Good-Enough Memory Consistency

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

> C is for [consistency], that's good enough for me
> C is for [consistency], that's good enough for me
> C is for [consistency], that's good enough for me
> Oh, [consistency], [consistency], [consistency] starts with C!
>
> Cookie Monster, 1972

To maximize performance, compiler and hardware optimizations assume that multithreaded code is correctly synchronized. Consequently, code that is *incorrectly* synchronized (i.e., has a data race) generally has undefined semantics on modern shared-memory systems.

We argue for deviating from existing solutions to this problem in two major ways:

1. A minimal, "good-enough" memory consistency model would eliminate the most serious memory consistency problems and provide a sufficient foundation for languages and systems.
2. Restricting compiler optimizations is impractical, and we must allow compilers to optimize with abandon in synchronization-free regions.

A necessary and sufficient solution is to enforce a memory consistency model in which there is not a cycle of flow (write–read) dependencies among synchronization-release-free regions of code, a property we call *acyclic region flow (ARF)*.

Perhaps surprisingly, an existing runtime system—*Dthreads* by Liu et al. [9]—*already provides ARF*. Although designed for runtime determinism, Dthreads provides thread isolation between synchronization operations, effectively enforcing ARF. A prior evaluation shows that Dthreads' runtime isolation actually improves performance over pthreads [9].

The upshot is that compilers should perform unrestricted optimizations, and compiled programs should run under Dthreads (with determinism disabled) all the time, to get efficient, good-enough memory consistency.

1 Problem

Compilers and hardware perform optimizations assuming that all communication in a shared-memory execution is well synchronized. These assumptions allow compilers to eliminate redundant loads and stores and out-of-order processors to reorder loads and stores.

Unfortunately, in their quest for performance, programmers often omit necessary synchronization, producing incorrectly synchronized code, i.e., code that has a *data race*: two memory accesses to the same variable, at least one of which is a write, that are not order by synchronization (equivalently: that can happen simultaneously). C/C++ programs with data races have undefined semantics and have unexpected, erroneous behaviors [1, 2, 4, 5]. Java's memory model tries to preserve memory and type safety by avoiding so-called *out-of-thin-air (OOTA)* results, but the model precludes common compiler optimizations performed by commercial JVMs [6, 10, 13, 14]. Data-race-free C++ programs that use *relaxed atomic* accesses can, after common compiler optimizations, erroneously produce OOTA results [6, 13].

Figure 1 shows three programs. The programs in Figures 1(a) and 1(b) have data races. If either racy program is a C++ program and `x` and `y` are regular variables, then the program has undefined semantics. If either racy program is a Java program and `x` and `y` are regular variables, or the program is a C++ program and the accesses to `x` and `y` are relaxed atomic accesses, then compiler and hardware optimizations should avoid OOTA results, but they do not necessarily do so in practice. The code in Figure 1(c) has no data race, so it should execute with SC semantics regardless of language or variable/access type.

Experts generally think that for Figure 1(a), `r1 == 42` is *not* an OOTA result and should be allowed. However, `r1 == 42` in Figure 1(b) is an OOTA result because 42 materializes "out of thin air" and must *not* be allowed (imagine if `x` and `y` were Java references).[1] Likewise, `r1 == 42` in Figure 1(c) is an OOTA result—the program does not even have a data race!—and must *not* be allowed regardless of the language or variable/access type.

However, allowing unrestricted compiler or hardware optimizations can lead to OOTA results or undefined semantics for Figures 1(a), 1(b), and 1(c). Imagine an "optimization" that speculates that the loads of `x` and `y` will return 42, then later checks that `x` and `y` have the value 42—a self-fulfilling prophecy. This transformation would be correct assuming data race freedom. Figure 2 shows the code from Figure 1(b) after such a transformation; the transformed code clearly permits `r1 == 42`.

