Simple thread semantics require race detection

Hans-J. Boehm
HP Labs
Hans.Boehm@hp.com

Abstract

Data race detectors are commonly viewed as debugging tools. We argue that if we knew how to make them both fully accurate and sufficiently fast for "always on" use, they could address an additional and much more foundational role: They would allow us to give precise, simple, and safe semantics to shared variables in multithreaded programs, a problem that has so far defied a complete solution.

1. Introduction

Data races are well-recognized as a common source of particularly difficult-to-diagnose bugs in parallel programs. As a result many tools have been built to explicitly detect data races, either at compile time, or as the program is executing (cf. [6, 8]).

Although code correctness typically requires stronger properties, such as atomicity[7] or even determinacy, data-race-freedom remains interesting since it is a well-defined condition that is easy to check, even in the absence of any additional programmer-supplied specifications. This is particularly true for a number of important language specifications, notably the expected upcoming revisions of the C and C++ language standards [14, 11, 15] and the much earlier Posix threads [10] and Ada [17] standards, that explicitly treat all data races as programmer errors [1]. In these languages, an accurate (no false positives) data race detector, such as [6] or [8], by definition diagnoses only actual errors.

Here we argue that there is another, different argument for accurate data-race detection: If we could guarantee that data races are always detected, we get much simpler program semantics, and hence it becomes far easier to reason about programs.

Part of this is already fairly widely appreciated: Many modern programming languages promise *sequentially consistency* [12] i.e. that the behavior of a program can be understood as simply interleaving the actions of its threads, but only in the absence of data races [1]. Both Java and the upcoming C and C++ standards promise sequential consistency for data-race-free programs that avoid some, relatively esoteric, library routines.

But there are two additional reasons we would really like to see an accurate mechanism for detecting and avoiding data races, e.g. by throwing an exception as in [6]:

1. Data-race-free programs are independent of the granularity at which memory accesses are performed. They exhibit the same behavior on a machine that accesses memory a byte-at-a-time as it does if memory is accessed 64 bits at a time. Similarly, accesses to library or user-defined synchronization-free data structures behave atomically. In both cases, a half-updated data structure can't be observed by another thread, since the observer thread would introduce a data race.

   Note that this property is orthogonal to sequential consistency.

2. It has proven to be very difficult to define the meaning of programs with data races in a way that both disallows behavior that can result in blatant security holes, and allows simple meaning-preserving compiler transformations on source programs.

The first point above would allow us to, for example, remove a small, but ugly, wart from the Java language specification. If we could ensure that no data races were ever executed, we would no longer need the special exception (section 17.7 in [9]) that allows a half-updated `long` or `double` to be observed by the program. A program that would otherwise have seen a half-updated `long` would instead throw a data-race exception, so that the program would never see such a value.

But the second point is much more important: If we didn't have to define the semantics of data races, the most complex piece of the Java memory model specification, the so-called "causality" treatment could be removed, solving some open problems with the specification in the process.

We present an overview of why it is so important and attractive to remove this specification, and how we got here to start with. Although we believe that a few members of the community are generally aware of these issues, we are only aware of a hints of this in the literature [16, 6], and we believe it is under-appreciated. By specifying the problem more directly, we point out possible avenues for addressing the remaining performance problems, and argue that a practical solution might actually be feasible.

2. Enforcing "causality"

Languages like Java must ensure that malicious code cannot generate "out of thin air" results. A simple memory model that always allows a load to see the result of a racing store to the same memory location does allow "out-of-thin-air" results. To see this consider

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

where `x` and `y` are shared variables, and all variables are initially zero.[1] (Effectively one thread copies `x` to `y` while the other simultaneously copies `y` to `x`.) If each load of a shared variable can see the store of the shared variable in the other thread, `r1` and `r2` may contain any value whatsoever, e.g. `r1 = r2 = 42` becomes an acceptable final outcome. If both loads see a value of 42, both stores will store 42, circularly validating the originally loaded values.

This even corresponds to an intuitively semi-plausible execution. The compiler might predict, e.g. based on past profiling information, that this code is likely to be executed when `x = y = 42`, speculatively store 42 in each thread, and then verify that the loaded value matches the expected one. Such code would be correct for sequential execution, but result in our questionable outcome.

