Document number: P1916R0
Date: 2019-10-06 (Pre-Belfast)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: SG1

There might not be an elegant OOTA fix

“The three rules of the Librarians of Time and Space are: 1) Silence; 2) Books must be returned no later than the last date shown; and 3) Do not interfere with the nature of causality.” - “Guards! Guards!”, Terry Pratchett

Introduction

Here’s one way of thinking about why the out-of-thin-air problem is tricky to resolve: we want additional constraints on the concurrent part of the memory model, but the desire for those constraints is motivated by the dependency ordering established during execution of the non-concurrent parts of the program on real hardware. So we want sequential code to maintain a(n implicit) memory ordering constraint on concurrent code. The problem of outlawing OOTA, then, is about defining which sequential dependencies the compiler is not allowed to optimize away.

Much recent OOTA work has focused on abstracting out the sequential parts of a program behind some relatively comprehensible mathematical formalism that distinguishes which syntactic dependencies are true semantic dependencies and which are false (for instance: this is done with event structures in the modular relaxed dependencies paper). The reasoning goes: “this subset of dependencies are true semantics ones, and so the compiler must maintain them in the generated code; thus, regardless of how clever the sequential optimizations are, they must leave an ordering constraint in place”.

These approaches have worked well (in the sense of matching both our intuitions and implementations) in small litmus tests, and can distinguish between the OOTA examples that everyone agrees can’t and shouldn’t ever occur, and the RFUB-like ones that real compilers may allow.

I don’t think these solutions can scale up to the whole language, though. Litmus tests largely ignore undefined behavior and non-concurrent sources of nondeterminism. We’ll look at cases where a compiler optimization can take an obviously-bad-OOTA example and counterintuitively remove the ordering constraint when a very slight tweak is applied to the surrounding code. Any OOTA fix must therefore either disallow a large class of sequential optimizations, or else must mathematically model most of C++'s quirks.

Undefined behavior can eliminate dependencies

Consider the following program:

```
std::atomic<int> x{0}, y{0};
int r1, r2;

void t1() {
  r1 = x.load(memory_order_relaxed); // A
  int local = (r1 & 1);
  if (local == 0) {
    y.store(1, memory_order_relaxed); // B
  }

void t2() {
  r2 = y.load(memory_order_relaxed); // C
  x.store(r2, memory_order_relaxed); // D
}
```

In the memory model as it exists today, this program must terminate with r1 == 0. Any other outcome would require a reads-from edge from D to A, where D stores a nonzero value. Then r2 must be nonzero, and so there's a reads-from edge from some store to C, which could only come from B. But in any execution consistent with the intra-thread semantics, if r1 is nonzero then B never executes, so there is no candidate store to y. (On the reflector, I claimed that disallowing the r1 == 0 outcome relied on the standard's current informal recommendation against OOTA. Mark Batty pointed out that this was incorrect, and r1 == 0 is required, not just suggested).

However, consider this slightly tweaked version of t1:

```
// UB if this function is ever called.
[[noreturn]]
inline void unreachable() {
  return;
}
void t1() {
  r1 = x.load(std::memory_order_relaxed);
  int local = (r1 & 1);
  if (local == 0) {
    y.store(1, std::memory_order_relaxed);
  }
  if (local != 0) {
    // If we get here, UB.
    unreachable();
  }
}
```

On recent versions of GCC and clang, this is compiled to something like the following[1] on ARM, a weakly ordered architecture:

```
t1():
adrp x0, .LANCHOR0
add x0, x0, :lo12:.LANCHOR0
ldr w1, [x0]
str w1, [x0, 4]
add x0, x0, 8
mov w1, 1
str w1, [x0]
ret
```

The dependency between the load and the store has been eliminated, and the outcomes disallowed above may occur here.

Is this a miscompilation? I don't think you can come to a firm conclusion just by reading the text of the standard. It does a good job describing UB that originates solely from the (axiomatically defined) concurrent semantics, and a more-or-less adequate job describing UB that originates solely from the (operationally defined) single-threaded semantics, but does not really cover how they might interact.

Regardless of whether or not this compilation *is* broken, you can construct arguments both ways as to whether or not it *should be*.

The argument for preventing reordering: The outcome is extremely surprising, and not exhibited on any underlying hardware memory models. We have no way of knowing if these sorts of examples appear in practice. In a large program, across unbounded inlining, they may manifest in surprising ways.

The argument for allowing reordering: the optimizer benefits (or may benefit) from removing the branches. These optimization benefits are real and measurable. Programmers in the real world

[1] https://gcc.godbolt.org/z/4vNQQs

don’t actually employ the sort of reasoning we used above, and don’t produce programs that require it. *De minimis non curat lex*.

