How to Miscompile Programs with "Benign" Data Races

Hans-J. Boehm
Date: 5/26/2011

Data Races

Two memory accesses conflict if they
- access the same memory location, e.g. variable
- at least one access is a store

A program has a data race if two data accesses
- conflict, and
- can occur simultaneously in a sequentially consistent execution.

A program data-race-free (on a particular input) if no sequentially consistent execution results in a data race.

Ada 83

[ANSI-STD-1815A-1983, 9.11] 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.

Data races are errors!

Posix Threads Specification

[IEEE 1003.1-2008, Base Definitions 4.11] 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.

Data races are errors!

C++ 2011 / C1x

[C++ 2011 FDIS (WG21/N3290) 1.10p21] 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.

Data races are errors!

(and you get atomic operations to avoid them)

Java

Data races are not errors.
But we don’t know what they really mean.

November 2010 CACM Technical Perspective

"Data races are evil with no exceptions."

- Sarita Adve, technical perspective for Goldilocks and FastTrack papers on data race detection.

On the other hand:

- Narayanasamy et al, “Automatically Classifying Benign and Harmful Data Races Using Replay Analysis”, PLDI 2007.
- Flanagan and Freund, “Adversarial memory for detecting destructive races”, PLDI 2010.
- Several other recent papers make similar distinctions.
- 4500 Google results for “benign data race”.

Our Claim:

Data races in Ada, C, or C++ code:

- Are specified to be incorrect.
- May start failing the next time you recompile the program.
  - No matter what type of data race it was.
  - Or how benign you thought it was.

Data races at the machine level

- Are OK.
- Though mainly because ISAs don't distinguish synchronization accesses.

Data races in Java?

How a data race can cause things to go wrong

Paper considers each kind of "benign" data race in

Narayanasamy et al, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis", PLDI 2007.

We instead give a general overview of what can go wrong:

- Operations may become visible out of order.
- Write may fail to become visible.
- Read may see a value not written.
- May execute a path that corresponds to no possible read value.
- Even redundant writes of the same value may fail.

Write may fail to become visible

Thread 1

```
while (!flag) {}
printf("1\n");
```

Thread 2

```
flag = true;
printf("2\n");
```

If load of flag is moved out of loop, a likely outcome:

Thread 2 will print 2.

Thread 1 will loop forever.

Operations may become visible out of order

Thread 1

```
data = 42;
done = true;
```

Thread 2

```
while (!flag) {}
assert (data == 42);
```

Writes to data and done may be reordered by compiler or hardware.

Unobservable without data races.

Racing read may see value not written

Thread 1

```
x = 300;
```

Thread 2

```
x = 100;
```

If memory is accessed a byte at a time (or x is misaligned), this may be executed as:

```
x_high = 0;
x_high = 1;   // x = 256
x_low =  44; // x = 300;
x_low = 100; // x = 356;
```

Execution path may correspond to no read value.

```
unsigned x;

If (x < 3) {
   … // async x change
   switch(x) {
      case 0: …
      case 1: …
      case 2: …
   }
}
```

Assume switch statement compiled as branch table.

May assume x is in range.

Asynchronous change to x causes wild branch.

- Not just wrong value.

Paper has a completely different example.

Even redundant writes can be problematic!

Thread 1

```
x = 17;
```

Thread 2

```
x = 17;
```

- *Is* `x` *guaranteed to be 17 when both threads finish?*
  - –Hard to see why not.
  - –*But future optimizers may break this.* (Next slides.)
  - –Probably not in all contexts.

    But dangerous contexts are hard to characterize.

Self-assignments

Self-assignments:

- Adding `x = x;` self-assignment in sequential code is OK.
- Normally *not* OK in multi-threaded code:

```
r1 = x;
                    x = 17;
x = r1;
```

- Introduces data race; hides concurrent assignment to `x`.
- OK in multi-threaded code next to a store to x:

```
x = 17;   ➔   x = 17; x = x;
```

- Cannot introduce data race or hide assignment!
  - Original program would have had to already contain race!

How to miscompile redundant writes

Thread 1	*Thread 2*
`x = 17;`	`x = 17;`

x is set to 17 (twice!)

Thread 1	*Thread 2*
`r1 = x;`	
	`x = 17;`
`x = r1;`	
	`r2 = x;`
`x = 17;`	
	`x = r2;`

x retains its original value!

But why self-assignments? (1)

```
struct { char a; char b; char c; char d;} x;

x.a = 1; x.b = 1; x.d = 1;
```

```
r1 = x.c; x = 0x01010001; x.c = r1;
```


Why self-assignments? (2)

```
int count;   // global, possibly shared
…
for (p = q; p != 0; p = p -> next)
   if (p -> data > 0) ++count;
```

```
int count;   // global, possibly shared
…
reg = count;
for (p = q; p != 0; p = p -> next)
   if (p -> data > 0) ++reg;
count = reg; // may be spurious count = count;
```

Conclusions

Even the most seemingly benign data races can easily be miscompiled.

- Clearer programming language memory models ➔ Compilers are more likely to leverage data-race-freedom assumptions.
- Aggressive use of data-race-freedom assumption ➔ More interesting ways to break code violating the assumption.
- Most of the resulting breakage is hard to test for.

Thank you

Subtitle placeholder goes here

First line of copy goes here.

- First level bullet goes here and can be quite long
  - Second level bullet goes here. Try to keep bullet lists simple
    - Third level bullet goes here. Use no more than you need to explain your point