Prior work limits compiler and hardware optimizations to avoid OOTA results, in many cases providing stronger guarantees such as sequential consistency (SC). Several runtime and hardware approaches enforce end-to-end SC with a

[1] In particular, `r1 == 42` should be disallowed for Figure 1(b) if the accesses are Java accesses or C++ relaxed atomic accesses, not regular C++ accesses. However, this paper's solution provides well-defined semantics for all access types, including regular C++ accesses.

ASPLOS WACI, 2019, Providence, RI, USA
2019.

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = 42;

(a) Is r1 == 42 possible?

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

(b) Is r1 == 42 possible?

Thread 1	Thread 2
if (x)	if (y)
r1 = y = 42;	x = 42;

(c) Is r1 = 42 possible?

Figure 1. Three multithreaded programs, based on examples from prior work (e.g., [5, 6, 10]). In all cases, x and y are initially 0. Note that because (a) and (b) have data races, if the accesses to x and y are regular C++ accesses, then the programs have undefined semantics, as the text explains.

Thread 1	Thread 2
y = r1 = 42;	x = r2 = 42;
if (x != 42)	if (y != 42)
y = r1 = x;	x = r2 = y;

Figure 2. Code from Figure 1(b) after speculative compiler "optimization."

combination of compiler optimization restrictions and hardware support [3, 11, 12]. Boehm, Ou, and Demsky propose to eliminate only OOTA results, by restricting load–store reordering by the compiler and hardware [6, 13].

Observations and Constraints

We make two key observations that constrain the problem.

First, **memory consistency models only need to be "good enough."** Stronger memory models have not caught on because they can hurt performance and require changing compilers or hardware. To minimize these costs, we argue for good-enough memory consistency that ensures only well-defined semantics for all programs.

Good-enough memory consistency is sufficient for addressing the primary memory consistency problems afflicting languages and systems today. Good-enough consistency enables checkers and analyses to reason about programs and allows language implementations to provide key properties such as memory and type safety. Under good-enough memory consistency, data races would still be considered errors that should be avoided; programmers should *not* try to reason about complex, albeit well-defined, semantics for data races.

Second, **restricting compiler optimizations is impractical.** Modifying compilers to restrict optimizations hurts performance or adds too much complexity or both. Furthermore, it is difficult to correctly restrict optimizations to ensure certain behaviors (cf. [14]). Boehm, Ou, and Demsky show how compilers can eliminate OOTA results by protecting *dependent* load–store ordering, but tracking dependencies throughout compiler transformations is complex [6, 13]. Alternatively, restricting reordering more generally adds performance overheads.

Modern compilers optimize aggressively, assuming no interactions with other threads except at synchronization operations.[2] We argue that the compiler should be able to treat a synchronization-free region of code as a "black box" that provides the same external behavior as the original region in the source code assuming no interaction with other threads. As a result, to provide good-enough memory consistency, runtimes or hardware must execute the compiled program in a way that does not expose the compiler's unrestricted optimizations within synchronization-free regions of code.

Given these constraints, the rest of this "paper" addresses the following open question: Is it possible to provide a good-enough memory model at very low cost without modifying or restricting compilers?

2 A Good-Enough Memory Model

We propose the *acyclic region flow (ARF)* memory consistency model for compiled programs, which ensures every program execution has well-defined, OOTA-free behavior with respect to the original source program. ARF ensures that every program execution is equivalent to some execution in which the flow (write–read) dependencies among *synchronization-release-free regions (RFRs)* are acyclic.[3] That is, ARF ensures that for any execution of a program, there exists a strict partial order $\prec_{ARF}$ that is subject to the following constraints:

- For two RFRs R_i and R_j, $R_i \prec_{HB} R_j \implies R_i \prec_{ARF} R_j$, where $\prec_{HB}$ is the happens-before relation (union of program order and synchronization order).
- If a read of shared variable x by a region R_j reads from a write to x by region $R_i \neq R_j$, then
  - $R_i \prec_{ARF} R_j$ and
  - there does *not* exist a region R_k that writes x such that $R_i \prec_{HB} R_k \prec_{HB} R_j$.

