A Safety-First Approach to Memory Models

Recent efforts to standardize concurrency semantics for programming languages assume memory accesses are data-race-free ("safe") by default and require explicit annotations on data races ("unsafe" accesses). Simple programming mistakes can result in counterintuitive program semantics. Instead, the authors argue for and demonstrate a safety-first approach that treats every memory access as potentially unsafe unless proven otherwise.

Abhayendra Singh
Satish Narayanasamy
University of Michigan

Daniel Marino
Symantec Research Labs

Todd Millstein
University of California

Madanlal Musuvathi
Microsoft Research

A *safe* programming language is one that protects its own abstractions.[1] Safe languages provide strong guarantees to programmers for all programs that are allowed to execute, obviating large classes of subtle and dangerous errors and cleanly separating a language's interface to programmers from its implementation details. Modern languages and programmers have embraced the compelling programmability benefits of safety despite the additional runtime overhead. Specifically, memory- and type-safe languages such as Java, C#, Python, and JavaScript protect the abstraction of memory as a collection of disjoint entities, each with a well-defined structure and set of operations based on its type.

Unfortunately, recent work to standardize concurrency semantics in mainstream programming languages is reversing this trend, providing multithreading support that subverts fundamental programming language abstractions, exposes the complexities of compiler and hardware optimizations to programmers, and makes it easy for programmers to shoot themselves in the foot in ways that are difficult to detect and correct. This is the case not only for "unsafe" languages such as C and C++ but also for "safe" languages like Java.

Consider the simple C++ program in Figure 1. One thread creates an object and publishes it to another thread by setting the `init` variable. Most programmers would expect statement `D` to correctly dereference the object allocated in statement `A`. This expectation is borne out of the fact that we usually take two programming abstractions for granted when analyzing a program snippet: *program order*, which requires that instructions in a thread execute one after the other in the order they appear in the program text; and *shared memory*, which requires the memory to behave as a map from addresses to values with each memory operation taking effect immediately. The memory model that preserves this intuitive program behavior for multithreaded programs is *sequential consistency* (SC).[2]

Contrary to intuition, statement `D` can trigger a null-pointer exception, because C++ and other mainstream languages[3,4]

Published by the IEEE Computer Society

provide weaker semantics known as data-race-free-0 (DRF0). DRF0 guarantees SC only if all the data races in the program are explicitly annotated. The accesses to `init` in the program in Figure 1 form a data race. Failure to annotate this data race can expose the effects of sequentially valid (that is, correct when considered on a single thread in isolation) compiler and hardware transformations. For instance, in the absence of an annotation on the `init` variable, the compiler and hardware can freely reorder statements `A` and `B`, because there is no data or control dependence between them, but this transformation can result in a null-pointer exception at statement `D`. Such counterintuitive and implementation-dependent behavior can significantly compromise a program's safety, correctness, and debuggability.

We argue instead for a safety-first approach to memory models, where the compiler and hardware conservatively treat every memory access as *unsafe*—that is, possibly participating in a data race. Aggressive optimizations are enabled only on *safe* accesses that the compiler and hardware can prove are data-race-free through static or dynamic analysis. In this way, SC semantics is guaranteed for all programs, whether data-race-free or not.

Contrary to popular belief, we show in our ISCA 2012 paper that we can provide SC hardware without incurring significant performance or even design costs,[5] leveraging the fact that most memory accesses can be proven to be safe. Our SC hardware achieves high efficiency by simply relaxing all the memory-model constraints on safe accesses, obviating the need for the complex out-of-window speculation techniques previously proposed for supporting SC in hardware.[6,7] (For more information, see the "Related Work in Sequential Consistency" sidebar.) In addition to the first-in, first-out (FIFO) store buffer for unsafe stores, our design employs an unordered store buffer that fast-tracks safe stores and allows later memory accesses to proceed without a memory-ordering-related stall.

