The Silently Shifting Semicolon

Daniel Marino[1], Todd Millstein[2], Madanlal Musuvathi[3], Satish Narayanasamy[4], and Abhayendra Singh[5]

1 Symantec Research
daniel_marino@symantec.com
2 University of California, Los Angeles
todd@cs.ucla.edu
3 Microsoft Research
madanm@microsoft.com
4 University of Michigan, Ann Arbor
nsatish@umich.edu
5 University of Michigan, Ann Arbor
ansingh@umich.edu

Abstract

Memory consistency models for modern concurrent languages have largely been designed from a system-centric point of view that protects, at all costs, optimizations that were originally designed for sequential programs. The result is a situation that, when viewed from a programmer's standpoint, borders on absurd. We illustrate this unfortunate situation with a brief fable and then examine the opportunities to right our path.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases memory consistency models; sequential consistency; safe programming languages; data races

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 A fable

Off to a good start

Alice is a bright and enthusiastic freshman in your programming languages (PL) class, who has also signed up for a class on French cuisine — computer science and cooking are her passions. In your first class you proudly proclaim that PL research has simplified the lives of programmers and increased their productivity. Type safety and automatic memory management, concepts that were once esoteric, have achieved mainstream success today. You summarize: "Modern programming languages enforce simple yet powerful abstractions that hide the complexities of the machine and make programming easier and safer."

You then explain and formalize *sequential composition*, a fundamental concept that strings instructions together. "In the program

$$A \ ; \ B$$

the semicolon instructs the machine to perform A and *then* B." This makes immediate sense to Alice. From the time she was a toddler (A: wash your hands, and B: you can eat a cookie) through today when she follows a cooking recipe (A: heat oil in a skillet, and B: add in chopped vegetables), Alice has implicitly used sequential composition without thinking about it. She realizes the importance of formalizing such intuitive notions. Alice is thrilled by the mathematical rigor of your class.

© Daniel Marino, Todd Millstein, Madanlal Musuvathi,Satish Narayanasamy, and Abhayendra Singh;
licensed under Creative Commons License CC-BY

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Soon after, Alice learns a related concept, *parallel composition*, but this time in her cooking class. Some recipes require executing two or more sequences of instructions in parallel, thereby maximizing the usage of kitchen resources while minimizing the time taken for the recipe. While the meaning of such recipes is specified informally, Alice uses her PL skills that she so admirably learned from you to formalize parallel composition. “The recipe

$$A\ ;\ B \quad \| \quad C$$

instructs me to do A before B but gives me the freedom to start C at any time — the presence of C is not a justification to do B before A.” Alice smiles to herself and is excited by her formalization as she bikes to your class.

A chink in the armor

Unfortunately, Alice’s excitement is short-lived as she writes her first parallel program in Java:

```
A: Pasta p = new Pasta();   ||  C: if (cooked)
B: cooked = true;                     p.drain();
```

This program mysteriously throws a null-pointer exception every so often and sometimes has even stranger behavior. Alice, for the life of her, cannot understand what’s happening, since she is pretty sure her program is correct. Imagine her surprise when you respond “Oh, there is a bug in your program. You are assuming that A is always executed before B.”

“A bug? You taught me that $A; B$ means ‘A followed by B,’ and we even formalized it! If I ask a chef to boil some pasta in a pot and set a timer for two minutes, but the pot is empty when the timer fires, I will blame the chef for not executing the recipe properly. I will *never* blame the recipe.”

Fortunately you have a simple explanation. “You see Alice, a compiler performs some optimizations by reordering instructions in your program. At runtime, the hardware can execute instructions out of order, depending on whether the instructions hit or miss in the cache, and for various other reasons. So sometimes B is executed before A.”

Alice is dumbstruck. “Optimizations? Caches? I am hearing these terms for the first time. Do I need to understand all of them to understand what $A; B$ means? Programming languages are supposed to hide all these details from me.”

“They are supposed to, and they do. The problem is that you have a *data race* in your program.”

Alice is puzzled. “And ... what is a data race?”

“Ah, let me explain. In a multithreaded program, there exists a partial order among instructions called the *happens-before* relation. A program is said to have a data race if two instructions access the same memory location, one of them is a write, and they are not ordered by the happens-before relation.”

