How to miscompile programs with "benign" data races

Hans-J. Boehm
HP Laboratories

Abstract

Several prior research contributions [15, 9] have explored the problem of distinguishing "benign" and harmful data races to make it easier for programmers to focus on a subset of the output from a data race detector. Here we argue that, although such a distinction makes sense at the machine code level, it does not make sense at the C or C++ source code level.

In one sense, this is obvious: The upcoming thread specifications for both languages [6, 7] treat all data races as errors, as does the current Posix threads specification. And experience has shown that it is difficult or impossible to specify anything else. [1, 2]

Nonetheless many programmers clearly believe, along with [15] that certain kinds of data races can be safely ignored in practice because they will produce expected results with all reasonable implementations. Here we show that *all kinds* of C or C++ source-level "benign" races discussed in the literature can in fact lead to incorrect execution as a result of perfectly reasonable compiler transformations, or when the program is moved to a different hardware platform. Thus there is no reason to believe that a currently working program with "benign races" will continue to work when it is recompiled. Perhaps most surprisingly, this includes even the case of potentially concurrent writes of *the same value* by different threads.

1 Background

We define a data race as simultaneous access to the same memory location by multiple threads, where at least one of the accesses modifies the memory location. As is discussed in [5], this is essentially equivalent to other common definitions, e.g. that used in [10] based on accesses not ordered by a *happens-before* relation.

Definitions of most mainstream programming languages provide simple semantics (sequential consistency [12]) for well-behaved programs without data races, but treat data races in one of two ways:

1. They treat data races as errors that are not necessarily detectable by the implementation, but may result in arbitrary application results. This is commonly described as "catch-fire" semantics for data races. For reasons discussed in [2] and [5], this treatment was chosen by Ada 83 [19], Posix threads [11], and the upcoming C++ [5, 6], and C [7] memory model.

2. They attempt to give some semantics to data races that are (1) weak enough to allow standard compiler optimizations, (2) strong enough to preclude behavior that would invalidate important security properties, and (3) simple enough to be usable. Java [14] is the prime example of this, but this piece of the Java memory model is now widely recognized as failing to satisfy either (1) or (3) [16, 2], leaving the actual meaning of Java programs with data races in doubt. There is growing concern about the feasibility of this approach, motivating research on more speculative alternatives [2, 13].

In the first case data races are clearly disallowed, while in the second case their semantics are merely unclear. We assume the first case here.

2 "Benign" data races

Although data races have arguably never been allowed in multithreaded versions of C and C++, they are fairly commonly used in existing code, and there is a strong perception that they are acceptable in certain contexts. For example, [15] addresses the problem of distinguishing "benign" and "destructive" races, so that programmers can concentrate on repairing the latter[1], and data races are regularly reported in well-debugged code (cf. [13]). Even the OpenMP 3.0 specification [18] accidentally contains examples with data races[2] in spite of

the fact that the specification, unlike its predecessors, explicitly disallows them.

Since [15] looks at machine language programs, the notion of a "benign race" is in fact meaningful in its context. Although data races essentially have no defined semantics in C or C++, they are perfectly meaningful in x86 assembly code [17]. An assembly program with a data race is not necessarily an error. In fact synchronization primitives are commonly implemented with assembly code that has data races.

However, we argue here that such a distinction is not useful if the data race existed in a C or C++ which was then compiled, and the code was intended to be portable.[3] Not only is the original source code technically incorrect in such cases, *future recompilation of the code by reasonable compilers may realistically introduce bugs*.

We make our case by going through all of the different types of "benign races" discussed in section 5.4 of [15] in turn, and demonstrating how each such source-code-level "benign race" could very reasonably be compiled into incorrect code.[4] We discuss them in a somewhat different order to facilitate the presentation.