When combined with our earlier work on efficient SC-preserving compilation,[8] the result is an end-to-end SC guarantee at the language level with low overhead. Our experimental study on several scientific and server benchmarks written in C shows that the overhead of our SC hardware over total store order (TSO) is less than 2 percent on average. We also found that the overhead of providing end-to-end SC (running our SC-preserving compiler's output on our SC hardware) when compared to running the stock LLVM C compiler's output on TSO hardware is about 6 percent on average.

```
X* x = null;
bool init = false;

// Thread u          // Thread v
A: x = new X();      C: if(init)
B: init = true;      D:   x->f++;
```

Figure 1. Incorrect program under the C++ memory model. Under DRF0, the compiler or hardware can freely optimize unannotated memory accesses `A` and `B` which could cause `D` to raise a null-pointer exception.

Memory-access-type-driven SC hardware

Past SC hardware designs have uniformly enforced memory-model constraints on all nonsynchronization memory accesses, distinguishing only between stores and loads. This is overly conservative and unnecessary for a significant fraction of memory accesses.

We leverage the simple observation that memory-model constraints need not be enforced for private locations and shared read-only locations.[9,10] Because most memory accesses are to private or read-only data,[11,12] this observation provides an opportunity to design efficient SC hardware by simply relaxing the ordering constraints on many memory accesses, obviating the need for complex speculation techniques.[6,7]

If either the compiler or the runtime system (or perhaps an expert programmer through annotations) can guarantee the absence of data races on a particular memory access, then the processor can safely reorder that access in any manner that preserves intrathread data dependencies. We refer to memory accesses with this property as *safe accesses* and the rest as *unsafe accesses*.

Related Work in Sequential Consistency

Past research has proposed aggressive speculation techniques to allow store-buffer optimization in sequential consistency (SC) hardware.[1,2] These designs extend the idea of in-window speculation to speculatively commit loads from the reorder buffer (ROB) even when the store buffer is not empty. This requires fairly complex hardware for checkpointing the program state before each load commit, detecting potential SC violations, and performing a rollback on detecting a potential SC violation. Although Lin et al. eliminated the need for checkpoint and rollback support,[3] their design still requires significant changes to the coherence protocol to efficiently perform conflict detection before committing a memory instruction from the ROB.

Region-based solutions like TCC[4] and BulkCompiler[5] guarantee end-to-end SC by partitioning programs into regions and ensuring that regions are both executed atomically and serializable in any execution. Regions can be formed either by programmers (transactions in TCC) or by the compiler (for example, BulkCompiler). These solutions rely on fairly expensive speculation hardware (checkpointing, versioning, conflict detection, and recovery) to guarantee serializability of regions.

Researchers have also proposed software-only solutions that use static analysis for guaranteeing SC on hardware supporting weaker memory models. Shasha and Snir proposed delay set analysis,[6] which finds the minimum number of fences required for an SC execution. To reduce the overhead due to fences, Sura et al.[7] and Kamil, Su, and Yelick[8] used static analysis to identify shared accesses and insert fences only for these accesses. However, all of these static approaches use whole program analyses, which are difficult to scale to real-world programs.

References

1. P. Ranganathan, V. Pai, and S. Adve, ''Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models,'' *Proc. 9th Ann. ACM Symp. Parallel Algorithms and Architectures,* ACM, 1997, pp. 199-210.
2. C. Gniady, B. Falsafi, and T.N. Vijaykumar, ''Is SC + ILP = RC?,'' *Proc. 26th Ann. Int'l Symp. Computer Architecture,* IEEE CS, 1999, pp. 162-171.
3. C. Lin et al., ''Efficient Sequential Consistency via Conflict Ordering,'' *Proc. 20th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2012, pp. 273-286.
4. L. Hammond et al., ''Transactional Memory Coherence and Consistency,'' *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS, 2004, pp. 102-113.
5. W. Ahn et al., ''BulkCompiler: High-Performance Sequential Consistency Through Cooperative Compiler and Hardware Support,'' *Proc. 42nd Int'l Symp. Microarchitecture,* ACM, 2009, pp. 133-144.
6. D. Shasha and M. Snir, ''Efficient and Correct Execution of Parallel Programs That Share Memory,'' *ACM Trans. Programming Languages and Systems,* Apr. 1988, pp. 282-312.
7. Z. Sura et al., ''Compiler Techniques for High Performance Sequentially Consistent Java Programs,'' *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* ACM, 2005, pp. 2-13.
8. A. Kamil, J. Su, and K. Yelick, ''Making Sequential Consistency Practical in Titanium,'' *Proc. ACM/IEEE Conf. Supercomputing,* IEEE CS, 2005, p. 15.

