what types of accesses might be reordered.

Perhaps surprisingly, many of LLVM's passes do not relax the order of memory operations at all and these SC-preserving passes can be left unmodified. These passes include sparse conditional constant propagation, dead argument elimination, control-flow graph simplification, procedure inlining, scalar replication, allocation of function-local variables to virtual registers, correlated value propagation, tail-call elimination, arithmetic re-association, loop simplification, loop rotation, loop unswitching, loop unrolling, unreachable code elimination, virtual-to-physical register allocation, and stack slot coloring.

Other LLVM optimizations can relax the order of memory operations. Table 1 lists these optimization passes and classifies the kinds of relaxations that are possible in each. To ensure that the compiler would be SC-preserving, we disabled a few of these passes and modified the remaining passes to avoid reordering accesses to potentially shared memory.

3.2 Implementation

Our compiler does not perform any heavyweight and/or whole-program analyses to establish whether or not a location is shared (e.g., thread-escape analysis). Rather we use simple, conservative, local information to decide if a location is potentially shared. During an early phase of compilation, LLVM converts loads and stores of non-escaping function-local variables into reads and writes of virtual registers. Operations on these virtual registers can be freely reordered. In certain situations, structures that are passed by value to a function are accessed using load and store operations. Our compiler recognizes these situations and allows these memory operations to be reordered in any sequentially valid manner. In addition, shared memory operations may be reordered with local operations. Thus, for instance, we can safely allow the "instcombine" pass to transform `t=X; t+=u; t+=X;` into `t=X ≪ 1; t+=u;` when both `t` and `u` are local variables.

Incorporating our modifications to LLVM was a fairly natural and noninvasive change to the compiler code. LLVM already avoids reordering and removing loads and stores marked as being *volatile*. Therefore, in the IR optimization passes we were often able to use existing code written to handle volatiles in order to restrict optimizations on other accesses to shared memory. The primary mechanism by which we avoided reordering during the x86 code generation passes was by "chaining" memory operations to one

Table 1: This table lists the passes performed by a standard LLVM compilation for an x86 target that have the potential to reorder accesses to shared memory. The table indicates which memory orderings may be relaxed and whether our SC compiler disables the pass entirely or modifies it to avoid reordering.

Short Name	**Description**	$L \rightarrow L$	$L \rightarrow S$	$S \rightarrow L$	$S \rightarrow S$	**SC Version**
	LLVM IR Optimization Passes					
instcombine	Performs many simplifications including algebraic simplification, simple constant folding and dead code elimination, code sinking, reordering of operands to expose CSE opportunities, limited forms of store-to-load forwarding, limited forms of dead store elimination, and more.	yes	no	yes	no	modified
argpromotion	Promotes by-reference parameters that are only read into by-value parameters; by-value struct types may be changed to pass component scalars instead.	yes	no	yes	no	disabled
jump-threading	Recognizes correlated branch conditions and threads code directly from one block to the correlated successor rather than executing a conditional branch. While this threading in itself would not reorder memory accesses, this pass performs some partially redundant load elimination to enable further jump threading, and that may have the effect of performing an eager load.	yes	no	yes	no	modified
licm	Performs loop-invariant code motion and register promotion.	yes	yes	yes	yes	modified
gvn	The global value numbering pass performs transformations akin to common subexpression elimination, redundant and partially redundant load elimination, and store-to-load forwarding.	yes	no	yes	no	modified
memcopyopt	Performs several optimizations related to memset, memcpy, and memmov calls. Individual stores may be replaced by a single memset. This can cause observable reordering of store operations (e.g. `A[0]=-1; A[2]=-1; A[1]=-1` becomes `memset(A,-1,sizeof(*A)*3)`. This pass can also introduce additional loads not present in the original program through a form of copy propagation.	no	yes	no	yes	disabled
dse	Performs dead store elimination and redundant store elimination as described in Figure 5	no	yes	yes	yes	disabled
	x86 Code Generation Passes					
seldag	Builds the initial instruction selection DAG. Performs some CSE during construction.	yes	no	no	no	modified
nodecombine	Performs forms of CSE, constant folding, strength reduction, store-to-load forwarding, and dead store elimination on the selection DAG. Can reduce atomicity of certain operations; for instance a store of a 64-bit float that can be done atomically on some architectures may be changed to two 32-bit integer stores. Also, bit-masking code may be recognized and changed to smaller operations without masking. This can have the effect of reordering a store with prior loads.	yes	yes	no	no	modified
scheduling	Schedules machine instructions.	yes	no	no	no	modified
machinesinking	Sinks load instructions and dependent computation to successor blocks when possible to avoid execution on code paths where they are not used.	yes	no	no	no	modified

another in program order in the instruction selection DAG. This indicates to the scheduler and other passes that there is a dependence from each memory operation to the next and prevents them from being reordered.

3.3 Example

The example in Figure 6 helps illustrate why an SC-preserving compiler can still optimize programs effectively. The source code shown in part (a) of the figure is a simplifed version of a performance-intensive function in one of our benchmarks. The function calculates the distance between two n-dimensional points represented as (possibly shared) arrays of floating point values. In addition to performing the floating point operations that actually calculate the distance, directly translating this function into x86 assembly would allocate space on the stack for the locally declared variables and perform four address calculations during each iteration of the loop. Each address calculation involves an integer multiply and an integer add operation as hinted by the comments in Figure 6 (a). Our SC-preserving compiler is able to perform a variety of important optimizations on this code:

- Since the locally declared variables (including the parameters) do not escape the function, they can be stored in registers rather than on the stack.
- CSE can be used to remove two of the address calculations since they are redundant and only involve locals.
- Loop-induction-variable strength reduction allows us to avoid the multiplication involved in the two remaining address calculations by replacing the loop variable representing the array index with a loop variable representing an address offset that starts at zero and is incremented by 4 each iteration.
- Using loop-invariant code motion (and associativity of addition), we can increment the array addresses directly during each iteration rather than incrementing an offset and later adding it to the base addresses.