“This all sounds confusing.” Alice thinks for a bit...

Java is an unsafe language

In a short moment, Alice’s eyes gleam. “I think I got it. I just need to ensure that a happens-before ordering exists between two writes to the same variable, and between a write and a read. Then I will not have data races. Which means I can operate under the illusion that semicolons represent sequential composition: $A; B$ will mean ‘A and then B.’ ”

“Exactly Alice! You nailed it.”

But as Alice thinks harder she begins to get worried again. “But ... this illusion breaks the moment I introduce a data race in my program.”

“Yes.”

“Hmm, doesn’t this mean that Java is unsafe, since it does not protect the abstraction of the semicolon? If I inadvertently introduce a data race then my program goes haywire in unpredictable ways. This is similar to the illusion of types in C/C++. A bug, like an out-of-bounds array access, can break this abstraction.”

You are taken aback. “I guess you are right. This *is* a safety issue.” You think for a moment and then respond, “But ... protecting the semantics of semicolons would make programs run too slowly.”

“How slowly?”

“We don’t know exactly. The overheads obviously depend on the program, the programming language, the compiler, and the hardware. Experiments suggest an average overhead of 26% for a set of Java benchmarks [34]. With appropriate hardware changes, we can reduce this overhead significantly [16, 29, 5, 22, 33, 11].”

Alice is dumbfounded. “That seems entirely acceptable and is well within the cost I am already paying by moving from C++ to Java in order to obtain memory safety guarantees.”

Understanding data races

You are undeterred. “Yes, but PL design is a delicate balance between programmability and performance. We have to ensure that the utility of an abstraction justifies its cost. For instance, taking the effort to protect semicolons will not eliminate the possibility of concurrency errors. You need to avoid data races anyway.”

Alice takes another pause to understand what you just said. And her eyes gleam once again. “Ah, I think I am finally getting it. Data races are concurrency errors. So fixing my data races both provides sequential composition and ensures that there are no concurrency errors in my program.”

You: “Er, not really. Data races are neither necessary nor sufficient for concurrency errors.”

Alice is now visibly disappointed. “So I can write a data-race-free program that still has a concurrency error?”

You nod.

Alice shakes her head. “On the other hand, I can write a program that contains a data race but is nonetheless correct?”

You: “Yes, like the program you wrote earlier — it is correct in a language where semicolons means sequential composition. Fortunately there is a simple fix. Java will provide the desired semantics as long as you make all of your data races explicit. You can fix the bug in your program just by adding a `volatile` annotation on the `cooked` variable.”

The horror, the horror

Alice takes another pause. “`Vola`-what? So, you want me to mark every data race in my program with this `volatile` annotation.” Alice sighs. “I guess this is all fine as long as there are tools to help novice programmers like me to detect and remove data races. Maybe like the red squigglies I get when I make a syntax error?”

You wish you had good news. “This happens to be an active field of study. Static data-race detectors currently suffer from too many false positives [28, 25]. Dynamic data-race detectors [14] have performance overheads that are orders-of-magnitude more than the gain from compiler and hardware optimizations that are causing these issues in the first place. Finally, there are research proposals [8, 6, 20] for specialized programming models that guarantee data-race freedom, but they are not ready for the mainstream yet."

Alice takes a deep breath. "Ok, I guess with no tool support, I have to always assume that my programs contain data races. So I will just have to get used to this weaker semantics for semicolons. By the way, what is that semantics and what does it guarantee?"

You: "Unfortunately, this also turns out to be a hard problem. We have made several attempts to formalize the precise semantics and to ensure some basic sanity properties, but doing so while safely allowing the optimizations we desire is still an open problem."

Alice: "Ouch! How in the world are you programming with these semicolons then?"

You: "Luckily, it doesn't seem to matter in practice. Programs just seem to run ok even though there is the potential for unpredictable behavior."

Alice just cannot believe what her professor just said. "I quit! You taught me that modern programming languages enforce simple and powerful abstractions, but apparently performance trumps everything else. From now on I will stick to cooking — at least my kitchen never reorders my instructions behind my back!"

2 The moral of the story

"Safe" languages like Java have improved the world by bringing type and memory safety to the mainstream. We all know the benefits and preach them to our students and colleagues: providing 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.

