Position paper: Nondeterminism is unavoidable, but data races are pure evil

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

Abstract

Modern mainstream programming languages distinguish between "atomic" (or sometimes "volatile") variables, and ordinary data. Atomic accesses are treated as synchronization constructs, and support concurrent access with well-defined semantics. In contrast, concurrent accesses to ordinary data, if at least one access is an update, constitute a data race. Code with data races does not have well-defined semantics. Usually such code may fail completely when recompiled or run on a different operating system version. In C and C++ data races are equivalent to assignments to out-of-bounds array elements; any data race can result in arbitrary failures, including application crashes, hangs, and inexplicably and completely wrong answers.

These language specifications, combined with implementation realities, make it unsafe to exploit "benign" data races to obtain performance, even if we are willing to tolerate approximate answers. Furthermore, even if we happen to get lucky, and code with data races happens to execute correctly with our current compiler, data races provide at best inconsequential performance advantages over atomics. In fact, there are interesting, and probably common, cases in which data races provide only a minor performance advantage, even over pervasive locking to avoid them, *at sufficiently large core counts*. We demonstrate such a case.

1. Context

The call-for-papers for the "RACES – SPLASH 2012 Workshop on Relaxing Synchronization for Multicore and Manycore Scalability" [21] states "A new school of thought is arising: one that accepts and even embraces nondeterminism (including data races), and in return is able to dramatically reduce synchronization, or even eliminate it completely." There has also been much other discussion of "benign data races", including for example [20]. A Google search for the phrase "benign data race" returns thousands of results.

2. Introduction

Conventional multithreaded applications perform nondeterministic operations. Often that nondeterminism is inherent in the application. For a parallel system that processes large volumes of on-line ticket orders, it is not meaningful to require that we deterministically choose which customer gets the last ticket.

In other cases, nondeterminism may be hidden from the user. For example, memory allocators return addresses that are dependent on the interleaving of allocation calls, and those addresses typically affect hash functions and hence the layout of hash tables, etc. These typically do not affect the program output, but enforcing such a restriction seems impractical. For example, it would prevent us from printing debugging statistics summarizing the occupancy of hash buckets.

There has been recent work on deterministic execution of conventional multithreaded programs (cf. [3, 8]). But such systems reduce performance in order to ensure determinism, especially at higher core counts. And my experience has been that nondeterministic execution significantly increases the number of observed bugs. I believe such approaches tend to trade easier debugging for appreciably worse test coverage.

Other recent work accepts nondeterminism, as I do, but confines its impact to program sections that really require it. [16] This is much more consistent with the direction we pursue here.

Although tolerating nondeterminism, in various forms, can clearly reduce synchronization overhead, and improve performance, data races are an entirely different matter. *Data races invalidate core semantic guarantees provided by the programming language and compiler. In the presence of data races, it becomes impossible to reason about program behavior.* Simply removing data races from programs that happen to work with them, typically does not worsen scalability, defined informally as the number of processor cores we

can effectively take advantage of. And, in the case of C11 and C++11, the absolute performance cost of removing data races from racy programs is also essentially zero.

I first argue that reasoning about programs with data races is impossible. I make this argument from two different, redundant, angles: First, language specifications disallow data races. Second, I argue that programs with data races can, and do break in practice, where "break" can include totally unexpected behavior, not merely reading a slightly obsolete value. And there are strong reasons to believe that such breakage becomes more likely and important as code is recompiled with more modern compilers. Finally, I briefly summarize C11, and C++11, and Java facilities for avoiding data races at near zero performance cost.

Most of these arguments are not terribly new (cf. [7, 1]). I recast them in the context of this workshop and further motivate them with some new small, but interesting, empirical results.

3. Data races vs. programming standards

For purposes of this discussion, I use a slightly informal version of the standard definition of a data race: A *memory location* is an any independently updateable block of memory, typically a scalar variable or object field. Detailed definitions vary. Posix [13] threads leave it implementation defined; Java, C11, and C++11 give precise definitions. Two memory accesses *conflict* if they access the same memory location, and at least one of them updates it. Two accesses constitute a *data race* if they conflict and can occur simultaneously.

We have a long tradition of disallowing data races in programming language environments that support threads. Some older specifications did not very explicitly address shared variable semantics, but the rest have generally been quite clear on this issue, even if they lacked precision in certain other respects [4]. The most notable examples follow:

The 1983 ANSI Ada standard [25] states:

> "For the actions performed by a program that uses shared variables, the following assumptions can always be made:
>
> - If between two synchronization points in a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.
> - If between two synchronization points in a task, this task updates a shared variable whose task type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.
>
> The execution of the program is erroneous if any of these assumptions is violated."