The final result of applying the above SC-preserving optimizations is shown in part (b) of the figure (using C syntax rather than x86 assembly). The fully optimizing compiler that does not preserve SC is able to perform one additional optimization: it can use CSE to eliminate the redundant floating point loads and subtraction in each iteration of the loop. The resulting code is shown in part (c) of the figure.

```
float Distance(
    float* x, float*y, int n){
  float sum = 0;
  int i=0;

  for(i=0; i<n; i++){
     sum += (x[i]-y[i])
           *(x[i]-y[i]);
     // Note: x[i] is *(x+i*4)
     //  and  y[i] is *(y+i*4)
  }

  return sqrt(sum);
}
```
(a)

```
float Distance(
    float* x, float* y, int n){
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    sum += (*px-*py)
          *(*px-*py);
  }

  return sqrt(sum);
}
```
(b)

```
float Distance(
    float* x, float* y, int n){
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    register t = (*px-*py);
    sum += t*t;
  }

  return sqrt(sum);
}
```
(c)

Figure 6: Example demonstrating the allowed optimizations in an SC-preserving compiler. The function in (a) computes the distance between two n-dimensional points x and y represented as arrays. An SC-preserving compiler is able to safely perform a variety of optimizations, leading to the version in (b). However, it cannot eliminate the common-subexpression `*px - *py` involving possibly-shared accesses to the array elements. A traditional optimizing compiler does not have this restriction and is able to generate the version in (c).

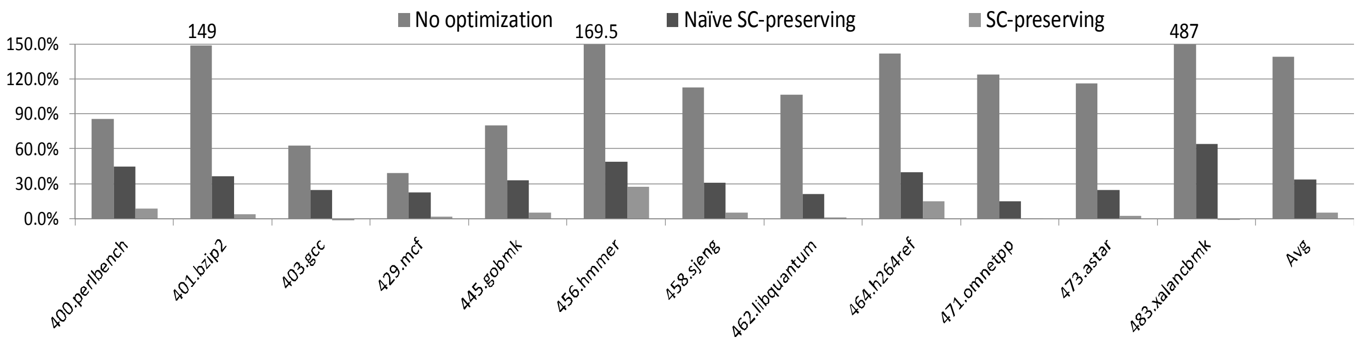

Figure 7: Performance overhead incurred by various compiler configurations compared to the stock LLVM compiler with -O3 optimization for SPEC CINT2006 benchmarks.

3.4 Evaluation

We evaluated our SC-preserving compiler on a variety of sequential and parallel benchmarks. Even though the topic of this paper only concerns multi-threaded programs, we included the sequential benchmarks in our evaluation as optimizing compilers are tuned to perform well for these benchmarks. Our experimental results indicate that the vast majority of the optimizations in LLVM responsible for good performance are in fact SC-preserving.

We executed all programs on an Intel Xeon machine with eight cores, each of which supports two hardware threads and 6 GB of RAM. We evaluated each program under three compiler configurations. The configuration "No optimization" is the stock LLVM compiler with all optimizations disabled; "Naïve SC-preserving" enables only those LLVM passes that are already SC-preserving, because they never reorder accesses to shared memory; and "SC-preserving" is our full SC-preserving compiler, which includes modified versions of some LLVM passes.

Figure 7 shows the results for the SPEC CINT2006 benchmarks. The figure shows the performance overhead of each benchmark under the three compiler configurations, normalized to the benchmark's performance after being compiled with the stock LLVM compiler and all optimizations enabled (-O3). With no optimizations, the benchmarks incur an average 140% slowdown. Re-enabling just the optimizations guaranteed to preserve SC reduces this overhead all the way to 34%. Our full SC-preserving compiler reduces the average overhead to only 5.5%, with a maximum overhead for any benchmark of 28%.

The results for parallel applications from the SPLASH-2 and PARSEC benchmark suites are shown in Figure 10 from Section 6 (the last two compiler configurations shown in the figure pertain to the notion of interference checks that we introduce in the next section). The results agree with those of the sequential benchmarks. Without optimizations the benchmarks incur an average 153% slowdown. Re-enabling "naïvely SC" optimizations reduces the overhead to 22%, and our full SC-preserving compiler incurs an average overhead of only 2.7%, with a maximum overhead for any benchmark of 34%.

4. Speculation for SC-Preservation

As shown in Table 1, most of the optimization passes that reorder shared memory accesses only relax the L $\rightarrow$ L and S $\rightarrow$ L orderings. In other words, these optimizations have the potential to perform eager loads but no other memory reorderings. In order to evaluate how important these eager-load optimizations are for performance, we fully enabled the four (SC-violating) eager-load IR optimization passes in our SC-preserving compiler and re-ran our parallel benchmarks. The "Only-Eager-Loads" configuration in Figure 10 illustrates the results. The benchmark with the largest overhead in our SC-preserving compiler, facesim, rebounded from a 34% slowdown to only a 6.5% slowdown, and many other benchmarks regained of all of their lost performance.

This experiment motivates our desire to *speculatively* perform eager-load optimizations and then dynamically recover upon a possible SC violation in order to preserve SC. This section describes how our compiler performs such speculation via a notion of *inter-*

```
                      DOM'
                      ORIG'
DOM                   i.chk monitoredAccesses, rcvr
ORIG      ⇒           jump cont
CONTINUE        rcvr: RECOVER
                cont: CONTINUE'
```

Figure 8: Introducing interference checks when performing eager-load transformations in `ORIG`, a single-entry, single-exit region of code with no stores. Either or both of `DOM'` and `ORIG'` contain the definitions for `monitoredAccesses` for the eager loads involved in the transformation.

ference checks, which conservatively determine whether a memory location's value has been modified since it was last read by the current thread. First we specify the instruction set architecture (ISA) extensions in the hardware that support interference checks. Then we show how a compiler can use these new instructions to speculatively perform eager-load optimizations, and we argue for the correctness of the approach.