2.1 Double checks for lazy initialization

This is well-known to be incorrect at the source-code level [8]. A typical use case looks something like

```
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

Nothing prevents an optimizing compiler from either reordering the setting of `my_data` with that of `init_flag`, or even from advancing the load of `my_data` to before the first test of `init_flag`, reloading it in the conditional if `init_flag` was not set. Some non-x86 hardware can perform similar reorderings even if the compiler performs no transformation. Either of these can result in the final read of `my_data` seeing an uninitialized value and producing incorrect results.

2.2 Both values are valid

This refers to the case in which a reader and writer may race, but in the event of a data race it does not matter whether the reader sees the old value (before the write), or the new value (after the write). One example used in [15] is a consumer-producer application in which the producer adds an element to a shared buffer by adding an element to the array and then incrementing a counter of the number of elements.

The problem here is that it is quite possible that the reader will see a result *inconsistent with reading either the old or the new value*. For example, if the hardware only supports 16-bit indivisible writes, but the counter is a 32-bit value, and if the counter is incremented from $2^{16}-1$ to 2^{16}, then the reader may see the high bits of the old value, and the low bits of the new value, unexpectedly yielding a value of zero.

In other cases, the observed value may be inconsistent with reading *any* particular value. For example, assume the consumer code from above were written as something like:

```
{
    int my_counter = counter;  // Read global
    int (* my_func) (int);

    if (my_counter > my_old_counter) {
        ... // Consume data
        my_func = ...;
        ... // Do some more consumer work
    }
    ... // Do some other work
    if (my_counter > my_old_counter) {
        ... my_func(...) ...
    }
}
```

If the compiler decides that it needs to spill the register containing `my_counter` between the two tests, it may well decide to avoid storing the value (it's just a copy of `counter`, after all), and to instead simply re-read the value of `counter` for the second comparison involving `my_counter`, effectively transforming the code to:

```
{
    int my_counter = counter;  // Read global
    int (* my_func) (int);

    if (my_counter > my_old_counter) {
        ... // Consume data
        my_func = ...;
        ... // Do some more consumer work
    }
    ... // Do some other work
    my_counter = counter;  // Reread global!
    if (my_counter > my_old_counter) {
        ... my_func(...) ...
    }
}
```

This may lead to the first test failing, and the second succeeding, with the call to `my_func` resulting in a wild branch. Another transformation also resulting in a wild

branch is presented in [5]. In either case, the outcome does not correspond to just reading either the old or new value of `counter`, and clearly results in unexpected and incorrect behavior.

As we point out in [5], the core problem arises from the compiler taking advantage of the assumption that variable values cannot asynchronously change without an explicit assignment. Such an assumption is entirely legitimate if data races are disallowed by the language specification as in our setting. No such asynchronous changes are possible in the absence of a data race. As far as we have been able to determine, existing compilers such as gcc are designed to allow such transformations, though such effects are rarely observed in practice.

2.3 User constructed synchronization

User constructed synchronization should not rely on ordinary data operations. Doing so generally runs into the same potential problems as in section 2.2. The upcoming C [7] and C++ [6] standards provide `atomic` operations for this purpose. These serve roughly the same purpose as Java `volatile` fields. They make user-defined synchronization easily recognizable. By maintaining this information a race detector can easily ignore races on such accesses.

Using ordinary variables for such accesses is unsafe in portable code for at least three reasons:[5]

1. The operations may again fail to be indivisible, as in the preceding section. For example, on an x86 machine, a pointer that was misaligned and spanned a cache-line boundary might appear to be updated in two steps. Although most compilers avoid generating such code, they are not prohibited from doing so.

2. As in the preceding section, they violate the compiler's assumption that ordinary variables do not change without being assigned to. This can lead to surprising results, such as the register-spilling example above. If the variable is not annotated at all, the loop waiting for another thread to set `flag`:

   ```
   while (!flag) {}
   }
   ```

   could even be transformed to the, now likely infinite, but sequentially equivalent, loop:

   ```
   tmp = flag;  // tmp is local
   while (!tmp) {}
   ```

3. In portable code, there is no way to prevent hardware or compiler reordering of memory operations that is inconsistent with the synchronization semantics. For example, consider a user-defined barrier implementation (in the sense of e.g. OpenMP barriers [18]) that eventually waits for the last thread to set a flag, using a loop like the one above. Assume that `data` is a shared variable accessed after the barrier. Either the compiler or hardware could effectively reverse the order of the two statements in

   ```
   while (!flag) {}
   tmp = data;
   ```

   causing `data` to be accessed *before all threads have reached the barrier*.

The last two are sometimes addressed through the use of implementation-specific fence constructs and/or C `volatile`. These solutions are either not portable or incomplete.

2.4 Redundant writes

None of the above concerns apply to the case of two threads writing *the same value* to the same location without synchronization. So long as this is the only data race, and the code is compiled in a straightforward manner, all readers of the affected location must either happen-before both writes, and see an earlier one, or happen-after the writes, and see the written value. It seems unlikely on any conventional architecture that a store would be implemented such that rewriting the same bits would result in a final value different from either of the written values.[6] Thus the behavior of such a program initially appears to be equivalent to one in which only one thread writes to the location.

However, there are again legitimate compiler transformations justified by the data-race-free assumption that introduce incorrect behavior. We are not aware of any compiler that currently correctly performs such transformations, but we expect that to change as compilers start to conform to, and take advantage of, recently revised and clarified memory model specifications.

To illustrate this, we first observe that it is generally incorrect for a C or C++ compiler, meeting either the Posix threads [11] or upcoming language threads specifications [7, 6], to introduce a spurious assignment of a variable to itself. Borrowing our example from [4], assume we define `f()` as follows:

```
f(x)
{
    ...;
    for (p = x; p != 0; p = p -> next) {
        if (p -> data < 0) count++;
    }
}
```

where count is a global variable that could possibly be concurrently accessed by another thread.

It is unacceptable [4] (though common among modern compilers [3][7]) to transform the loop by promoting count to a register, effectively transforming it to:

```
reg = count;
for (p = x; p != 0; p = p -> next) {
    if (p -> data < 0) reg++;
}
count = reg;
```

The original loop writes count only if the list p contains a negative element; the transformed version writes count unconditionally. In the case of a list of positive elements, the compiler introduced a spurious assignment of count to itself. Thus if the original program calls f on a list of positive elements concurrently with an increment of count by another thread, there is no data race. However the transformed program does have a data race. And if the increment of count occurs in the middle of the execution of the transformed loop, the write to count by the transformed loop will cause the increment to be lost.

However the transformation once again becomes legal if the variable count is already unconditionally written in the same synchronization-free code region. In that case, no concurrent access to count by another thread is possible. Such an access would create a data race with the unconditional write.

Now consider the case in which Thread 1 runs

```
count = 0;
f(positives);
```

while Thread 2 runs

```
f(positives);
count = 0;
```

where f is still defined as above.

This program accesses the shared variable count only in that both threads clear it. Thus the only data race is the redundant store to count. If the program is run with the untransformed version of f(), count is properly (and redundantly) cleared after both threads finish.

But the presence of the stores to count *legitimizes the register promotion transformation*. Since count is known to be updated, and the language rules guarantee the absence of races, the compiler can safely assume that there are no concurrent accesses to count, and may thus inline the call to f(positives) and register promote count in the loop, as described above. Since the argument to f() contains no negative elements, each thread then effectively performs the following accesses to count:

Thread 1:

```
count = 0;
reg = count;
count = reg;
```

Thread 2:

```
reg = count;
count = reg;
count = 0;
```

Assuming the original value of count is 17, these may interleave as

