How to Miscompile Programs with "Benign" Data Races

Hans-J. Boehm, HotPar 2011

nick black <dank@qemfd.net>

Atlanta PWL #09, 2018-10-09

"Benign" data races do not exist at the source level.

Effects of data races depend on compilation flags, compiler version, hardware, and OS.

Attempts to assign useful semantics to data races (Java, .NET) prohibit classes of compiler optimizations.

Compilers can make any transformation that preserve observational expectations *within a thread*. Across threads, all bets are off without synchronization.

historical context and results

Multithread-aware memory model (declaring all data races to be errors having undefined behavior) only added to C11, C++11.

Portable, performant atomics added by C11 `<stdatomic.h>`, C++11 `<atomic>`.

Complex compiler/CPU technology stresses naive mental models.

Race detection tool research at the time hoped to distinguish "destructive" from "benign" races (this paper is, in part, a response to that trend).

Data races are difficult to detect in mainstream PRAM-model languages, Recognizing their seriousness motivated programming language research, better software engineering practices, and work on tools.

data races, how do they work

"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."

Essentially equivalent to e.g. Java's "happens-before" definitions.

source: let's speed up lazy init

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

doubly-checked lazy initialization disasters

Bad idea: Bypass "expensive" lock acquisition in common (initialized) case.

Problem 1: Compiler reorders writes to `my_data`, `init_flag`. Thread sees a high init_flag, but my_data is not actually prepared, `tmp` gets bogus value.

Problem 2: Processor reorders stores to `my_data`, `init_flag`. Without lock, there's no memory barrier ensuring ordering visibility, same result as above.

Proper solutions: `pthread_once()`, C++11/C11 `call_once()`, C++11 static block scope (can be optimal), lift initialization out of concurrent path (probably optimal)

why would this happen?

Store motion eliminates redundant assignments by lifting them out of subblocks. *Instruction scheduling* optimizes for instruction cache loads, frontend decoding, and OOO resources, and can move independent code arbitrarily. *Dependency optimization* might perform a write earlier, to have it ready by the time of its use.

Processor store reordering is performed to eliminate redundant stores, make better utilization of buses/caches, better balance bank accesses, and reduce interprocessor traffic.

source: let's turn a light off and on

```
t1() {
  while(1) {
    sleep(1);
    set_light(brightness);
  }
}

t2() {
  t = 0; brightness = 0;
  wait_until(event) {
    brightness = (t++ % 2) ? 0 : 4;
  }
}
```

both values are valid, until they're not

Bad idea: Avoid lock complexity between publisher and polling consumer signalling via a bimodal shared integer.

Problem 1: Compiler optimizes away check entirely, as there's no way for the variable to be modified in consumer control flow. Code never runs.

Problem 2: Architecture doesn't support integer width, compiler makes up for it in multipart software load. Alternatively, compiler materializes constant on the fly in multistep operation. Thread sees value outside the two expected values.

Proper solution: atomics, or good ol' locks for complex data (*not* `volatile`!)

source: let's write an arbitrary value

```
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(...) ...
}
```

Bad idea: Avoid lock complexity between publisher and polling consumer signalling via a shared integer where all values are acceptable.

Problem: Compiler never actually generates temporary variable, reloading from the shared variable, which changes between conditionals. Thread “impossibly” branches into the grim lands of undefined behavior, is eaten by a grue.

source: let's count negatives

```
static int count = 17;

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

t1() { count = 0; f(positives); }

t2() { f(positives); count = 0; }
```

adding writes results in fewer writes

Bad idea: Avoid lock complexity by only writing a single constant (with only positive inputs, only 0 ought ever be written to count).

Problem: Compiler uses register for accumulation, writes back modified result.

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

come on mary, don't fear the mutex

Acquisition of an uncontended Linux mutex since 2.6.0 (2003) is merely an atomic lock instruction on a memory location. No system call, no bus locking. no fuss.

Locking is cheap. Contention is expensive.

Worried about performance? `perf-lock`, you're welcome. Off-CPU analysis can currently best be effected via `perf`+eBPF cantrips.

does this apply to me, a non-dinosaur?

Any PRAM-model concurrency implementation allowing sharing of mutable data is susceptible to data races, and most further susceptible to unexpected results due to compiler/interpreter optimization.

- Go
- Rust (unsafe dialect)
- Java
- Python
- Javascript...

tools and techniques for avoiding data races

- Languages supporting non-PRAM models (CSP, immutable MP, etc.)
- Scope-based RAII synchronization (C++ `std::lock_guard`)
- Raw checkers (clang's `scan-build`, Java's RacerD)
- Annotations (clang's TSA, `javax.annotation.concurrent`)
- Dynamic analysis tools aplenty:

Go 1.1's `-race`
TSan (clang 3.4+),
ThreadSanitizer (gcc 4.8.1+)
Valgrind's `helgrind`