Every Posix standard [13] since 1995 has stated:

> "Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it."

The C11 [14] and C++11 [15] standards state:

> "The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior."

This "undefined behavior" treatment is identical to that afforded an assignment to an out-of-bounds array location. It does not just mean that a racing load can read an arbitrary value; it means that any behavior whatsoever is allowed. This is often informally described as "catch-fire" semantics.

Java is notably different from all of the above. It gives a simple guarantee [10] along the above lines:

> "A program is correctly synchronized if and only if all sequentially consistent executions are free of data races.
>
> If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent."

However, it then proceeds to provide a very complex semantics for programs, including those with data races.[1] Unfortunately the semantics for data races are now known to have serious deficiencies. In particular, they are inconsistent with compiler optimizations commonly implemented by Java virtual machines. [22, 2] It appears that clean specifications for data races are fundamentally at odds with common compiler optimizations which assume that variables do not change asynchronously, and there is no accepted solution to the Java specification problem. Thus the semantics of Java data races are supposedly defined, but not, in fact, understood.

In the absence of data races, and without use of the loopholes we discuss below, all of the above arguably provide simple sequentially consistent [18] semantics[2], though this is much clearer for Java, C11, and C++11 than it is for the older standards.

4. Data races vs. compilers

There are excellent reasons for allowing arbitrary misbehavior in the presence of a data race. Many common compiler

[1] This was primarily motivated by the very legitimate need to provide some guarantees for untrusted code running inside a trusted program. Performance considerations were a secondary issue.

[2] And data races are defined with respect to those same sequentially consistent semantics.

optimizations result in data race behavior that is not easily explained to a programmer who is not a compiler expert. A thorough analysis of how this may happen for essentially all kinds of so-called "benign" data races is given in [5]. Here I present only a few examples to illustrate some common problems.

Perhaps the most common compilation surprise does involve only the value read in a data race, but results in no progress whatsoever, because the value tread is always the original one. This commonly happens when the original value is cached in a machine register.

For example, consider a simple flag `f` which is set by one thread, and waited for by another, with the loop:

```
while (!f) {}
```

Most compilers, with optimization enabled, would observe that `f` is loop invariant, and hence can be read once outside the loop. Thus the code is transformed to

```
r = f; while (!r) {}
```

where *r* is a register. If `f` is not already set when the loop starts, this becomes an infinite loop, which most probably hangs the application and prevents any further progress.

The fundamental problem here, as in all other cases, is that compiler optimizations assume that variables not explicitly modified don't change. Language definitions support this assumption; by violating it, the programmer lies to the compiler, causing it to operate under false assumptions.

In the above case, the code behaves as though a single valid, though very obsolete, value is read from `f`. In other cases, program behavior is harder to explain. Consider:

```
{
  bool r = x; // read shared var.
  if (r) y = new T();
  ...  // r not modified.
  if (r) y->f = ...;
}
```

The compiler may load `r` from `x` before the first conditional, decide it needs to spill `r` between the conditionals, and reload it *directly from* `x` before the second conditional. If `x` was modified by a race in the interim, the allocation may not be executed, while the assignment to `f` is.

Other data-race induced issues include:

- Counters incremented by only one thread appear to decrease in value because an older, previously loaded, value is reused by the compiler or hardware.
- A value is read with two separate load instructions, or by two separate memory transactions, resulting in a partially updated integer, or possibly even pointer.
- In C++, wild branches through not-quite-initialized method table pointers.

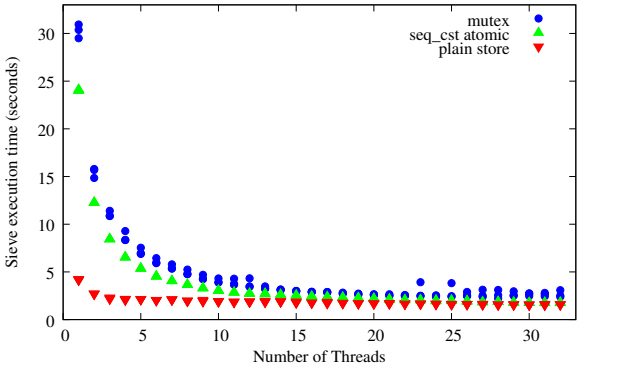

Figure 1. Parallel Eratosthenes Sieve to 10^8

Hidden data races in programs not only confuse compilers, they also confuse human readers. In a data-race-free program a synchronization-free code section s is atomic. A thread observing or modifying an intermediate state of s would have to race race with s. We use that observation routinely when reasoning informally about programs. When analyzing an assignment `x = y` in C, we don't have to ask how large `x` is, or how many load and store instructions are required to implement the assignment. We can treat it as a single atomic operation because of the data-race-free restriction.