```
thread2_reg = count;
count = 0; // thread 1
count = thread2_reg;  // Writes 17
thread1_reg = count;  // Reads 17
count = 0; // thread 2
count = thread1_reg;  // Writes 17
```

This leaves count set to its original value of 17. Paradoxically, the presence of the redundant write race enabled compiler transformations that allowed an outcome in which neither write is seen, producing incorrect results.

Although current compilers are likely to perform this transformation unconditionally (i.e. even without the data race) or not at all, it appears likely that future compilers will have exactly the behavior described above. Performing the transformation unconditionally is not standards conforming, and potentially breaks code under very rare, but hard to describe, conditions. Nonetheless, this kind of register promotion transformation is occasionally important, and we expect compilers to look for opportunities to preserve it, such as when the promoted variable is already unconditionally accessed.

Thus even redundant write data races are likely to reintroduce the kind of very rare, but unpredictable, failures that recent work on memory models has been trying to avoid.

2.5 Disjoint bit manipulation

This category is described in [15] as:

> There can be data races between two memory operations where the programmer knows for sure that the two operations use or modify different bits in a shared variable.

However, such data races are not generally benign, even at the machine code level, if both of the racing accesses modify the variable. Specifically, an update to one

such group of bits generally involves:

1. Read the whole variable.
2. Update the relevant bits.
3. Write back the whole variable.

If two such updates to different groups of bits are perfectly interleaved, one update (the one whose write operation occurs first) will be lost, certainly not a benign effect.

A much more plausible case can be made that such races are benign if only one of the racing threads performs an update. But even in that case, it is possible to construct, admittedly convoluted, examples, in which the racing write again gives the compiler license to generate code that leads to a racing read seeing an incorrect value. These roughly parallel the construction from the last section.

Consider, for example, the case in which we have a shared variable `x` with bit-fields `bits1` and `bits2`. Thread 2 simply reads `x.bits2`. Thread 1 executes the following code, in a context in which the programmer, but not the compiler, knows that `i` = 3:

```
x.bits1 = 1;
switch(i) {
    case 1:
        x.bits2 = 1;
        ...;
        break;
    case 2:
        x.bits2 = 1;
        ...;
        break;
    case 3:
        ...;
        break;
    default:
        x.bits2 = 1;
        ...;
        break;
}
```

Since this code is only executed when `i` = 3, it is data-race-free, except that Thread 1 writes `x.bits1`, while Thread 2 reads `x.bits2`.

The upcoming C and C++ standards (as well as existing compilers) treat adjacent bit-fields, like `bits1` and `bits2` in the example, as a single memory location. Thus the initial assignment to `x.bits1` allows a correct compiler to conclude that no concurrent accesses to `x.bits2` may take place, since these would introduce a data race. Thus the compiler is perfectly justified in introducing a speculative store to `x.bits2`, e.g. to reduce code size. Thus the above might effectively be transformed to:

```
x.bits1 = 1;
tmp = x.bits2;
x.bits2 = 1;
switch(i) {
    case 1:
        ...;
        break;
    case 2:
        ...;
        break;
    case 3:
        x.bits2 = tmp;
        ...;
        break;
    default:
        ...;
        break;
}
```

This again introduces writes to `x.bits2` on the actually taken code path, introducing a data race on `x.bits2`, and allowing Thread 2 to see a `bits2` value that was logically never stored. Again the race on disjoint bits gave the compiler license to perform a transformation that broke the code.

3 Conclusions

Although there is a perfectly reasonable notion of benign data race at the level of x86 or similar machine code, the same is not true at the C or C++ source program level. Standards are increasingly clear about prohibiting data races. We have argued that ignoring this prohibition introduces a practical risk of future miscompilation of the program. Depending on the platform and type of race, this risk varies from essentially guaranteed miscompilation (e.g. if the platform doesn't easily support indivisible data accesses of the right kind) to rather obscure consequences of compiler data-race-freedom assumptions that are not yet common. But even the latter are likely to be a real issue in future mainstream compilers. And they are difficult to avoid with certainty, since the problematic transformations are usually difficult or impossible to characterize for a non-compiler-expert.

C or C++ data races are always unsafe in portable code. In Java, the implemented semantics of data races are unclear. [16, 2]. Although "benign" races in Java are not clearly incorrect, they nonetheless put the programmer into an uncertain and dangerous territory that can be avoided by avoiding data races.

4 Acknowledgments

We thank Dhruva Chakrabarti, Pramod Joisha, and the reviewers for helpful comments on an earlier draft.

References

[1] ADVE, S. Data races are evil with no exceptions. *Communications of the ACM* (November 2010), 84–84.

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

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

[4] BOEHM, H.-J. N2338: Concurrency memory model compiler consequences. C++ standards committee paper WG21/N2338=J16/07-198, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2338.htm`, August 2007.

[5] BOEHM, H.-J., AND ADVE, S. V. Foundations of the C++ concurrency memory model. In *Proc. Conf. on Programming Language Design and Implementation* (2008), pp. 68–78.

[6] C++ STANDARDS COMMITTEE, PETE BECKER, ED. Programming Languages - C++ (final draft international standard). C++ standards committee paper JTC1 SC22 WG21 N3290, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3290.pdf`, April 2011.

[7] C STANDARDS COMMITTEE (WG14). Committee Draft: Programming Languages — C. C standards committee paper WG14 N1539, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf`, November 2010.

[8] DAVID BACON ET AL. The double-cchecked locking is broken declaration. `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[9] FLANAGAN, C., AND FREUND, S. N. Adversarial memory for detecting destructive data races. In *Proc. Conf. on Programming Language Design and Implementation* (2010), pp. 244–254.

[10] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java Language Specification, The (3rd Edition)*. Sun Microsystems, 2005.

[11] IEEE, AND THE OPEN GROUP. *IEEE Standard 1003.1-2001*. 2001.

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

[13] LUCIA, B., CEZE, L., STRAUSS, K., QADEER, S., AND BOEHM, H.-J. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA* (2010).

[14] MANSON, J., PUGH, W., AND ADVE, S. The Java memory model. In *Proc. Symp. on Principles of Programming Languages* (2005).

[15] NARAYANASAMY, S., ET AL. Automatically classifying benign and harmful data races using replay analysis. In *Proc. Conf. on Programming Language Design and Implementation* (2007), pp. 22–31.

[16] SEVCIK, J., AND ASPINALL, D. On validity of program transformations in the java memory model. In *ECOOP 2008* (2008), pp. 27–51.

[17] SEWELL, P., SARKAR, S., OWENS, S., NARDELLI, F. Z., AND MYREEN, M. O. x86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53*, 7 (July 2010), 89–97.

[18] THE OPENMP ARB. OpenMP application programming interface: Version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[19] UNITED STATES DEPARTMENT OF DEFENSE. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*, 1983. Springer.

Notes

[1] [9] explores another interesting technique to make a similar distinction for Java

[2] For example, see A.2.2. Most examples involving explicit use of the `flush` primitive have similar issues, which we believe will be fixed in the next draft.

[3] We do not thoroughly consider the case of data races introduced by the compiler. These may occur either because the compiler is incorrect, at least based on upcoming standards [3, 4], or because it introduced a speculative load of a value that was subsequently not used. We continue to observe the first case in code designed to test for it, but both appear to be rare in practice [15, 13]. If speculative dead loads were a serious issue, it should be possible to have the compiler replace them with prefetches [13] to eliminate the problem at very modest cost.

[4] We believe that our arguments for Section 2.4, and for the read/write case in Section 2.5 were not previously known, while the others have been discussed elsewhere.

[5] Recall that the original work in [15] identified benign races in machine code. Particularly for user-defined synchronization, the associated source code may have not been intended to be portable. It might have been in-line assembly code, for example. In our view, such code should be treated differently from C or C++ code by a data race detector.

[6] Surprisingly, *temporary* changes due to a redundant write are sometimes observable for large objects. David Dice, in `http://blogs.sun.com/dave/entry/memcpy_concurrency_curiosities` describes a scenario in which a large object store is implemented by a SPARC "block initializing store" instruction, clearing memory before writing to it, thus avoiding a cache line fetch, but allowing a racing reader to see a temporary zero value.

[7] We unfortunately confirmed this behavior for several recent versions of gcc, including 4.5.1, which otherwise generates surprisingly fast and clever code for this loop.