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.

Static analysis. The static analysis determines safe accesses using a conservative intraprocedural analysis. All function-local variables that do not escape the scope of their functions are classified as 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 is taken to ensure correctness as two function-local variables in different functions could be allocated to the same stack location. Our analysis also considers accesses to constant literals as safe.

We propose to extend the ISA to allow a compiler to flag safe memory instructions. When a processor core decodes a memory instruction and allocates a reorder buffer (ROB) entry, it sets a bit (`ss`) in the corresponding ROB entry (see Figure 2) if the compiler flags that instruction as safe. The processor later uses this information to relax memory-model constraints for that instruction. This static approach incurs little runtime complexity, but it must be conservative and could classify as unsafe some accesses to locations (especially those on the heap) that are actually safe.

Dynamic analysis. We augment our static analysis using a dynamic technique that

leverages operating system (OS) support for classifying accesses at page-level granularity.[11] The OS protects pages at the process level; we extend this mechanism to support thread-level page protection by adding a few fields to the page table entry (Figure 2). The first read or write from a thread will trigger an exception to the OS, which lets the OS keep track of the page's state (private, shared read-only, or shared read-write). The translation look-aside buffer (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 is taken to preserve memory-ordering constraints between memory accesses when the page's state changes.

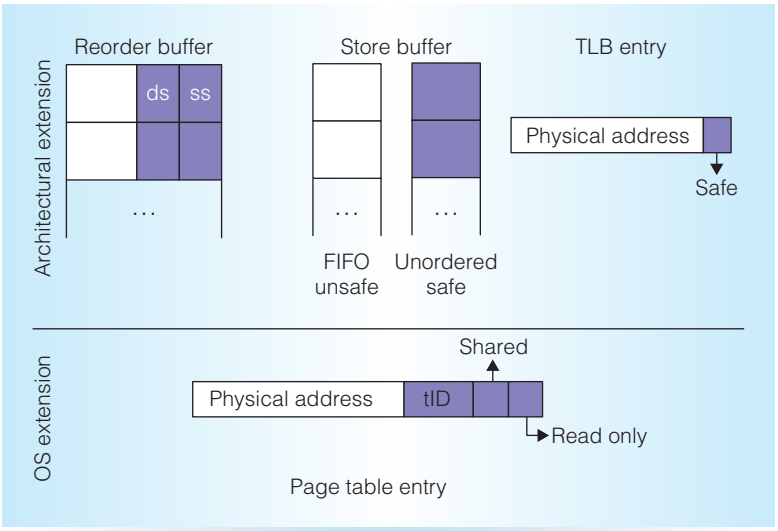

Figure 2. System extensions to exploit memory access type for providing sequential consistency (SC). Page table and translation look-aside buffer (TLB) entries are extended to identify safe pages. Safe stores and the following memory accesses are fast-tracked by committing them to the unordered store buffer. (ds: dynamic safe; ss: static safe; tID: thread ID of the last accessor.)

Hybrid analysis. Even if a page contains only one shared read-write byte, the dynamic scheme will treat accesses to any part of the page as unsafe. Thus, our static analysis, which classifies locations at a finer granularity, complements our dynamic analysis. Our proposed design uses a hybrid scheme. Because 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 (that is, either `ss` or `ds` is set on the ROB entry).

SC architecture design

Modern x86 processor implementations support a variant of the TSO memory model. To reduce the overhead due to TSO's load-load memory ordering constraint, mainstream processors include a speculative optimization called *in-window speculation*,[13] which allows loads to be speculatively executed out of order. This optimization is also useful for SC hardware.

Unlike SC, TSO allows loads to be reordered before stores, which permits the ordered store buffer optimization. Conventional SC designs require a store buffer drain for every load. Performance lost due to this restriction on the store buffer optimization is the only performance cost of SC hardware when compared to TSO hardware (assuming in-window speculation for both designs).

We propose a simple extension to significantly reduce this cost in SC: divide the store buffer into two parts, as shown in Figure 2. One of the store buffers 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 or store is safe by examining the `ss` and `ds` bits in its ROB entry.