I have considerable sympathy for the arguments in both directions. I want to highlight a reasonable-at-first-glance argument that I don’t think stands up to scrutiny, though:

> “This reordering should be allowed. Undefined behavior can do anything; it may well replace `unreachable()` with `y.store(1, std::memory_order_relaxed)`. Once it does that, then both branches of the conditional are the same, and the example devolves into one that everyone agrees is a fake dependency”:
>
> ```
> if (local == 0) {
>   y.store(1, std::memory_order_relaxed);
> } else {
>   y.store(1, std::memory_order_relaxed);
> }
> ```

I think that if you buy this argument, consistency demands that you buy a similar one about the following example:

```
std::atomic<int*> x{nullptr}, y{nullptr};
std::atomic<bool> misbehave{false};
void t1() {
  int* ptr = new int;
  x.store(ptr, std::memory_order_release);
}
void t2() {
  int* ptr = x.load(std::memory_order_acquire);
  y.store(ptr, std::memory_order_release);
}
void t3() {
  int* ptr = y.load(std::memory_order_acquire);
  if (ptr != nullptr) {
    *ptr = 1;
  }
}
void t4() {
  if (misbehave.load(std::memory_order_relaxed)) {
    unreachable(); // Defined as above.
  }
}
```

> “If t3 obtains an invalid pointer, then its dereference of that pointer is UB. That UB may cause a relaxed store of `true` to `misbehave`. The call to `unreachable()` may in turn be treated as a relaxed store of a garbage pointer value to `x`. This can result in reads-from and happens-before relations that satisfy all consistency predicates and obey the single-threaded semantics; this program therefore has UB and no guarantees are imposed on its execution.”

But this is awful; it’s the second-simplest example of release-acquire semantics that there is, mixed with a thread that touches no shared state and does nothing. For the language to mean anything, this program has to run as intended and without the possibility of UB.

This example used a particularly simple type of UB, in a way that might make it seem like this issue is restricted to simple “conditional-branch-to-UB means the condition was false” scenarios. But there are others:

- Type-based alias analysis: An equality comparison between two pointers-to-void may be removed if each is dereferenced after casting to incompatible types.
- Lifetime rules: an equality comparison between pointers A and B may be removed if A is dereferenced after B is passed to `free()`.
- Const qualification: An equality comparison between pointers A and B may be removed if A is known to point to an object declared as const, and *B is modified.
- Integer overflow: A comparison to INT_MAX can be removed if the integer is subsequently incremented.

Guarantees might be provided arbitrarily

Consider this function:

```
std::atomic<long long> x;
std::atomic<int> y;

void t1() {
  long long r1 = x.load(std::memory_order_relaxed);
  if (r1 % 4 == 0) {
    y.store(1, std::memory_order_relaxed);
  }
}
```

I think it will be mostly unobjectionable that a clean OOTA fix should establish some sort of ordering between the load and the store.

```
std::atomic<long long> x;
```

```
std::atomic<int> y;
template <typename T>
void t1() {
  long long r1 = x.load(std::memory_order_relaxed);
  if (r1 % 4 == 0) {
    y.store(1, std::memory_order_relaxed);
  }
  *(T*)r1 = *(T*)r1;
}
```

Here though, I don't know that we can get an unconditional guarantee. The implementation may infer from the load that `r1` must be as aligned as `alignof(T)`. So in this case, `t1<char>` and `t1<short>` might establish some sort of OOTA-preventing ordering between the load and store, while `t1<int>` would not (assuming reasonable values for the relevant alignments). This is ugly, but I'm not sure it's preventable without invalidating single-threaded optimizations (note that in general the atomic operations might occur someplace the compiler can't see, while the control dependency elimination relies only on non-atomic operations).

As described so far, this is the same issue as in the previous section; an unaligned pointer dereference is UB, so the compiler infers that the (pre-casted) pointer value must be properly aligned, and so eliminates the check. But, what if instead of `r1 % 4`, we had `r1 % (2 * alignof(T))`? The pointer dereference doesn't (as a language rule) require *over*alignment. But imagine that, for some type `T`, the compiler can prove that it sees all the places where a `T` object is constructed (say, because `T` is local to a particular function). Based on some black-box optimization heuristic, it decides that it would prefer to always overalign `T` objects, and to rely on that over-alignment when accessing them (perhaps it determines that some member of `T` could benefit from using SIMD instructions). It's perfectly valid to do so, and, having overaligned all its `T` objects, it's valid for it to generate code that assumes overalignment. The check could then be eliminated, and no amount of static or dynamic work could inform the programmer whether the ordering that seems present genuinely is.

What might an inelegant fix look like?

These issues have obvious similarities to those we've seen with `memory_order_consume`. P0750 suggested fixing those with a class that explicitly tracks dependencies. This works even more straightforwardly for an OOTA fix, which does not need to concern itself with minimizing syntactic overheads for pointer chasing.

Once we reify the notion of a dependency, we can ban cycles in `rf U sb` in which the store has a dependency passed from the load. We can also add a fence type taking a dependency as an argument, which bans the cycles in `rf U sb` where both a load and a fence taking its dependency from that load participate in the cycle:

```
std::atomic<int> x, y;

void f() {
  std::atomic_dependency d;
  int r1 = x.load(std::memory_order_relaxed, d);
  int r2 = r1 ^ r1; // A syntactic but not a semantic dependency.
  int r3 = r1 + 1; // A true dependency, both syntactic and semantic.
  if (r2 == 0) {
    // OOTA outcomes allowed
    y.store(1, std::memory_order_relaxed);
  }
  if (r3 == 0) {
    // OOTA outcomes allowed, despite the true dependency. We can still
    // keep the current informal recommendation, though.
    y.store(1, std::memory_order_relaxed);
  }
  if (r2 == 0) {
    y.store(1, std::memory_order_relaxed, d); // No OOTA.
  }
  if (r2 == 0) {
    std::atomic_dependency_fence(d);
    y.store(1, std::memory_order_relaxed); // No OOTA.
  }
}
```

In this phrasing, the implementation of `memory_order_load_store` is a relaxed-load-with-dependency followed by a dependency fence. Explicitly tracking the dependencies would be more optimizable, but also more verbose, than making every relaxed operation `memory_order_load_store`.