Yet the PL research community is largely silent as this trend is being inadvertently reversed in the standardization of memory consistency models for concurrent programming languages [23, 7]. Just as achieving memory safety in unsafe languages requires unrealistic programmer discipline, achieving sane semantics in Java requires programmers to meticulously avoid data races with little language support for doing so. In this world, data races become the new dangling pointers, double frees, and buffer overflows, exposing programs to hardware and compiler implementation details that can affect program behavior in unpredictable and unsafe ways. Worse, contrary to popular belief, most data races are not concurrency errors. Our attempt with Alice's story is to bring these issues to the forefront of PL researchers' minds and agendas, hopefully in a humorous and thought-provoking way.

We propose two tenets for the semantics of multithreading in safe languages:

▶ Tenet 1. Following Pierce's definition [27], safe programming languages should protect their own abstractions — no program, including a buggy one, should be able to break the language abstractions.

▶ Tenet 2. Sequential composition is a fundamental abstraction in programming languages.

We believe that these tenets are uncontroversial and widely accepted in the PL community. The second tenet may not be explicitly considered often, but a bit of thought shows it to be self-evident. If sequential reasoning is no longer valid, how do we understand pre- and post-conditions of functions? What does it mean for a library to have an interface specification? In short, how do we modularly build and reason about programs?

The necessary conclusion from these two tenets is that *safe programming languages should preserve sequential composition for all programs.* That is, safe languages should guarantee sequential consistency (SC) [21][1] — it should not be possible for programs to expose the effects of compiler and hardware reorderings.

Put another way, recent work on standardizing memory models is *system-centric*: it assumes that essentially all sequentially valid hardware and compiler optimizations are sacrosanct and instead debates the semantics one should assign to semicolons. Somehow the *programmer-centric* view, which argues for treating the semantics of semicolons as sacrosanct and designing the software/hardware stack to efficiently provide this semantics, has gotten lost.

The memory models in Java as well as C and C++ stem from the classic work on data-race-free-0 (DRF0) memory consistency models for hardware [2]. DRF0 argues that consistency models weaker than SC are hard for programmers to understand and therefore insists that platforms provide mechanisms for programmers to regain SC. Specifically, DRF0 guarantees that *data-race-free* (DRF) programs will have SC semantics. DRF0 therefore argues for Tenet 2, but by providing weak or no guarantees when programmers fail to follow the DRF discipline, it violates Tenet 1. This situation is particularly problematic because, despite decades of research, enforcing data-race-freedom is hard statically [28, 25], dynamically [14], or through language mechanisms [8, 6, 20]. Therefore, it is safe to assume that most programs will have data races and thus will be exposed to counter-intuitive non-SC behaviors.

Given the arguments above, it is perhaps surprising that no mainstream language design committee considers SC to be a viable programming interface. Even languages like Python, which hugely favor productivity over performance, are unwilling to guarantee SC. Why are PL designers so willing to trample on the fundamental abstraction of a semicolon? In our experience, we have seen three arguments made against SC:

1. "Why encourage 'racy' programming?" — SC is unnecessary as programs should be data-race free anyway.
2. "Didn't the Java memory model (JMM) solve this problem?" — SC is unnecessary since Java provides safety guarantees for non-SC programs.
3. "It's simply too expensive." — SC is not practical for modern languages and hardware.

The next three sections rebut these arguments.

3 Data races as red herrings

One argument against SC is that data races are concurrency errors, and thus the effort to provide SC over DRF0 only benefits buggy programs that should be fixed anyway. The need for safety immediately nullifies this argument — programmers, as humans, will violate any unchecked discipline, and safe languages should protect their abstractions despite such violations. But it is still important to ask if data races should be treated as concurrency errors.

A *memory race* (or simply a race) occurs when two threads concurrently access the same memory location and at least one of them is a write. Memory races are abundant in shared-memory programming and are used to implement efficient inter-thread communication and synchronization, such as locks. *A* data race *is simply a memory race that is not syntactically identified in the program.* Such explicit identification occurs through the use of standard

[1] SC preserves a global shared-memory abstraction in addition to preserving sequential composition.

language synchronization mechanisms, like locks, as well as through programmer annotations. Therefore it is easy to show that data races are neither necessary nor sufficient for a concurrency error [30, 19]. By a concurrency error, we mean a violation of an intended program invariant that is due to the interactions among multiple threads.