Our SC design allows safe loads to commit from the ROB without a store buffer drain. It also allows an unsafe load to commit from the ROB without waiting for the unordered store buffer to drain. Finally, safe stores in the unordered store buffer can be coalesced. Thus, when compared to the TSO design implemented in today's x86 processors, the only additional memory-ordering restriction that our SC design imposes is that unsafe loads are stalled until the FIFO store buffer containing unsafe stores is drained.

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). In addition, we ensure that for any valid read-after-write dependency, the dependent memory accesses are of the same type. To perform store-to-load forwarding, it is sufficient to search only the unordered store buffer for safe loads and only the FIFO store buffer for unsafe loads.

Illustration

Figure 3 depicts an example to illustrate our SC hardware design's performance advantages. The left half of the figure illustrates a baseline SC hardware design, and the right half illustrates how our design (Type-SC) works. Figure 3a represents the initial states of the ROB and the store buffers for a program, and Figure 3b 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)` incurs 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 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 (right half of the picture), the long latency `St(X)` is sent to the unordered store buffer. This allows all the following safe (`St(Y)`) and unsafe (`St(A)`) stores to retire. It also allows safe (`Ld(Z)`) and unsafe (`Ld(B)`) loads to commit from the ROB. 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, the 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.

Performance evaluation

We evaluated our SC hardware's performance using the Apache web server and applications from the Parsec and Splash-2 benchmark suites. We used the FeS2 simulator to model a 16-core chip multiprocessor (CMP). We assumed in-window speculation for all the processor configurations.[13] We implemented our compiler analysis on top of our SC-preserving LLVM compiler.[8]

We consider a TSO processor running the stock LLVM compiler's binary as our baseline, as it represents the most commonly used systems today. Figure 4 compares the performance of different memory models. We found that the cost of preserving SC in the compiler is on average about 4.3 percent (Figure 4). This overhead could be further reduced using interference checks described elsewhere.[8] If we use baseline SC hardware, the total end-to-end SC cost is about 12.7 percent on average. However, by using the hybrid classification scheme, our SC design reduces the cost to 6.2 percent on average. The overhead for SC is only slightly higher (7.4 percent) when compared to executing the stock LLVM compiler's output on hardware that uses relaxed memory ordering (RMO), a model that is weaker than TSO but still sufficient to support the C++ and Java memory models.

Because we rely on time-consuming simulations, we couldn't study the entire executions of applications. As an application executes for longer periods of time, we might observe a higher overhead if locations are increasingly classified as unsafe by our dynamic scheme over time. This problem can be easily mitigated by periodically resetting the state of all the pages to safe.

Discussion

We evaluated SC overhead for out-of-order processors. Out-of-order execution helps mask some of the cost of enforcing SC constraints. Even for in-order architectures, our preliminary studies indicate that the performance cost of enforcing SC