Any data race violates this atomicity property. Thus clearly such data races should be clearly labeled in the source code to keep the code comprehensible to human readers. I argue in the next section that for C11 and C++11, if we are willing to annotate data races, we can eliminate them just as easily, with essentially no added cost. By doing so, we both make the code more readable, and avoid the risk of unexpected compiler optimizations.

5. Data races rarely help scalability

Protecting write operations to x with a lock specific to x in order to avoid data races doesn't qualitatively hurt scalability. Even without synchronization, write accesses to x are effectively serialized by the cache coherency protocol. Every write to x needs to acquire exclusive access to the cache line holding x. If x is protected by lock l_x, we need to acquire exclusive access to l_x, possibly a small number of times, in addition to x. Each access to x effectively becomes more expensive, but there is no reason to expect the overhead to increase with the number of cores or threads; what used to be a single cache line acquisition becomes a small constant number of cache line acquisitions.

This effect is illustrated in Figure 1. This gives execution times in seconds for a variant of the parallel Sieve of Eratosthenes program from [4], in which bits are set unconditionally even if it they were already set, thus making writes dominant.[3] The main loop nest is

[3] The underlying problem is similar to mark bit handling in a parallel mark-sweep collector, though there we would always test the bit before setting

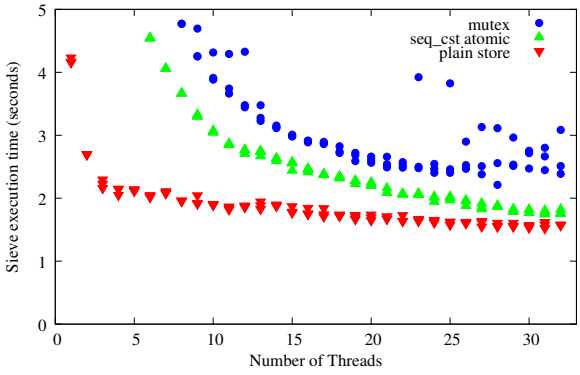

Figure 2. Parallel Sieve to 10^8, expanded vertical scale

```
for (my_prime = start; my_prime < SQRT;
     ++my_prime)
    if (!get(my_prime)) {
      for (multiple = my_prime;
           multiple < MAX;
           multiple += my_prime)
        set(multiple);
    }
```

where `MAX` is 10^8 and threads are assigned consecutive `start` values beginning at 2. Each entry in the sieve array accessed by `get` and `set` is a separate byte. The benchmark was run on an HP DL785 32-core server containing 8 quad-core 2.2GHz AMD Opteron 8354 processors running Red-Hat Enterprise Linux 5.2.[4] Each configuration was run three times, and point for all results are included.

Figure 2 repeats the results of Figure 1 with an expanded vertical scale to separate results for higher processor counts.

The top (blue circle) set of measurements gives results with individual sieve entries protected by one of a large number of locks. Entries are hashed to locks, and these are acquired and released in `set()` (and `get()`). The bottom (red) measurements use ordinary load and store instructions as would be generated by a variant with races. I manually confirmed that the compiler, in this case, generates correct code in spite of the data race. (The middle measurements are addressed in the next section.)

Clearly the lock-based version is slower, as expected. But the lock-based version actually benefits substantially from more processors, exhibiting a speedup, relative to the same code running on a single thread, of more than a factor of 11 on anything close to the full number of processors. On the other hand, the racy version speeds up by less than a factor of 3, and the two approach each other at higher thread/processor counts. In a sense, the synchronized program *scales better* than the racy version. I conjecture that this is due to memory bandwidth limitations. There are many locks, but due to the hashing scheme, they all easily fit in cache. The data does not.

Lock-based synchronization does not limit scalability; contention does. This benchmark generates relatively little contention on any specific piece of data. However all threads contend for hardware memory bandwidth.

