Google

Using Weakly Ordered C++ Atomics Correctly

Hans-J. Boehm

Why atomics?

Programs usually ensure that memory locations cannot be accessed by one thread while being written by another. “No data races.”
Typically we acquire locks when accessing shared variables:

```
void inc_shared() {
    lock_guard<mutex> _(mtx);
    x++;
}
```

+: Operates at different granularities: Can correctly handle complex operations.
-: Lock ordering/deadlock concerns.[1] Doesn’t work in signal/interrupt handlers.
Performance may be suboptimal. Preempted thread can block others.

[1] See also transactional memory specification.

Why atomics? (2)

Pre-C++11 experience:

- If all accesses need to be protected by locks, programmers invent ways to evade the rules:
  - Usually dubious ones:
    - `volatile` + assembly code
  - Sometimes terrible ones:
    - Racing accesses that invalidate compiler assumptions.
  - Gains are sometimes imagined, sometimes real. (See Fedor Pikus' talk.)

C++ atomics design

Declaring a location as `atomic<T>` allows concurrent access.
Individual operations on `atomic<T>` appear indivisible:

- Another thread sees them as completed or not done at all.
- By default, you get interleaving “sequentially consistent” semantics
  - so long as there are no data races (no non-atomic location is accessed while being modified)[1]
- It appears that at each time step an arbitrary thread performs next operation.

[1] And you avoid some really silly things. Like implementing Dekker’s mutual exclusion algorithm using `std::get_terminate()` and `std::set_terminate()`.

C++ sequentially consistent atomics

- Behave as though a mutex were protecting a single operation.
- As easy to use as a mutex,

  but only if all protected critical sections naturally contain a single operation.
- A bit faster than a mutex

  in these easy cases.

On the other hand:

- There is much research on lock-free algorithms.
- Its goal is to decompose more complex atomic operations into such single variable atomic operations, preserving appearance of atomicity.
- This is *hard*. Many research publications have been buggy.

Requirements on sequentially consistent atomics

Very roughly, sequentially consistent atomics need to ensure that:

I: Operation is indivisible.

Usually free for loads and stores of small, well-aligned, operands.

S: Stores become visible to other threads after prior memory operations.

Basically free on x86.

Requires “memory fence” a.k.a. “memory barrier” on ARMv7.

L: Loads must complete before subsequent memory accesses take effect.

Basically free on x86.

Requires memory fence on ARMv7.

SL: atomic stores are not reordered with subsequent atomic loads.

Requires memory fence on ARMv7 and x86. Universally expensive.

Sequentially consistent atomics, MessagePassing example

```
int x;
atomic<bool> x_init(false);
```

Thread 1:

```
x = 17;
x_init = true;
```

Thread 2:

```
if (x_init) {
    assert(x == 17);
}
```

- Ensures that `x = 17` is visible to `assert` call.
- Prevents reordering of expressions in either thread.
- Needs S & L (& I), not SL

Sequentially consistent atomics, Dekker's example

```
atomic<bool> x(false), y(false);
```

Thread 1:

```
x.store(true);
if (!y.load())
  turn_EW_lights_green();
```

Thread 2:

```
y.store(true);
if (!x.load())
  turn_NS_lights_green();
```

- Ensures that one of the store calls goes first, and is visible to other thread.
- Prevents store → load reordering in either thread.
- Needs SL.