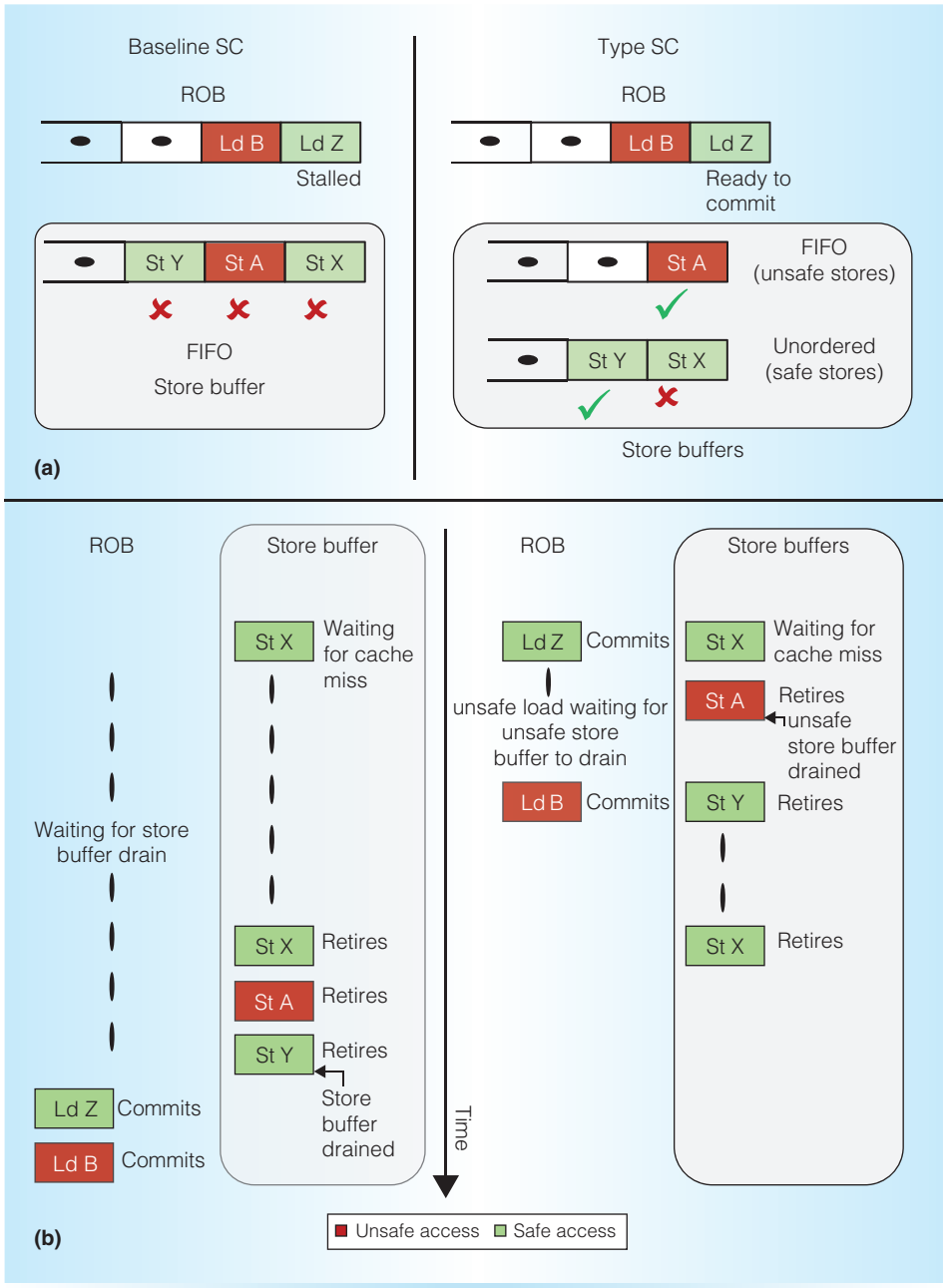

Figure 3. Comparison of a program's execution in baseline SC (left) and proposed SC hardware (right) designs. Processor states (a); execution timeline (b). Shaded cells represent unsafe accesses. Store buffer entries marked as ✘ are not ready to retire, and the entries marked as ✓ are ready to retire.

constraints is dwarfed by the penalty imposed by the limitations of an in-order core. Also, we find that our SC optimizations were equally effective for in-order processors.

The SC-preserving compiler's performance is only about 4.3 percent less than that of stock LLVM and gcc compilers at their higher optimization levels. We can't rule out future SC-violating compiler optimizations that yield more compelling performance benefits than today, but we expect that such optimizations are likely to require a more precise alias analysis than is currently available. Such improvements in static