4.1 ISA Extensions

Interference checks rely on three new instructions to be provided by the architecture: `m.load` (monitored load), `m.store` (monitored store), and `i.check` (interference check). The `m.load` and `m.store` instructions behave as regular loads and stores but additionally instruct the processor to start monitoring possible writes to the memory location being accessed. We assume that the processor can monitor up to a maximum of N locations simultaneously. These instructions therefore take as an additional parameter a tag from 0 to $N - 1$, which is used as an identifier for the memory location being monitored.

The `i.check` instruction provides a mechanism to query the hardware as to whether or not writes could have occurred to a set of memory locations. It accepts an N-bit mask and a recovery branch target as a parameter. The instruction conditionally branches to the recovery target based on whether or not writes may have occurred for any of the monitored memory addresses indicated by the mask. If the instruction does not branch, it is guaranteed that no thread has written to any of the locations indicated by the mask since the instructions that initiated their monitoring were executed. When using an `i.check` in the examples below, we will list the tags explicitly for clarity rather than using a bit mask.

Note that our use of tags to identify accesses, rather than simply identifying them with the address they access, allows the compiler to safely use interference checks in the face of potential aliasing. The compiler may end up monitoring two accesses to the same location using separate tags due to unknown aliasing. The hardware will correctly report interference between the time when the monitored access for each tag was executed and the time of the `i.check` for that tag. This design places the burden on the compiler to manage the resources available for monitoring. It must ensure that when it reuses a tag, the access that was previously assigned to that tag no longer needs to be monitored.

4.2 Interference Check Algorithm

Figure 8 illustrates how our compiler performs eager load optimizations with interference checks. Informally, the algorithm works on code in Static Single Assignment form (SSA) in the following steps:

1. Find a contiguous, single-entry, single-exit block of code without stores. Call this block `ORIG`.
2. Create a branch target at the first instruction after `ORIG`. Call the following instructions, starting at this new target, `CONTINUE`.
3. Make a copy of `ORIG` in its entirety and call it `RECOVER`. Note that, since we are manipulating SSA code, all local and temporary values will be given a new SSA name in the copied code.
4. Apply eager-load transformations in `ORIG` and call the resulting block of code `ORIG'`. The transformations may include any combination of the following:
   (a) Eliminate a load and replace its uses with a value from a previous load or store to that address that dominates the current load. This prior memory access may or may not be in `ORIG`. Convert this previous memory access to an `m.load` or `m.store` if it is not already one. If multiple definitions reach the load to be removed, all of them have to be converted.
   (b) Hoist a load from `ORIG` to a position dominating all of its uses, potentially reordering with previous load and/or store operations. Its new position may or may not be in `ORIG`. Convert the hoisted load to an `m.load`.

   We'll call the code that dominated `ORIG` and may now contain monitored instructions `DOM'`. Each access that is converted to a monitored instruction must use a distinct tag, so the compiler is limited to at most N eager-load conversions in this step.
5. Perform any desired SC-preserving optimizations on the code remaining in `ORIG'`.
6. Insert an `i.check` instruction after `ORIG'` that checks for interference on all accesses that were marked as monitored by step 4 and branches to the recovery code on failure.
7. For all values that are live-out of `ORIG`, transform `CONTINUE` by inserting an SSA phi instruction at the beginning choosing the appropriate value based on whether code flowed from `ORIG'` or `RECOVER`. Call the transformed block `CONTINUE'`.

4.3 Implementation and Example

We modified LLVM's global value numbering (GVN) pass to make use of interference checks in order to allow more aggressive optimization while maintaining SC. The GVN pass performs a variety of eager-load optimizations, including CSE, partial redundancy elimination, and copy/constant propagation. Due to time limitations, we have not implemented the algorithm on other passes.

Figure 9 shows some LLVM IR code that calculates $\mathtt{X}^2 + \mathtt{Y} + \mathtt{X}^2$, along with the transformations that take place on it during the GVN pass in order to eliminate the redundant computation of $\mathtt{X}^2$. Virtual registers, or temporaries, are prefixed by the `%` symbol and are in SSA form. First, the GVN pass removes the second load of memory location `X` (which defines `%5`) and replaces all of its uses with the first load of `X`. After this load elimination, we are left with the code in (b), where it is clear that the second `mul` instruction is unnecessary, so it is removed and its use is replaced with the previously calculated value in virtual register `%2`. The final code with the load and multiply eliminated is shown in (c). Figure 9(d) shows how our algorithm adds interference checks to make this transformation SC-preserving.

4.4 Correctness of the Algorithm

We now argue that our algorithm for inserting interference checks is SC-preserving. First consider the case when the interference check fails. Neither `ORIG` nor `ORIG'` contains any stores. Thus, the state of non-local memory does not change during the execution of `ORIG'`. As the code is in SSA form, all the effects of `ORIG'` on local state become dead once the code switches to `RECOVER`, which is a copy of `ORIG`. Hence, other than needlessly executing `ORIG'`, the transformed program has the same behavior as the original program when the interference check fails.