Here is a simple example of a DRF program in Java that has a concurrency error:

```
synchronized(lock){ temp = balance; }
temp++;                                  ||  synchronized(lock){ balance++; }
synchronized(lock){ balance = temp; }
```

In this code, the update to `balance` from the right thread can be lost if the threads interleave in the wrong way. But this program is DRF as the only memory race occurs on the `lock` variable via the locking mechanism that is provided by the language. In this case, ensuring that no updates are lost requires each update to be *atomic*, for example by modifying the left thread to hold the lock for its entire duration.

Conversely, programs can contain data races but yet behave as intended (assuming SC semantics), like the one Alice wrote above. That program properly preserves the invariant that the pasta is not drained until it has been cooked, but it has a data race because the `cooked` variable has not been explicitly annotated as `volatile`. Research on data-race detection amply shows [31, 37, 26, 12, 19] that most data races (e.g., [12] reports 90%) are not concurrency errors. They are either intentional races that the programmer failed to mark as such or unintentional races that are nonetheless correct under SC (e.g., writing the same value to a variable from multiple threads).

Thus, from a program correctness perspective, the primary reason to remove data races from programs today is to protect against compiler and hardware optimizations. DRF0's requirement to exhaustively annotate all memory races is therefore largely a distraction from what should be the main goal: to help programmers identify and avoid concurrency errors. To this end, languages should provide mechanisms that allow programmers to specify and statically check desired *concurrency disciplines*. For example, `guarded-by` annotations [13] allow programmers to specify which locks are intended to protect which data structures, and atomicity annotations [15] allow programmers to specify which blocks of code are intended to execute atomically. In stark contrast to these kinds of annotations, `volatile` annotations do not convey the intended concurrency discipline and do not help the compiler to identify concurrency errors.

Nevertheless, data-race-free programming provides some benefits from a program understanding perspective. First, despite the `volatile` annotation's limited expressiveness as described above, it allows programmers to explicitly document inter-thread communication. However, note that the DRF discipline does not require all inter-thread communication to be documented. For example, a reference to a thread-safe queue does not require a `volatile` annotation even if threads use the queue operations to communicate — the DRF discipline only pertains to the individual, low-level memory races in the queue's implementation.

Second, a DRF program has the nice property that any synchronization-free region of code is guaranteed to be atomic [1]. This allows reasoning about program behavior as interleavings of these coarse-grained atomic regions. Programmers have to conservatively assume that dynamically dispatched method calls as well as library calls could introduce synchronization and accordingly break atomicity. Even so, the resulting granularity is still coarser in practice than what is possible in a non-DRF program, where only regions of code with no thread-shared accesses can be treated as atomic.

However, these benefits of the DRF discipline are *not* a justification to choose DRF0 over SC. DRF0 is a strictly weaker memory model than SC. This means that, from a programmer's standpoint, DRF0 shares all the deficiencies of SC, and SC shares all the strengths of DRF0. For instance, neither under SC nor under DRF0 can programmers rely on the benefits of the DRF discipline by default, due to the lack of language or tool support for enforcing that discipline. Instead, programmers must conservatively assume the presence of data races. If a programmer can somehow ensure that a program obeys the DRF discipline, however, then all the benefits of that discipline are guaranteed, both under DRF0 and under SC.

Hence, SC and DRF0 are equivalent in terms of their benefits relative to the DRF discipline. But SC additionally provides safety by protecting all programs from the counterintuitive effects of hardware and compiler optimizations, while DRF0 does not.

4 SC and the Java Memory Model

Another common objection we hear is the claim that Java's memory model has solved the safety issues with the DRF0 memory model, leaving no further incentive for SC. Indeed, the Java memory model is intended to provide a semantics to programs containing data races that still ensures important safety properties, including type safety, memory safety, and preventing "out of thin air" reads, whereby the read of a variable returns a value that was never written to that variable [23].