Note that this benchmark produces a deterministic output, but its internal behavior, and even the total amount of work performed, is highly nondeterministic. Which thread processes multiples of which `my_prime` prime value is schedule dependent.[5] Since locks are held for a single memory access, their purpose is only to avoid data races and the associated unpredictable compiler and hardware optimization effects, not to enforce determinism.

The situation with potentially racing read accesses is more complicated, but not likely to be qualitatively different. Simply protecting a read access to x with a mutex or traditional reader-writer lock adds a write access to the lock, where previously there was only a read access. That may introduce contention where otherwise all cores could have had concurrent access to the same shared cache line. However, if no actual contention exists without the lock, then either only one thread is accessing (the cache line containing) x, or there are very few writes to x.[6] In the former case, the lock adds no contention. In the latter case, we can use the techniques from the next section, possibly combined with techniques like RCU (read-copy-update [19]) or seqlocks [12, 17, 6] to avoid writes to locks for otherwise read-only operations.

So far I've argued that data races are not helpful in improving speedup relative to one or two threads running the same code. But our data clearly also shows that on an absolute scale, i.e. the one that really matters, locks to prevent data races are still quite expensive, though sometimes primarily at *lower* core counts.[7]

6. Data races no longer help performance

In environments like traditional Posix threads, locks are the usual mechanism for avoiding data races, and there often were no other real options. Modern languages introduce other mechanisms to avoid data races at less cost. C++11[15] and C11 provide `atomic` objects. Java provides `volatile` fields and `java.util.concurrent.atomic`. I will refer to them generically as *atomics*. Atomics can be safely concurrently accessed without introducing data races. By default they behave as though each access were individually pro-

it. I focus on the write-dominated version to simplify the issues for now. If reads are involved, the techniques from the next section become more important.

[4] This is the highest core count machine I had convenient access to. It is not the latest model. I would be surprised if other modern x86 machines behaved qualitatively differently.

[5] This is a possible explanation for the somewhat variable execution times under ostensibly identical conditions.

[6] If there are no potentially concurrent writes to x at all, no lock is needed in any case.

[7] I did not measure power consumption. It may well be that for power the overhead persists at higher processor counts, but it clearly remains a small constant.

tected by a lock specific to that atomic; program behavior remains sequentially consistent even in the presence of concurrent accesses to atomics.

Atomics cannot be used to ensure atomicity of longer code sequences; they provide atomicity for only a single access. In many cases, this makes them much harder to use than locks. But in a number of simple cases (e.g. "done" flags, counters) they may be the most straightforward way to express an algorithm. Since they can typically be implemented directly using a combination of atomic hardware operations and memory fences, the implementation overhead is typically significantly less than that of acquiring locks around a single access. And in the cases considered here, there is never a need to add something like a write to a lock object for read operations; hence there is no added cache line contention. Default (sequentially consistent) loads (reads) of atomics can be implemented on x86 processors with a single `MOV` instruction. The upcoming ARMv8 [11] architecture in addition provides direct hardware support for sequentially consistent atomic stores.

In addition to the default behavior, all three of the above languages provide for *weakly ordered* accesses to atomics that allow the guarantees of sequentially consistent behavior for atomics to be relaxed. Java provides the (admittedly under-specified) `lazySet()` method on (`java.util.concurrent.atomic`) atomics; C++11 and C11 support explicit `memory_order_` specifications for accesses to atomics. These support faster operation at a huge increase in the complexity of the programming rules that must be understood by the programmer. But this complexity is still far less than what is required to understand data-race behavior. In particular, weakly ordered atomics:

- Prevent unexpected and unintended compiler transformations that unexpectedly break code, by informing the compiler that concurrent access and concurrent updates are possible.
- Provide portable, well-defined semantics, independent of the hardware memory model and compiler idiosyncrasies.
- Provide some minimal visibility ordering guarantees that prevent the most counter-intuitive behaviors. All variants that I have mentioned guarantee at least "cache coherence", i.e. the accesses from all threads to a single memory location behave as though they were simply interleaved. For example, a counter that is only atomically incremented cannot appear to decrease, as it can with data races and in a few hardware memory models.
- By providing well-defined portable guarantees, and explicit options to trade off performance against simpler semantics, they at least force programmers to think about memory ordering issues before writing code with serious memory ordering bugs.