Now consider the case when the interference check succeeds. This means that each monitored memory location is guaranteed to

Original

```
// DOM
%1 = load X
%2 = mul %1, %1
%3 = load Y
%4 = add %2, %3

//ORIG
%5 = load X
%6 = mul %5, %5

//CONTINUE
%7 = add %4, %6
```

(a)

⇒

Load Eliminated

```
// DOM
%1 = load X
%2 = mul %1, %1
%3 = load Y
%4 = add %2, %3

// ORIG'
%6 = mul %1, %1

// CONTINUE
%7 = add %4, %6
```

(b)

⇒

CSE

```
// DOM
%1 = load X
%2 = mul %1, %1
%3 = load Y
%4 = add %2, %3

// ORIG'

// CONTINUE
%7 = add %4, %2
```

(c)

SC with `i.check`

```
// DOM'
%1 = m.load X, 0
%2 = mul %1, %1
%3 = load Y
%4 = add %2, %3

// ORIG'
i.check 0, rcvr
jump cont

// RECOVER
rcvr:
%5 = load X
%6 = mul %5, %5

// CONTINUE
cont:
%merge = phi (orig, %2, rcvr, %6)
%7 = add %4, %merge
```

(d)

Figure 9: GVN first transforms program (a) into (b) by eliminating the "available load" from `X`, then notices that the result of the second multiplication has already been computed and performs common subexpression elimination to arrive at (c). This transformation is not SC since it reorders the second load of `X` with the load of `Y`.

be unmodified from the start of monitoring through the execution of `ORIG'`. The key property of our algorithm is that every memory location involved in an eager load is monitored from the point where the eager load occurs until at least the point at which the load would have occurred in the original program (since it would have occurred somewhere within `ORIG`). Thus the value loaded in the optimized code is the value that would have been read by the original program, thereby preserving SC.

5. Hardware Support for Interference Checks

In this section we describe hardware support for efficiently implementing the `m.load`, `m.store`, and `i.check` instructions described in the previous section. The hardware changes required are simple and efficient, and therefore practical. In fact, the new instructions we propose are similar to the data speculation support in the Itanium's ISA [29], which was designed to enable speculative optimizations in a single thread in the face of possible memory aliasing. Our design safely supports both our goal (to ensure sequential consistency) as well as Itanium's speculative load optimizations. Our required hardware support is simple: a structure to store N addresses (32 in our implementation), each with an associated bit indicating whether the address was possibly written.

5.1 Hardware Design

We propose a hardware structure called the Speculative Memory Address Table (SMAT) which is similar to the Advanced Load Address Table (ALAT) used in Itanium processors [29]. SMAT is a Content-Addressable-Memory (CAM). It has N entries, enabling the compiler to monitor interference on N addresses at any instant of time. Each entry in the SMAT contains an address field and an interference bit.

We collectively refer to `m.load` and `m.store` instructions as *monitor* instructions. As described in the previous section, each monitor instruction contains a tag between 0 and $N-1$. When executing a monitor instruction, the hardware stores the address accessed by that instruction in the SMAT entry specified by the tag, resets that entry's interference bit, starts to monitor writes to the address, and executes the memory operation requested by the instruction.

A processor core can easily detect when another processor core wants to write an address by monitoring invalidation coherence requests. When a processor core receives an invalidation to a cache block, the interference bit of each SMAT entry holding an address from that block is set. The interference bit of an entry is also set when a store to the associated address commits from the current processor. While the latter behavior is not necessary to preserve SC, it enables Itanium-style speculative load optimizations [29].

The compiler generates an `i.check` instruction with an N-bit `mask` to check for interference on a maximum of N different addresses. Each bit in the `mask` corresponds to an entry in the SMAT. The hardware executes the `i.check` instruction by checking the interference bits of the SMAT entries specified in its `mask`. If any of the checked interference bits is set, the hardware branches to the recovery code whose target is specified in the `i.check` instruction.

The hardware updates the SMAT for a monitor instruction and executes `i.check` instructions only when they are ready to commit from a processor core's instruction window. This ensures that the hardware does not update SMAT entries speculatively while executing instructions on an incorrect path taken due to branch misprediction. The next section explains a subtlety in implementing the monitor instructions in out-of-order processors.

5.2 Relation To In-Window Hardware Speculation

Our approach of monitoring invalidation coherence requests to detect interference for a set of addresses is similar to what many processors already implement for efficiently supporting TSO at the hardware level [23]. TSO does not allow a load to be executed before another load in program order even if they are accessing different addresses. To achieve good performance, Gharachorloo et al. [23] proposed to speculatively execute loads out-of-order. However, instructions are still committed in order from a FIFO queue called the reorder buffer (ROB). Therefore, to detect misspeculation the hardware simply needs to detect when another processor tries to write to an address that has been read by a load that is yet to commit from the ROB. This is achieved by monitoring the address of invalidation coherence requests from other processor cores. On detecting a misspeculation, the hardware flushes the misspeculated load and its following instructions from the pipeline and restarts execution.

Our proposed hardware design essentially extends the above hardware mechanism to detect interference for addresses of certain memory operations (specified by the compiler) even after they are committed from the ROB. This allows our compiler to eagerly execute loads and later check for interference at the original location of the load in the source code. On a `m.load`, the monitoring needs to start logically when the processor receives the value of the load. However, the SMAT entry is updated only when the instruction is committed. In between the two events, when the load instruction is in flight in the ROB, we rely on the monitoring performed above to provide the required semantics of the `i.check`.

5.3 Conservative Interference Checks

While an implementation of interference checks must detect interference whenever it occurs, it is legal to signal interference when none actually exists. Such false positives are acceptable in our design because they simply result in execution of the unoptimized code, losing some performance but maintaining SC. The ability to tolerate false positives allows us to avoid a number of potentially complex issues and keep the hardware simple.

First, our hardware monitors interference at the cache block granularity as coherence invalidation messages operate at cache block level. This may result in false positives when compared to a detector that monitors byte-level access. But the probability that a cache block gets invalidated between a monitor instruction and an `i.check` is very low. Moreover, frequent invalidations or "false sharing" of hot cache lines result in performance degradations and thus can expected to be rare in well-tuned applications.

Second, we conservatively invalidate SMAT entries for a cache block that gets evicted due to capacity constraints. Monitoring interference for uncached blocks would require significant system support (similar in complexity to unbounded transactional memory systems [17]), but we believe it is not necessary for performance.

Third, in ISAs like `x86` one memory instruction could potentially access two or more cache lines, but our SMAT entry can monitor only one cache block address. To address this problem, if a monitor instruction accesses more than one cache block we immediately set the interference bit for the SMAT entry that monitors the associated address, which could cause a future `i.check` to fail forcing execution down an unoptimized path. Fortunately, such unaligned cache accesses are rare.