In the decade since its introduction, the current JMM has been the subject of much scrutiny in the academic literature. It is now known to prevent compiler optimizations that it intended to allow [9]. In fact, certain optimizations that would always be valid under SC (such as redundant read elimination) can lead to behavior that violates the JMM. As a result, commonly used compilers for Java, while likely DRF0-compliant, generate code that violates the JMM for some racy programs [36]. Yet we know of no replacement that is under consideration. In other words, despite years of research, the community has not found a viable middle ground between SC's safety guarantees and DRF0's performance.

Further, even if these issues are resolved and the JMM can achieve its goals, SC is still necessary for two reasons. First, the JMM is complex and difficult to understand. Any future compiler optimization should be carefully analyzed to ensure its JMM compliance. Given the experience to date, it is likely that this will lead to subtle compiler bugs that can subvert the JMM's guarantees in unpredictable ways. Second, we argue that the JMM's safety guarantees are much weaker than what most programmers would expect and rely upon. While the JMM has focused on obviating out-of-thin-air reads for security reasons, it is easy for other non-SC behaviors that *are* JMM-compliant to nonetheless cause serious security vulnerabilities. For example, the following variant of Alice's program can exhibit an injection attack when the right thread unknowingly accesses the unsanitized version of d:

```
Data d = getInput();        if (sanitized)
d.sanitize();          ||        d.use();
sanitized = true;
```

In other words, any behavior that violates the programmer's SC-based reasoning can potentially lead to unauthorized data access, and hence imply serious consequences for privacy and security. This concern is not merely hypothetical: we are aware of an instance where a C++ compiler optimization introduced a time-of-check-to-time-of-use security vulnerability in a commonly used operating-system kernel. Perhaps surprisingly, the vulnerability was fixed not by changing the code, but by disabling the problematic compiler optimization.

5 The path forward

In our fable, the thoroughly frustrated Alice accused computer scientists of prioritizing performance above safety. While we believe that achieving safety is worth sacrificing some speed, pragmatism insists that we still retain reasonable performance. We would be wrong to claim that providing SC is extremely efficient on existing hardware platforms. But equally wrong is the blanket claim that “SC is impractical as it disables most compiler and hardware optimizations.” The truth is somewhere in between and we will attempt to quantify the cost of providing SC below. Then we will describe a feasible path toward achieving SC in modern languages.

In many ways, the current situation mirrors the situation for garbage-collected languages a few decades ago. The PL community in the large understood the importance of type safety and automated memory management (well before the security implications for C/C++ programs became known) and collectively worked to improve the efficiency of type-safe languages. As a result type and memory safety has become the default in modern, mainstream programming languages. In the same vein, it is up to us as a community to determine if the abstraction of sequential composition is worth protecting.

Let us quantify the cost of SC over DRF0 languages. Using DRF0 terminology, memory accesses in a program can be classified into *synchronization* accesses and *data* accesses. Synchronization accesses are identified through the use of standard synchronization constructs and through annotations such as `volatile`. DRF0 languages must guarantee SC semantics for synchronization accesses, and they do so by both restricting compiler optimizations and emitting appropriate hardware fences that restrict hardware optimizations. On the other hand, the compiler and the hardware can freely optimize data accesses within synchronization-free regions of code.

Now consider a simple SC compiler which follows a “volatile-by-default” approach. It must ensure SC semantics for all accesses, whether synchronization or data. However, if the compiler can statically prove some accesses to be DRF, it is safe to aggressively optimize them. This includes accesses to local variables, compiler-generated temporaries, objects that do not escape a particular thread, and member objects that are consistently protected by the monitor of a class. The table below compares these two approaches:

	Optimized?	
Access Kind	DRF0	SC
synchronization	N	N
data, DRF provable	Y	Y
data, DRF not provable	Y	N

As we can see, the only difference between DRF0 and SC lies in their treatment of data accesses that the compiler cannot prove to be DRF. DRF0 simply assumes that such accesses are DRF, optimizing them at the expense of safety, while SC conservatively assumes that such accesses might not be DRF, obtaining safety at the expense of performance. This simple approach to ensuring SC semantics is, even today, much less expensive than is commonly thought. Research has shown ([34] for Java, [24] for C/C++) that the performance loss due to forgone compiler optimizations is negligible, since many optimizations are already compatible with SC and the rest can still be safely applied to thread-local variables. The hardware cost of additionally issued fences can be more significant, but even then, Alglave *et al.* [3] show the runtime overhead of this approach for `memcached` to be only 17.5% on x86 and 6.8% on ARM.