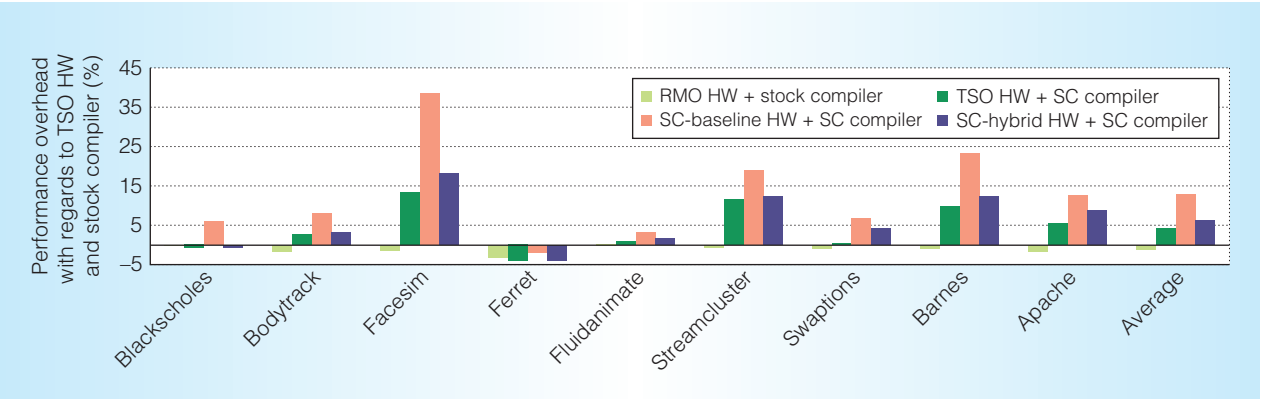

Figure 4. Comparison of our proposed system with baseline SC, total store order (TSO), and relaxed-memory-ordering (RMO) hardware (HW). Performance overhead of providing end-to-end SC is only about 6 percent on average.

analysis could also be used by an SC-preserving compiler to classify more shared-memory accesses as safe and optimize them. Any remaining memory-ordering-related overhead in the SC-preserving compiler could be further reduced using additional hardware support such as interference checks.[8]

The memory model is at the heart of a language's concurrency semantics. In the past few years, significant effort has gone into standardizing the memory-model semantics both at the language level (interface between a programmer and the system) and at the ISA level (the interface between the processor and software). These standards will be the foundation on which all future computer systems will be built. Thus, in defining these standards, it is imperative that we carefully consider all options, especially those that favor programmer productivity and reliability of software systems.

Among all the memory models, SC provides the most intuitive semantics by guaranteeing the program order and a simple shared-memory abstraction. These intuitive semantics can significantly improve programmability, debugging, and security, and they make it easy to ensure the safety properties of languages like Java. These SC memory-model advantages are well known. In fact, the spirit of even the weaker DRF0 (data-race-free-0) memory model adopted by languages like C++ and Java is also to provide SC for most programs. However, the DRF0 model guarantees SC only if a programmer manages to correctly annotate all variables potentially participating in a data race. If a programmer makes a mistake by failing to annotate even one data race in a program, all bets are off.

Lessons from C/C++ semantics in the presence of errors such as buffer overflows should teach us the following. First, programming languages should not assume programmer infallibility. Despite good intentions, even expert programmers will make simple mistakes. So, we expect there to be many unannotated data races in all but trivial programs. Second, hackers will find innovative ways to exploit weak or undefined language semantics. It is conceivable that future exploits will use data races to trigger arbitrary control flow and to compromise security. Although nondeterministic behaviors are more difficult to exploit, a hacker doesn't even have to communicate with the secure process. Simply inserting a load in the system—say, through a browser sandbox—will suffice.

To further compound this problem, concurrent software used in production today contains tens to hundreds of intentional data races[14] that arguably do not violate the intended program behavior under existing implementations of the compiler and the hardware. DRF0-based C++ and Java memory-model standards make these data races illegal, and by doing so, they risk a scenario in which future systems might execute legacy software incorrectly.

For these reasons, programming languages should instead provide safety by default. In the context of sequential programming languages, for instance, safe languages such as Java and C# provide many benefits over unsafe languages such as C. We believe that the same trend toward safety should hold for concurrent programming languages—they should provide SC semantics for all programs, with few or no mechanisms to break this abstraction.

The only reason for resorting to a memory model weaker than SC is concern for the performance overhead of ensuring SC semantics. Our work directly addresses this challenge. The memory-model interface we advocate provides ample opportunities for processor and compiler optimizations, as it allows them to freely optimize safe accesses. It also lets us take advantage of future advancements in static data-race detectors, which would increase the proportion of memory accesses that are provably safe. Also, if need be, the approach we advocate still allows expert programmers (for example, those who develop synchronization libraries) to explicitly identify safe accesses so that the processor and compiler can freely optimize them.