Finally, a context switch may occur while multiple addresses are monitored in the hardware SMAT. Instead of virtualizing this structure, we propose to set the interference bit in all SMAT entries after a context switch. This may cause future `i.check` instructions from the same thread to fail unnecessarily when it is context switched back in, but we expect this overhead to be negligible as context switches are relatively rare when compared to the frequency of memory accesses.

6. Results

The experimental results relating to the performance of our SC-preserving compiler were discussed in Section 3.4. In this section we discuss additional experiments which evaluate the potential effectiveness of our interference checks. In addition, we compare the performance of our SC-preserving compilers to a fully optimizing compiler running on simulated hardware that uses a DRF0 memory model which is more relaxed (allows more hardware reorderings) than TSO. This gives a sense of the performance burden of providing a strong, end-to-end memory model across hardware and software.

6.1 Compiler Configurations

As described in Section 3.4, our baseline compiler is the out-of-the-box LLVM compiler with all optimizations (`-O3`). For our experiments on parallel benchmarks, we used the three compiler configurations discussed in that section ("No optimization", "Naïve SC-preserving", and "SC-preserving"), as well as two additional configurations. The "Only Eager Loads" configuration includes all the optimizations from the SC-preserving compiler plus the unmodified (SC-violating) version of all IR passes that perform only eager load optimizations (GVN, instcombine, argpromotion, and jump-threading). This configuration is intended to give a sense of the opportunity for improvement available to optimizations based on our interference check technique and is only used for experiments on native hardware and not on simulated machines. Finally, the "SC-preserving+GVN w/ ICheck" configuration includes all of the optimizations from the SC-preserving compiler plus a modified GVN pass that is made SC-preserving using our interference checks and recovery code. When this configuration targets a simulated machine with appropriate support, it emits `m.load`, `m.store`, and `i.check` instructions. But when it targets native hardware, the configuration emits `m.load` and `m.store` instructions as regular loads and stores and emulates a never-failing `i.check` using a logical comparison of constant values followed by a conditional branch. Thus, when running on the native machine, the overhead caused by increased code size and the additional branch is captured, but the effect of actual or false conflicts on monitored accesses is not. In a real implementation, however, we expect the `i.check` instruction to be more efficient than a branch.

Table 2: Baseline IPC for simulated DRF0 hardware running binaries from the stock LLVM compiler.

Application	Avg. IPC	Application	Avg. IPC
blackscholes	1.94	bodytrack	1.61
fluidanimate	1.28	swaptions	1.67
streamcluster	1.42	barnes	1.57
water(nsquared)	1.66	water(spatial)	1.66
cholesky	1.78	fft	1.39
lu(contiguous blocks)	1.64	radix	0.99

6.2 Benchmarks

We evaluated the performance of the various compiler configurations on the PARSEC [7] and SPLASH-2 [49] parallel benchmark suites. Table 2 lists the average instructions executed per cycle (IPC) for each of these benchmarks when compiled with the stock LLVM compiler at `-O3` optimization and run on our simulated DRF0 hardware which implements weak consistency and is described below. All of these benchmarks are run to completion. For our experiments on actual hardware, we used the `native` input for PARSEC benchmarks, while for the simulated machines we used the `sim-medium` input set to keep the simulation time reasonable. (Since `streamcluster` was especially slow to simulate, we used the `sim-small` input.) For SPLASH-2 applications, we used the default inputs for simulation. We modified the inputs to increase the problem size for experiments on native hardware. We verified the correct behavior of the benchmarks under all compiler configurations by using a self-testing option when available, or by comparing results with those produced when compiling the benchmark using gcc.

6.3 Experiments on Native Hardware

We evaluated all six compiler configurations (including the baseline) on an Intel Xeon machine with eight cores each of which supports two hardware threads and 6 GB of RAM. Each benchmark was run five times for each compiler configuration and the execution time was measured. (The results given here are for CPU user time, though the results for total time elapsed were very similar.) The overheads given are relative to the baseline, fully-optimizing compiler and are shown in Figure 10. Let's consider the base SC-preserving compiler

Table 3: Processor Configuration

Processor	4 core CMP. Each core operating at 2Ghz.
Fetch/Exec/ Commit width	4 instructions(maximum 2 loads or 1 store) per cycle in each core.
Store Buffer	TSO: 64 entry FIFO buffer with 8 byte granularity. DRF0, DRFx: 8 entry unordered coalescing buffer with 64 byte granularity.
L1 Cache	64 KB per-core (private), 4-way set associative, 64B block size, 1-cycle hit latency, write-back.
L2 Cache	1MB private, 4-way set associative, 64B block size, 10-cycle hit latency.
Coherence	MOESI directory protocol
Interconnection	Hierarchical switch, 10 cycle hop latency.
Memory	80 cycle DRAM lookup latency.
SMAT	32 entries CAM structure, 1 cycle associative lookup

first. Notice that for many of our benchmarks, restricting the compiler to perform only SC-preserving optimizations has little or no effect. In fact, in some cases, disabling these transformations appears to speed the code up, indicating that the compiler ought not to have performed them in the first place. There are several benchmarks, however, for which the SC-preserving compiler incurs a noticeable performance penalty, 34% in the case of `facesim`.[2] On average, we see a 2.7% slowdown. Consider now the compiler configuration which re-enables various eager load optimizations. Several of the applications which suffered a significant slowdown under the SC-preserving compiler regain much of this performance in this configuration. Most notably, `facesim` vastly improves to 6.5% and `bodytrack`, `streamcluster`, and `x264` recover all (or nearly all) of their lost performance. On average, the compiler with eager load relaxations enabled is as fast as the stock compiler, indicating that our technique of using interference checks to safely allow eager load optimizations holds significant promise. Finally, the rightmost bar in the graph shows the slowdown of the aggressive SC-preserving compiler that includes the modified GVN pass with interference checks. (Remember, we are running on a native machine in this set of experiments, so a never-fail load check is emulated.) We see that this technique regains a good portion of the performance lost by the base SC-preserving compiler for `facesim`, reducing the overhead from 34% to under 20%, with `streamcluster` and `x264` showing a more modest improvement.