This overhead will continue to decrease as hardware fences become more efficient. Hardware vendors are already under pressure to optimize their fences to efficiently support synchronization accesses in DRF0 languages, and recent research suggests that significant improvements have been made [10]. Hill argued years ago [18] that there is little performance incentive to relax memory models in hardware since techniques abound to efficiently mask hardware optimizations [16, 29, 17, 5, 22, 33, 11]. Hardware platforms that are unwilling to commit to a strong memory model interface can still use these techniques to optimize fences, thereby enabling the compiler to efficiently guarantee SC to the programmer.

Orthogonally, the PL community should strive to increase the number of optimizations that the compiler can perform, and reduce the number of hardware fences it must emit, while still guaranteeing SC behavior. One approach is to prove more memory accesses as DRF, which allows them to be optimized within synchronization-free regions. But note that data-race freedom is just one sufficient condition to enable some SC-preserving optimizations, and many other opportunities exist. For instance, delay set analysis [32, 34, 4] can identify accesses that can be reordered without violating SC, in spite of the fact that they may race with accesses in other threads. Alglave *et al.* [3] report that this technique reduces the overhead of SC on `memcached` to 1.1% on x86 and 3% on ARM. Scaling such analyses to large programs, including techniques for sound modular analysis, is an important avenue for future research.

Further, an SC compiler can leverage the kinds of concurrency annotations that we mentioned in Sec. 3 to soundly and modularly identify SC-preserving compiler optimizations. For instance, the `guarded-by` annotation [13] communicates a particular locking discipline, allowing an SC compiler to optimize accesses to the protected location for the entire duration that the guarding lock is held. This includes optimizations that reorder operations across unrelated synchronization accesses — an opportunity that even a DRF0 compiler would not find since it lacks information about which synchronization accesses protect which memory locations. Therefore programmers have two good reasons to provide these annotations: avoiding concurrency errors and speeding up their code. We believe that continued research into new annotations and language constructs that establish statically checkable concurrency disciplines will uncover additional opportunities for compilers to produce fast SC code.

6 SC, DRF0, and Unsafe Languages

The safety argument we have made so far obviously does not apply to unsafe languages like C/C++. An explicit design goal in these languages is to avoid language abstractions that are not efficiently implementable on existing hardware. Given the current cost of hardware fences, it is unlikely that the overheads of SC are acceptable for these "bare metal" languages. Thus, at first blush, DRF0 seems to be the only feasible choice to ensure SC reasoning. Given that unsafe languages already require programmer discipline to retain important abstractions like memory safety, it seems acceptable to additionally require that programmers exhaustively annotate all memory races.

However, the experience with C/C++ standardization shows that even DRF0 is not efficiently implementable on current hardware platforms [7]. As a result, C/C++ has settled for a memory model *weaker* than DRF0, which we call Weak DRF0 (WDRF0). DRF programs are *not* guaranteed SC semantics in WDRF0. To get SC, programmers have to additionally avoid the use of the so-called *low-level* atomic primitives. The weak semantics of DRF programs in C++ is similar in complexity to the semantics of non-DRF programs in Java (and with similar problems in precisely pinning down this semantics [35]).

The C++ standard discourages the use of low-level atomic primitives by giving the vanilla `atomic` annotation (which is the analogue of Java's `volatile`) SC semantics. The expectation is that low-level atomics will only be used by a handful of experts to implement performance-critical libraries. Nevertheless, it is unclear how to effectively hide the weakness within these libraries such that "well behaved" programs that use them can enjoy SC reasoning. Additionally, given the predominance of benign data races in legacy software and the continued need for programmers to use ad-hoc synchronization idioms (from double-checked locking to redundant initialization of variables to write-only debug variables), it is likely that low-level atomic primitives will be present in application-level code as well.