On x86 and ARM, C/C++ `memory_order_relaxed` atomics can be implemented using the same instruction sequences as for ordinary memory accesses. They impose extremely minimal compiler constraints, and performance is unlikely to be measurably worse than for racy code that happens to compile correctly. On x86, the same is true for `memory_order_release` and `memory_order_acquire`, as well as Java `atomic/volatile` loads and `lazySet()`. [23]

Unfortunately, real support for C11/C++11 atomics is just beginning to appear. However our previous experiment also illustrates the expected performance. The racy ("plain_store") code in this case happens to implement `memory_order_relaxed`, `memory_order_release` and Java `lazySet()` semantics, and is essentially what we would expect a reasonable compiler to generate for that case. The middle green results reflect the cost of using atomics with default sequentially consistent behavior, as they would be implemented by a compiler that simply followed [23] without sophisticated analysis. Each atomic store is implemented with a trailing `MFENCE` instruction. This introduces significant overhead at low thread counts. But an x86 `MFENCE` instruction, like many such instructions, can be thought of as purely flushing a local buffer [24], and in our experience, processor performance characteristics match that view. (The microbenchmark results in [6] lead to a similar conclusion, across a wider variety of platforms and in a different context.) For the sieve example, it appears that this local overhead becomes increasingly insignificant at higher core counts, as memory bandwidth becomes the bottleneck.

7. Conclusions

I've argued that data races incur substantial risk of complete program failure, since they cause compilers to transform programs on the false assumption that variables are not concurrently accessed. At the same time they make programs less readable to humans by inducing them to make similar false assumptions. Even if programs with data races happen to be favorably compiled, as they often are today, the performance advantage over code that avoids those races is currently smaller than you might think, and will approach zero in the next year or two as implementations of C11 and C++11 atomics become real. I expect the odds of "favorable compilation" of data races to decline as compiler optimizers increasingly leverage better defined memory models. [5, 9]

Acknowledgments

Sarita Adve and Terence Kelly provided very useful comments on earlier drafts.

References

[1] S. V. Adve. Data races are evil with no exceptions: technical perspective. *Commun. ACM*, 53(11), 2010.

[2] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, August 2010.

[3] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *The Second Workshop on Determinism and Correctness in Parallel Programming: Wodet 2*, 2011.

[4] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. Conf. on Programming Language Design and Implementation*, 2005.

[5] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

[6] H.-J. Boehm. Can seqlocks get along with programming language memory models. In *MSPC*, 2012.

[7] H.-J. Boehm and S. V. Adve. You don't know jack about shared variables. *Communications of the ACM*, 55(2), February 2012.

[8] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: A relaxed consistency deterministic computer. In *ASPLOS*, 2011.

[9] L. Effinger-Dean, H.-J. Boehm, D. Chakrabarti, and P. Joisha. Extended sequential reasoning for data-race-free programs. In *MSPC*, 2011.

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The java language specification: Java se 7 edition. `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`.

[11] R. Grisenthwaite. Armv8 technology preview (slides). `www.arm.com/files/downloads/ARMv8_Architecture.pdf`, retrieved Aug. 3, 2012, 2011.

[12] S. Hemminger. Fast reader/writer lock for gettimeofday 2.5.30. Linux kernel mailing list August 12, 2002, `http://lwn.net/Articles/7388/`.

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

[14] ISO JTC1/SC22/WG14. ISO/IEC 9899:2011, information technology — programming languages — C. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853` or an approximation at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf`.

[15] ISO JTC1/SC22/WG21. ISO/IEC 14882:2011, information technology — programming languages — C++. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372` or a close approximation at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf`.

[16] R. L. B. Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a determinism-by-default parallel language. In *POPL*, pages 535–548, 2011.

[17] C. Lameter. Effective synchronization on Linux/NUMA systems. Proceedings of the May 2005 Gelato Federation Meeting (`http://www.lameter.com/gelato2005.pdf`), 2005.

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

[19] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Engineering at Oregon Health and Science University, 2004.

[20] S. Narayanasamy et al. Automatically classifying benign and harmful data races using replay analysis. In *Proc. Conf. on Programming Language Design and Implementation*, pages 22–31, 2007.

[21] RACES Workshop organizers. SPLASH 2012 RACES workshop call for participation. `http://soft.vub.ac.be/races/call-for-participation/`.

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

[23] J. Sevcik and P. Sewell. C/C++11 mappings to processors. `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`, retrieved Mar. 3, 2012, 2011.

[24] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocesors. *Communications of the ACM*, 53(7):89–97, July 2010.

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