6.4 Experiments on Simulated Machines

To study the performance of interference checks in hardware, we used a cycle-accurate, execution driven, Simics [35] based x86_64 simulator called FeS2 [19]. We simulated TSO hardware with and without support for interference checks and compared it to DRF0 hardware that supports weak consistency. The processor configuration that we modelled is shown in Table 3. For the TSO simulation, we modelled a FIFO store buffer that holds pending stores and retires them in-order. We also modelled speculative load execution support [23]. The weakly consistent DRF0 simulation allowed stores and loads to retire out-of-order.

[2] Additional profiling and investigation revealed that the slowdown in `facesim` was largely caused by a commonly invoked 3x3 matrix multiply routine. The SC-preserving compiler was unable to eliminate the two redundant loads of each of the 18 shared, floating point matrix entries involved in the calculation. This resulted in 36 additional load instructions for each matrix multiplication performed by the SC-preserving version of `facesim`. Our GVN pass with interference checks is able to relegate the 36 additional loads to the recovery code, eliminating them on the fast path. A straightforward rewrite of the source code to first read the 18 shared values into local variables would have allowed the base SC-preserving compiler to generate the fully optimized code.

Figure 11 shows the results of our simulation study. When compared to the fully optimizing compiler configuration running on the simulated DRF0 machine, the performance overhead of using our SC-preserving compiler on simulated TSO hardware is 3.4% on average. This cost is reduced to 2.2% when the GVN pass with interference checks is used. For several programs that incur significant overhead, such as `bodytrack` and `facesim`, our interference check optimizations reduce the overhead to almost zero. For `streamcluster`, the overhead is reduced from about 23% to 17%. We also found that the frequency of load-check failures is, on average, only about one in ten million instructions. This indicates that the performance overhead due to false positives arising from several hardware simplifications described in Section 5.3 is negligible.

7. Discussion and Related Work

7.1 Relationship to Data-Race-Free Memory Models

Today's mainstream concurrent programming languages including C++ and Java use variants of the *data-race-free* memory model known as DRF0 [2, 3]. These memory models are rooted in the observation that good programming discipline requires programmers to protect shared accesses to data with appropriate synchronization (such as locks). For such data-race-free programs, these models guarantee SC. For programs with data races, however, these models guarantee either no semantics [11] or a weak and complex semantics [36]. The compiler and hardware are restricted in the optimizations allowed across synchronization accesses, but they can safely perform most sequentially-valid optimizations on regular memory accesses.

Our work is primarily motivated by the need to provide understandable semantics to racy programs [1] both for debugging ease of large software systems that are likely to have data races and for guaranteeing security properties in safe languages such as Java [36].

In the context of a DRF0-compliant compiler (such as LLVM), our interference checks can be seen as a specialized form of data race detection. Our approach allows the compiler to speculatively perform the eager-load optimizations which would be allowed in the DRF0 model. However, rather than silently providing undefined or weak semantics upon a data race, our interference checks dynamically detect the race and recover in order to preserve SC. In this way, data-race-free programs can be aggressively optimized and incur just the additional cost of the interference check itself and the rare false interference check that may occur (Section 5.3). Furthermore, even racy programs benefit from our approach, because the interference check captures exactly the requirement necessary to justify an optimization's SC-preservation. For instance, data races on variables that are not eagerly loaded, as well as data races on eagerly-loaded variables that occur outside of the scope of that reordering, cannot violate SC preservation and so need not be detected.

While this paper focuses on eager-load optimizations, we hope to explore in future work speculation mechanisms that enable other DRF0-compliant optimizations while preserving SC.

7.2 A TSO-Preserving Compiler

Providing SC semantics to the programmer requires that the output of an SC-preserving compiler be executed on SC hardware. While most hardware platforms today support a relaxed memory model, popular platforms such as Intel's x86 and Sun's SPARC platforms provide a relatively-strong memory model known as total store ordering (TSO). Conceptually, TSO can be seen as a relaxation of SC that allows $\mathtt{S} \rightarrow \mathtt{L}$ reorderings [2] and has a precise understandable operational semantics [40].

Our approach to developing an SC-preserving compiler can be naturally adapted to instead preserve TSO, thereby providing end-to-end TSO semantics on existing TSO hardware. Transformations

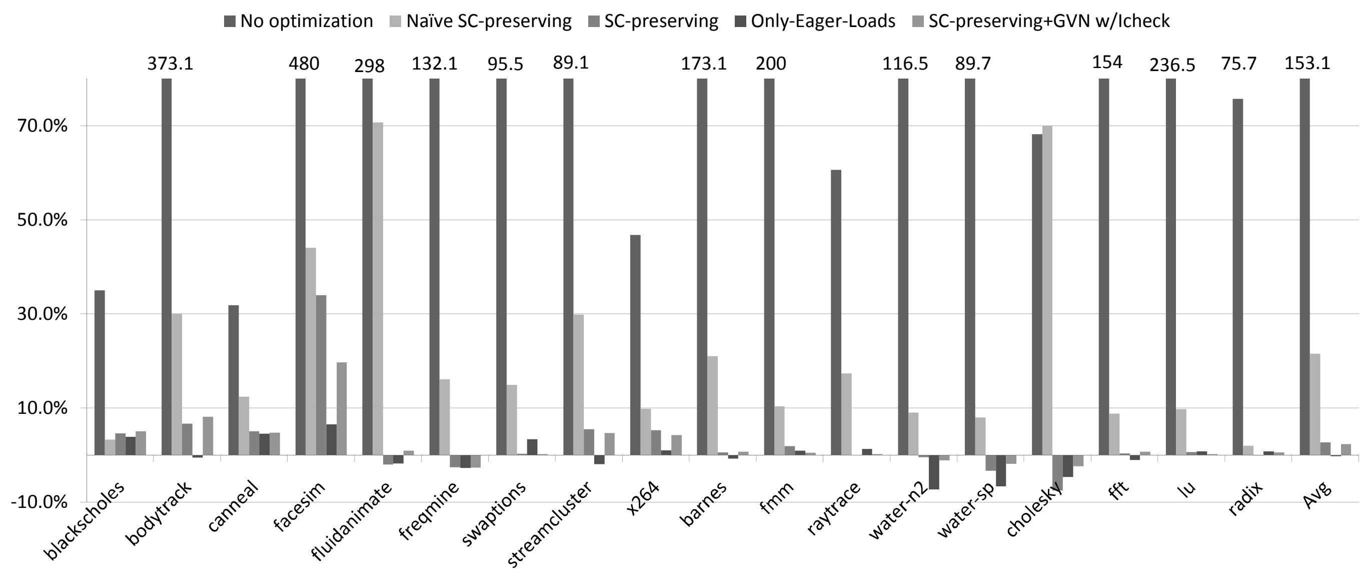

Figure 10: Performance overhead incurred by the various compiler configurations compared to stock LLVM compiler with `-O3` optimization running on native Xeon hardware for PARSEC and SPLASH-2 benchmarks.