However, in a Java-like language, this is completely unacceptable. If a piece of untrusted and malicious code introduced such a race on `String` variables, `x` and `y`, these semantics would allow

[1] There is an interesting related example in section 17.4.8 of [9]

the implementation to set `x` and `y` to any "out-of-thin-air" string, including the password of your bank account.

Obvious attempts to outlaw this behavior fail by also outlawing common and important compiler transformations. To see the beginnings of the difficulties, consider that `r1 = r2 = 42` must be a valid outcome if the stores in the above example are replaced by `y = 42` and `x = 42` respectively; anything else would be very expensive to enforce on some common hardware.

This problem was addressed by the causality treatment in the Java memory model[13], which was reflected in section 17.4.8 of [9]. Although this approach, unlike its predecessor, appears to be somewhat workable, it still looks problematic. In particular:

- This is arguably the most complex aspect of the Java specification.
- There is strong evidence that it has been correctly understood by very few people. For example, [13] claims (Theorem 1) that semantics are not affected by reordering independent statements, which is refuted by [2], a result that was also a surprise to even JSR133 (memory model) participants, including me.
- Although alternate specifications for Java-like languages have been proposed [16], these also appear too complex to have generated much of a following.

Although [13] is probably the best existing solution to this problem, and certainly represents substantial progress, it is not entirely satisfactory, and we'd like to do better.

3. Alternate solutions

The obvious solution here is to insist that every load of a shared variable can only see a store that "happens before" it, i.e. that ordering between stores and corresponding loads must be enforced with sufficient synchronization. Such an ordering avoids any risk of causal cycles as in the example above.

We can guarantee such ordering by avoiding data races. There are two common approaches to doing so:

1. Data races result in "undefined behavior". This solves the problem in C++0x [4, 11], except in the presence of the previously mentioned esoteric library calls.
2. Statically enforce the absence of data races, as in e.g. [3].

At a minimum, the former doesn't work for Java-like languages. The latter requires extremely challenging trade-offs between type system complexity and generality, which appears likely to continue to limit its acceptance in the mainstream.[2]

"Always on" data race detection provides a third option and a way out. If a racing load throws an exception rather than returning a value, there is no need to specify the value. The only substantial disadvantage appears to be performance.

4. Variations and implications

We can avoid specifying general semantics of racing loads by enforcing conditions weaker than data-race-freedom.

At a minimum, there is no need to detect write-after-read races. Even if we don't detect these, every load either sees a store that happened before it, or results in an exception. This significantly, but probably still insufficiently, reduces the cost of race detectors like [8] by eliminating the need to track reads from each location, while perhaps only modestly reducing their utility for debugging.

In a slightly different context, Luis Ceze et al. suggest [5] detecting only sequential consistency violations. This may be more practical, but seems to sacrifice more essential properties, such as insensitivity to memory access granularity.

We are currently exploring other intermediate properties that might represent more desirable tradeoffs.

[2] Clearly such a static solution may still be attractive for particularly critical applications.

Acknowledgments

This benefitted from discussions with Sarita Adve, Luis Ceze, Dhruva Chakrabarti, Cormac Flanagan, Vivek Sarkar, Rob Schreiber, Yin Wang, and others.

References

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

[2] D. Aspinall and J. Sevcik. Java memory model examples: Good, bad, and ugly. VAMP07 Proceedings `http://www.cs.ru.nl/~chaack/VAMP07/`, 2007.

[3] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overby, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. Technical Report UIUCDCS-R-2009-3032, UIUC, 2009.

[4] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. Conf. on Programming Language Design and Implementation*, pages 68–78, 2008.

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

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[7] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71:89–109, 2008.

[8] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 3rd edition*. Addison Wesley, 2005.

[10] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.

[11] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, programming language - C++ (committee draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`, 2008.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[13] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proc. Symp. on Principles of Programming Languages*, 2005.

[14] C. Nelson and H.-J. Boehm. Concurrency memory model (final revision). C++ standards committee paper WG21/N2429=J16/07-0299, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2429.htm`, October 2007.

[15] C. Nelson, H.-J. Boehm, and L. Crowl. Parallel memory sequencing model proposal. C standards committee paper WG14/N1349, `http://www.open-std.org/JTC1/sc22/wg14/www/docs/n1349.htm`, February 2009.

[16] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP 07*, March 2007.

[17] United States Department of Defense. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*, 1983. Springer.