Thus, we are currently in the unfortunate situation in which C/C++ programmers are not guaranteed SC, even if they take care to properly annotate all their races. A primary description of the C++ memory model [7] acknowledges this deficiency and states that DRF0 "is the model we should strive for," which will require hardware vendors to reduce the cost of fences. But as argued above, any improvement in the performance of hardware fences also improves the performance of SC. In effect, the DRF0 memory model is sandwiched between the currently feasible but unacceptably weak memory model WDRF0 on the one side and SC on the other side. Any improvement in the performance gap between WDRF0 and DRF0 will reduce the performance gap between DRF0 and SC even more. We predict that when DRF0 becomes acceptably efficient for C/C++, the performance of SC over DRF0 will also be acceptably small. In such a setting, it is unclear if the programmer effort to meticulously annotate all memory races and the potential for unsafe behaviors makes DRF0 worthwhile even for C/C++ programs.

7 Conclusion

The semantics of the semicolon is fundamental in mainstream programming languages. This paper makes a case that semicolons should mean what everyone already thinks they mean, namely sequential composition: $A; B$ should *always* mean A and then B. In such a world, Alice can write programs that behave according to her intuition, expert programmers writing low-level libraries don't have to carefully figure out what fences to insert where, and researchers don't have to debate what a four-line concurrent program should mean.

Language runtimes relax the meaning of semicolons today simply due to an *accident of history*. When multiprocessors were first designed, hardware architects did not know effective ways to hide single-processor optimizations from multithreaded programs, leading to relaxed-memory-model interfaces and associated expensive fences. As Hill elegantly argued more than 15 years ago [18], this is no longer necessary — architects now have a variety of techniques [16, 29, 17, 5, 22, 33, 11] to efficiently hide hardware optimizations from software (or equivalently, to drastically reduce the cost of fences), several of which are already implemented in commercial processors such as x86. Many of the design decisions that went into the Java and C/C++ memory models are centered around the need to minimize the number of hardware fences, thereby perpetuating the historical accident at the cost of program safety and programmer sanity.

We believe that it is high time to rectify this situation. We hope that Alice's story and our associated arguments will convince language designers to bring SC into the mainstream, just as Java did for type and memory safety a few decades ago. As we have argued, safe languages must preserve sequential composition, and the overhead of SC is much less than commonly thought and will be significantly reduced over time. We believe the safety benefits of SC are well worth the overhead, especially in this day and age when programmers routinely give up integer factors of performance, for example by using scripting languages, in exchange for increased productivity.

Current and future programmers need our help. If we don't save the semicolons, who will?

8 Acknowledgements

We would like to thank Hans Boehm, Mark Hill, Doug Lea, Kathryn McKinley, Ken McMillan, Todd Mytkowicz, Shaz Qadeer, Michael Scott, Karin Strauss, Ben Zorn, and the anonymous reviewers for their insightful feedback on this paper.

References

1 Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, August 2010.

2 Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

3 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *CoRR*, abs/1312.1411, 2013.

4 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *Computer Aided Verification - 26th International Conference, CAV 2014*, pages 508–524, 2014.

5 Colin Blundell, Milo M.K. Martin, and Thomas 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, New York, NY, USA, 2009. ACM.

6 Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116. ACM, 2009.

7 Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

8 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

9 Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 331–346, Berlin, Heidelberg, 2007. Springer-Verlag.

10 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48. ACM, 2013.

11 Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. Weefence: Toward making fences free in TSO. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 213–224. ACM, 2013.

12 John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

13 Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

14 Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

15 Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

16 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*, ICPP '91, pages 355–364, 1991.

17 Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 162–171, Washington, DC, USA, 1999. IEEE Computer Society.

18 Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31:28–34, 1998.

19 Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM.

20 Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84. ACM, 2013.

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

22 Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient sequential consistency via conflict ordering. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 273–286, New York, NY, USA, 2012. ACM.

23 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

24 Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an SC-preserving compiler. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 199–210, New York, NY, USA, 2011. ACM.

25 M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, 2006.

26 Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM.

27 Benjamin Pierce. *Types and Programming Languages.* The MIT Press, 2002.

28 Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 320–331, New York, NY, USA, 2006. ACM.

29 Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 199–210, New York, NY, USA, 1997. ACM.

30 John Regehr. Race condition vs. data race. `http://blog.regehr.org/archives/490`.

31 S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

32 Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

33 Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. *SIGARCH Comput. Archit. News*, 40(3):524–535, June 2012.

34 Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 2–13, New York, NY, USA, 2005. ACM.

35 Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 209–220, New York, NY, USA, 2015. ACM.

36 Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.

37 Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.