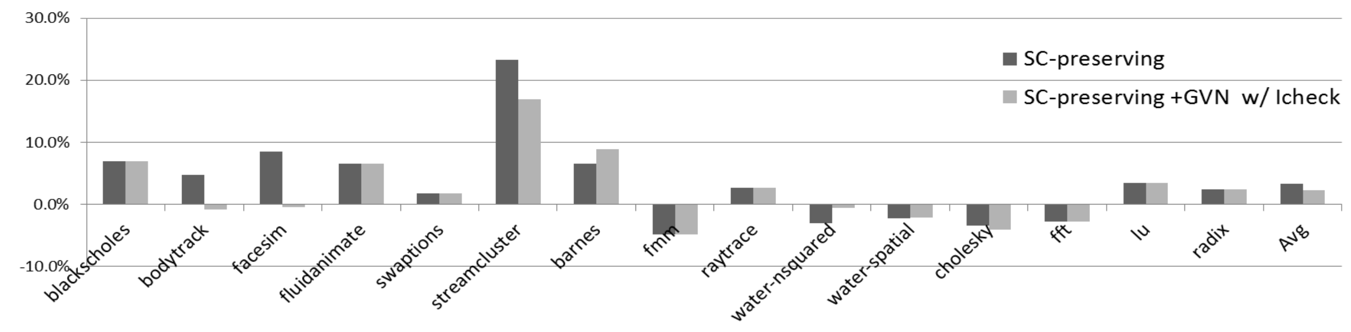

Figure 11: Performance overhead of SC-preserving compiler on simulated TSO hardware with and without using interference checks relative to fully optimizing SC-violating compiler on simulated DRF0 hardware.

that do not reorder memory accesses and those that only reorder thread-local variables are TSO-preserving [48]. In addition, all of the SC-preserving transformations shown in Figure 3 also preserve TSO except for redundant store elimination [14].

In the context of a DRF0-compliant compiler, the result of this approach would be a language-level memory model that provides SC semantics for data-race-free programs and TSO semantics for racy programs. This variation on DRF0 is significantly stronger and much simpler than both the C++0x [6, 11] and Java [36] memory models.

In recent work, Ševčík et al. describe a concurrency extension to a small C-like programming language that provides end-to-end TSO semantics [48]. They modify an existing compiler for the language and mechanically prove that the optimizations are TSO-preserving, thereby providing an end-to-end guarantee when the resulting binaries are executed on x86 hardware. Our performance measurements complement their work by indicating that a TSO-preserving compiler could be practical to use in a full-fledged programming language.

7.3 Dynamic Detection of Data Races and SC Violations

Others have argued for the use of dynamic race detection to improve the DRF0 memory model [10, 16, 18], in order to halt an execution when its semantics becomes undefined, analogous with Java's fail-stop approach to preventing array-bounds violations and null-pointer dereferences. However, detecting data races either incurs 8x or more performance overhead in software [20] or incurs significant hardware complexity [4, 38, 43] despite many proposed optimizations to the basic technique.

Recently, Marino et al. [37] and Lucia et al. [34] observed that it suffices to dynamically detect SC violations rather than races, and that this can be done much more efficiently. In their approaches, the compiler partitions a program into regions and may perform most sequentially valid optimizations within a region but cannot optimize across regions. Region boundaries are communicated to the hardware as memory fences, thereby also preventing hardware optimizations across regions. Given these requirements, the hardware can conservatively identify data races that potentially cause SC violations by detecting conflicting accesses in concurrently executing regions, similar to the conflict detection performed by transactional memory (TM) systems [27].

Our interference checks are a form of dynamic data-race detection that is sufficient to ensure SC-preservation of compiler transformations. Such detection provides a weaker guarantee than the above approaches, which additionally detect SC violations due to hardware reorderings. However, our approach has a number of advantages. First, our detection scheme is fine-grained, requiring only data race detection for variables that are involved in a compiler optimization and only during the dynamic lifetime of that optimization's effect. Second, we can perform this detection with relatively minimal hardware support based on existing hardware speculation mechanisms [50], rather than requiring the complexity of TM-style conflict detection. Finally, we show how to safely recover from interference for common compiler optimizations based on eager loads, thereby allowing the execution to safely continue while maintaining SC.

7.4 Optimistic Optimization via Hardware Speculation

Our interference checks are inspired by a common hardware mechanism for enabling out-of-order execution in the presence of strong memory models [23]. This mechanism [50] allows a memory load to be executed out-of-order speculatively, before earlier instructions have completed. Once those instructions have completed, the load need not be re-executed if the value has not changed in the meanwhile, and this can be conservatively detected by checking if the associated cache line has been invalidated. We illustrate how this technique can be adapted to the compiler by viewing common compiler optimizations as performing eager (i.e., speculative) reads, and we describe a simple way for the hardware to expose this mechanism to the compiler.

Others have proposed hardware support for dynamically detecting memory aliasing between local loads and stores in a single thread and expose that feature to the compiler so that it can perform optimistic optimizations [22, 41]. The Itanium processor implemented this feature using an Advanced Load Address Table (ALAT) to enable aggressive load optimizations [29]. Recently, Nagarajan and Gupta [39] extended Itanium's ALAT mechanism to detect memory aliasing with remote writes, enabling the compiler to speculatively reorder memory operations across memory barriers. While our hardware mechanism to detect memory aliasing is similar to these proposals, we apply it to solve a different problem: preserving SC in the face of common compiler transformations.