Finally, the techniques we've developed are also useful for enhancing the performance of programs under the DRF0 model. Every unsafe access in the DRF0 model requires a memory fence, which is expensive in modern processors. Our architecture design could provide higher performance for the DRF0 model, because it allows safe accesses to be optimized across fences.

Therefore, by debunking the myth that SC is too expensive to be practical, our work could mark the beginning of the end to the long-standing debate over memory models. MICRO

Acknowledgments

This work is supported by National Science Foundation awards CAREER-1149773, CNS-0725354, CNS-0905149, CCF-0916770, and CNS-1064844, as well as by DARPA award HR0011-09-1-0037.

References

1. B.C. Pierce, *Types and Programming Languages,* MIT Press, 2002.
2. L. Lamport, ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,'' *Computer,* Sept. 1979, pp. 690-691.
3. J. Manson, W. Pugh, and S.V. Adve, ''The Java Memory Model,'' *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 05), ACM, 2005, pp. 378-391.
4. H.J. Boehm and S.V. Adve, ''Foundations of the C++ Concurrency Memory Model,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* ACM, 2008, pp. 68-78.
5. A. Singh et al., ''End-to-End Sequential Consistency,'' *Proc. 39th Ann. Int'l Symp. Computer Architecture,* IEEE CS, 2012, pp. 524-535.
6. P. Ranganathan, V. Pai, and S. Adve, ''Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models,'' *Proc. 9th Ann. ACM Symp. Parallel Algorithms and Architectures,* ACM, 1997, pp. 199-210.
7. C. Gniady, B. Falsafi, and T.N. Vijaykumar, ''Is SC + ILP = RC?'' *Proc. 26th Ann. Int'l Symp. Computer Architecture,* IEEE CS, 1999, pp. 162-171.
8. D. Marino et al., ''A Case for an SC-Preserving Compiler,'' *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation,* ACM, 2011, pp. 199-210.
9. D. Shasha and M. Snir, ''Efficient and Correct Execution of Parallel Programs that Share Memory,'' *ACM Trans. Programming Languages and Systems,* Apr. 1988, pp. 282-312.
10. S. Adve, ''Designing Memory Consistency Models for Shared-Memory Multiprocessors,'' doctoral dissertation, Univ. of Wisconsin-Madison, 1993.
11. N. Hardavellas et al., ''Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture,* ACM, 2009, pp. 184-195.
12. B. Cuesta et al., ''Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks,'' *Proc. 38th Ann. Int'l Symp. Computer Architecture* (ISCA 11), ACM, 2011, pp. 93-104.

13. K. Gharachorloo, A. Gupta, and J. Hennessy, ''Two Techniques to Enhance the Performance of Memory Consistency Models,'' *Proc. Int'l Conf. Parallel Processing*, CRC Press, 1991, pp. 355-364.
14. S. Narayanasamy et al., ''Automatically Classifying Benign and Harmful Data Races Using Replay Analysis,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 07), ACM, 2007, pp. 22-31.

Abhayendra Singh is a graduate student in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests include parallel programming models, software-hardware interface, and parallel architecture. Singh has an MTech in computer science and engineering from the Indian Institute of Technology, Kanpur. He is a member of the ACM.

Satish Narayanasamy is an assistant professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests include computer architecture and systems, with a special focus on concurrency. Narayanasamy has a PhD in computer science and engineering from the University of California, San Diego.

Daniel Marino is a researcher at Symantec Research Labs. His research interests include programming languages, parallel programming models, and security. Marino has a PhD in computer science from the University of California, Los Angeles.

Todd Millstein is an associate professor in the Computer Science Department at the University of California, Los Angeles. His research focuses on making software systems easier to create, maintain, understand, and validate. Millstein has a PhD in computer science from the University of Washington.

Madanlal Musuvathi is a senior researcher at Microsoft Research, where he works on various aspects of concurrent and parallel systems, including program analysis, testing, verification, model checking, and memory consistency models. Musuvathi has a PhD in computer science from Stanford University.

Direct questions and comments about this article to Satish Narayanasamy, Bob and Betty Beyster Building, Room 4721, 2260 Hayward St., Ann Arbor, MI 48109-2121; nsatish@umich.edu.