We argue that ARF is both sufficient and necessary for avoiding undefined behavior:

- *Sufficient:* An ARF execution corresponds to an execution of the original program with acyclic flow (write–read) dependencies, ensuring defined behavior.
- *Necessary:* Given a non-ARF execution of a program, it is possible to derive a legal compiled program that results in an arbitrary OOTA result for that execution. Imagine compiler "optimizations" that perform

[2] With one exception: They must not introduce speculative writes, which can introduce data races into data-race-free programs such as Figure 1(c). With our proposed approach, compilers can ignore this restriction.

[3] While prior work has proposed similar models (cf. [13]), that work is in the context of restricting compiler transformations, not restricting executing programs compiled without such restrictions.

transformations that lead to arbitrary communication between two RFRs for a program that has data races permitting cyclic dependencies.

3 Enforcing Good-Enough Consistency

Runtime or hardware support can enforce the ARF memory model by providing isolation among release-free regions—specifically, by prohibiting a region's writes from becoming visible to other threads until the region completes. While hardware support could help provide such isolation, supporting unbounded regions would necessitate a complex design.

Alternatively, we note that existing runtime support called *Dthreads* [9] provides ARF out of the box. Dthreads is a runtime system that runs each thread in isolation, propagating writes to other threads only at synchronization operations, by running each thread as a process and merging a thread's writes into global state at synchronization operations. Dthreads provides thread isolation for the purposes of making programs execute deterministically [9]. Runtime determinism might hurt scalability, but Dthreads has a *protection-only* mode that disables the determinism mechanisms and enables only the thread isolation mechanisms. This mode still provides ARF.

An evaluation by the Dthreads authors shows that Dthreads actually *speeds up* a set of C programs on average compared with pthreads, mainly by eliminating false sharing [9]. Related work shows that isolating threads is helpful for eliminating false sharing [7, 8].

Interestingly, the Dthreads authors discuss memory consistency as a potential *limitation*. Dthreads provides a form of release consistency that (in the presence of data races) can lead to unexpected results compared with execution under pthreads. Nonetheless, Dthreads strictly strengthens the memory model over pthreads—providing ARF instead of DRF0 (i.e., undefined behavior for data races)—extending well-defined semantics to all programs.

4 Conclusion

Restricting compilers is costly and complex, so compilers should perform uninhibited optimizations assuming data race freedom. Good-enough memory consistency would provide a solid foundation and eliminate the most serious memory model issues.

ARF is a necessary and sufficient good-enough memory model. Dthreads provides ARF by isolating threads until synchronization operations, giving all programs well-defined semantics without restricting compilers. Furthermore, Dthreads is faster than pthreads on average, according to one evaluation [9]. Thus, unless and until more extensive evaluation says otherwise, all compiled programs should be run under Dthreads all of the time. Which just might be good enough.

Acknowledgments

Thanks to Vignesh Balaji, Swarnendu Biswas, Brian Demsky, Brandon Lucia, and Rui Zhang for helpful discussions and feedback.

References

[1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.
[2] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.
[3] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.
[4] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.
[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.
[6] H.-J. Boehm and B. Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.
[7] C. DeLozier, A. Eizenberg, S. Hu, G. Pokam, and J. Devietti. TMI: Thread Memory Isolation for False Sharing Repair. In *MICRO*, pages 639–650, 2017.
[8] T. Liu and E. D. Berger. Sheriff: Precise Detection and Automatic Mitigation of False Sharing. In *OOPSLA*, pages 3–18, 2011.
[9] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, pages 327–336, 2011.
[10] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.
[11] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.
[12] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.
[13] P. Ou and B. Demsky. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *PACMPL*, 2(OOPSLA):136:1–136:29, 2018.
[14] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.