7.5 Guaranteeing End-to-End Sequential Consistency

Our approach ensures that the compiler is SC-preserving but does not prevent the hardware from exposing non-SC behavior. We could augment our compiler to address this problem by inserting memory fences to prevent hardware reorderings that potentially violate SC. Such fences cause a significant performance penalty, so there has been research in minimizing the number of fences required. Shasha and Snir proposed the *delay sets* algorithm for determining the set of fences to insert [45]. Recent research has further reduced the number of fences required by incorporating analyses that detect which memory locations are possibly accessed by multiple threads [30, 46]. Finally, recent work describes a new hardware mechanism called a *conditional fence* [33], which uses the results of a compiler analysis to dynamically decide whether a given fence in the instruction stream can be safely ignored while still ensuring SC. All of these approaches rely critically on whole-program analyses to obtain sufficient precision.

A different way to ensure SC at the language level is by statically rejecting possibly racy programs. Several static type systems have been proposed that prevent races (e.g., [12, 13, 21]) and ensure stronger properties such as determinism [9]. By rejecting potentially racy programs, these type systems ensure that all program executions have SC semantics. However, these type systems enforce a restricted programming style that is necessarily conservative. For example, many static type systems for race detection only account for lock-based synchronization and will reject race-free programs that use other synchronization mechanisms. Further, even correct programs that employ locks can be rejected due to imprecise information about pointer aliasing. More precision in static race detection can be achieved through whole-program analysis [42].

Hammond et al. [25] proposed the transactional coherency and consistency (TCC) memory model. The programmer and the compiler ensure that every instruction is part of some transaction. The runtime uses transactional memory [27] to ensure serializability of transactions, which in turn guarantees SC at the language level. The Bulk compiler [5] and the BulkSC hardware [15] together also guarantee SC at the language level. The bulk compiler partitions a program into "chunks" and the BulkSC hardware employs speculation and recovery to ensure serializable execution of chunks. Conflicts are resolved through rollback and re-execution of chunks. These techniques obtain a strong guarantee of SC, but at the cost of significant hardware extensions that are similar to transactional memory support.

8. Conclusions

A memory model forms the foundation of shared-memory multithreaded programming languages. This paper empirically demonstrates that the performance incentive for relaxing the intuitive SC semantics in the compiler is much less than previously assumed. In particular, this paper describes how to engineer an SC-preserving compiler through simple modifications to LLVM, a state-of-the-art C/C++ compiler. For a wide range of programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites, our SC-preserving compiler results in a performance overhead of only 3.8% on average with a maximum of 34% overhead.

While the overheads, however small, might be unacceptable for certain applications, this paper argues that other avenues for improving the performance of SC-preserving compilers should be explored before resorting to relaxing the program semantics. We proposed a novel hardware-software cooperation mechanism in the form of interference checks, which enabled us to regain much of the performance lost due to restrictions imposed on compiler optimizations to preserve SC.

9. Acknowledgements

The authors would like to thank Sarita Adve, Vikram Adve, Hans Boehm, Sebastian Burckhardt, Mark Hill, Suresh Jagannathan, Milo Martin, Todd Mytkowicz, and Jaroslav Ševčík for insightful discussions regarding this work. We also like to thank the anonymous reviewers for valuable feedback on this paper. 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] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, 2010.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[3] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of ISCA*, pages 2–14. ACM, 1990.

[4] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, pages 234–243, 1991.

[5] 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 *42nd International Symposium on Microarchitecture*, 2009.

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

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

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

[9] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[10] H. J. Boehm. Simple thread semantics require race detection. In *FIT session at PLDI*, 2009.

[11] H. J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of PLDI*, pages 68–78. ACM, 2008.

[12] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of OOPSLA*, pages 56–69. ACM Press, 2001.

[13] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of OOPSLA*, 2002.

[14] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin / Heidelberg, 2010.

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

[16] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. The case for system support for concurrency exceptions. In *USENIX HotPar*, 2009.

[17] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[18] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

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

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

[21] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of PLDI*, pages 219–232, 2000.

[22] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *ASPLOS*, pages 183–193, 1994.

[23] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 1, pages 355–364, 1991.

[24] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *IEEE PACT*, pages 179–188, 2002.

[25] 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*, pages 102–113, 2004.

[26] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34:1–17, September 2006. ISSN 0163-5964.

[27] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of ISCA*, pages 289–300. ACM, 1993.

[28] M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31:28–34, 1998. ISSN 0018-9162.

[29] Itanium. Inside the Intel Itanium 2 processor. *Hewlett Packard Technical White Paper*, 2002.

[30] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 15. IEEE Computer Society, 2005.

[31] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(28):690–691, 1979.

[32] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2004.

[33] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *International Conference on Parallel Architectres and Compilation Techniques*, 2010.

[34] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict Exceptions: Providing simple parallel language semantics with precise hardware exceptions. In *37th Annual International Symposium on Computer Architecture*, June 2010.

[35] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[36] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of POPL*, pages 378–391. ACM, 2005.

[37] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI '10*, pages 351–362. ACM, 2010.

[38] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *ISCA*, 2009.

[39] V. Nagarajan and R. Gupta. Speculative optimizations for parallel programs on multicores. In *LCPC*, pages 323–337, 2009.

[40] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *In TPHOLs '09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*, pages 391–407. Springer, 2009.

[41] M. Postiff, D. Greene, and T. N. Mudge. The store-load address table and speculative register promotion. In *MICRO*, pages 235–244, 2000.

[42] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of PLDI*, pages 320–331, 2006.

[43] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of ISCA*, San Diego, CA, June 2003.

[44] 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 '97*, pages 199–210, 1997.

[45] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, 1988.

[46] Z. Sura, X. Fang, C. Wong, S. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of PPoPP*, pages 2–13, 2005.

[47] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, pages 27–51, 2008.

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

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

[50] K. Yeager. The MIPS R10000 superscalar microprocessor. *Micro, IEEE*, 16(2):28–41, 2002. ISSN 0272-1732.