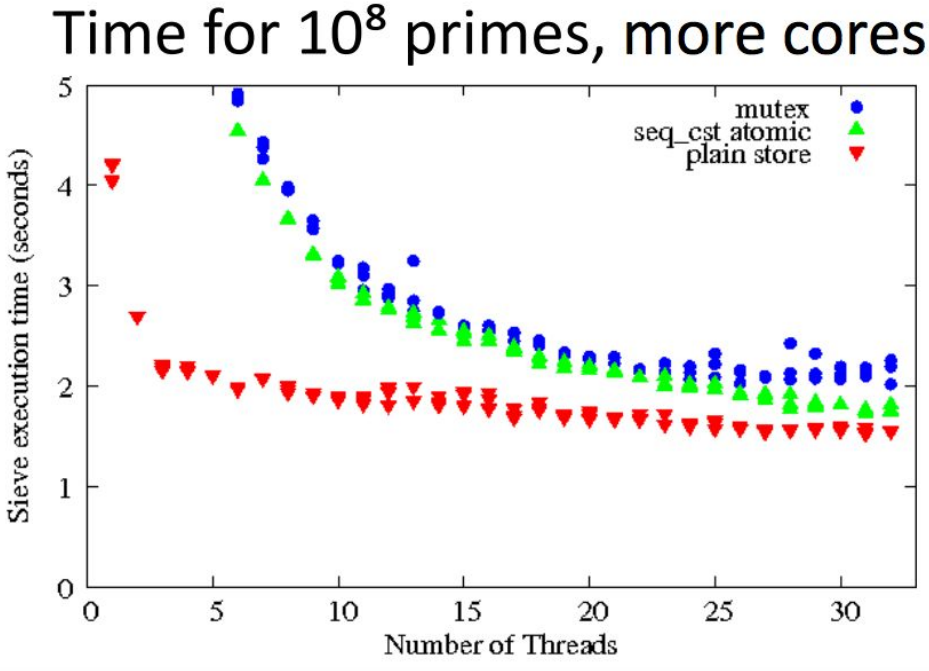
The cost of sequentially consistent atomics

- On simple microbenchmarks, a fence typically costs 2 to 200 cycles.
  - 20-30 typical on modern processors?
  - Cheap compared to cache miss, e.g. memory contention between threads.
  - Expensive compared to most instructions.
  - Dominates atomics cost in the absence of contention.
- Significant factor in mutex cost.
- Can easily make clever lock-free code slower than locking.
- On x86, a fence is only needed for stores.
  - Loads are sufficiently ordered anyway.
  - Atomic RMW operations already include the fence.
- On ARMv7, Load: one fence, store or RMW: two fences.
- ARMv8 essentially has instructions for SC atomics.

Important note on sequential consistency costs

In my experience:

- Extra fences usually add thread-local overhead.
- Overhead tends to be hidden by any other contention.
- Scalability may “improve”; performance will decrease.



ACM RACES ‘12

Reducing the cost: Weakly ordered atomics

Cost of sequentially ordered atomics is not always necessary.
C++ provides weakly ordered atomics, that sacrifice ordering guarantees.

`memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`:

- Sacrifice SL ordering.
- Avoids fence after store, allows some fences to be weakened.

`memory_order_relaxed`:

- Sacrifice SL, S, L ordering (leaving only I).
- Accesses to the same location can still not be reordered.
- Saves fences before store, after load, on ARM.

The ugly side of weakly ordered atomics

Extreme complexity.

- The rules are not obvious.
- They’re often downright surprising.
- And not even well understood.
- The committee still hasn’t figured out how to define `memory_order_relaxed`.

… and I’m not even going to talk about `memory_order_consume`.

And use of weakly ordered atomics in a library is often visible to clients.

- It doesn’t hide well.

My advice

If possible, use mutexes (or possible transactional memory).

- (Relatively) simple atomicity at any granularity.

If all critical sections accessing a variable involve a single access:

- Use sequentially consistent atomics.

If you need variable access from signal handlers:

- Very carefully use sequentially consistent atomics.

If you measured and the result is really too slow:

- Consider more complicated lock-free algorithms.
  - Remembering that this may not help.
- Use weakly ordered atomics.
  - Remembering that they’re a bug magnet.

Rest of talk

1. Weakly-ordered atomics pitfalls
2. Weakly-ordered atomics recipes

Atomics: Pitfall 1

```
atomic<int> x;
```

`x = x + 1;` is not the same as `x++;`!

- Only individual accesses are atomic!
- The former is not a single access!
- Only individual `atomic<T>` member function calls count as single accesses.
- *Writing code so that multiple atomic accesses appear indivisible is hard!*

Weakly ordered atomics: Pitfall 1

Ordering constraints are subtle!

- See reference counting example later.
- In my experience, this is a VERY common source of errors.

Weakly ordered atomics: Pitfall 2

Subtle interaction with other atomics:

Thread 1:

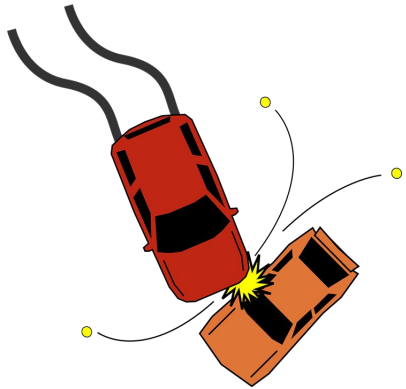
```
x.store(true, memory_order_release);
if (!y.load())
  turn_EW_lights_green();
```

Thread 2:

```
y.store(true, memory_order_release);
if (!x.load())
  turn_NS_lights_green();
```

`release` stores can be reordered with `seq_cst` loads!

SL reordering guarantee only applies to `seq_cst` ops!



Weakly ordered atomics: Pitfall 3

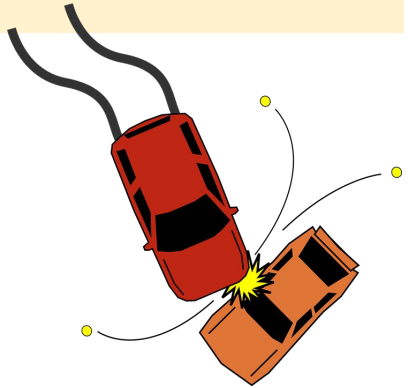
Subtle interaction with locks:

Thread 1:

```
x.store(memory_order_relaxed);
{ lock_guard<mutex> _(m1); }
if (!y.load(memory_order_relaxed))
  turn_EW_lights_green();
```

Thread 2:

```
y.store(memory_order_relaxed);
{ lock_guard<mutex> _(m2); }
if (!x.load(memory_order_relaxed))
  turn_NS_lights_green();
```



Critical sections do not act as a memory fence!
Memory model allows movement *into* critical sections.
(Does work if both both mutexes are the same.)

Pitfalls 2 and 3 are getting bigger:

These have traditionally worked because:

- Compilers did not reorder across synchronization.
- Synchronization operations where usually implemented as hardware fences.

Both are now false!

- Compilers always get more aggressive.
- ARMv8 provides non-fence-like ordering primitives. Your phone probably uses them.

Weakly ordered atomics: Pitfall 4

“Dependencies” do not enforce ordering w.r.t. threads:

```
ptr = x.load(memory_order_relaxed);
if (ptr == y) {  // hardware predicts condition true;
    result = ptr->field.load();  // compiler transforms to y->field
    // Both loads issued concurrently; second may complete first.
}
```

General principles for weakly ordered atomics

Acquire release:

- Ensure that memory operations preceding a release store are visible to (*happen before*) code reading the stored value with an acquire load.
- Other reasoning about operation ordering remains invalid.

Relaxed:

- Only ensures ordering for operations on that one memory location.
- Cannot assume anything about the rest of state from loaded value.
- (Currently makes code resistent to formal verification.)

Consume:

- Avoid (for now).

Some correct use recipes

- General use of weakly ordered atomics is hard.
- You need to fully understand the memory model!
- But here are some common cases.

Single word data structures

If the contents of a single-word data structure are not relied upon for other computations while it is being modified, it's OK to use `memory_order_relaxed`.

Example:

1. Counter that's only read after threads join. Use
   ```
   counter.fetch_add(1, memory_order_relaxed);
   ```
2. Accumulate information in a short bit vector. E.g.
   ```
   bit_set.fetch_or(1 << new_element, memory_order_relaxed);
   ```

Warning sign: Caring about the result.

Computing a guess for compare_exchange

Sometimes the value of a load just doesn't affect correctness:

```
old = x.load(memory_order_relaxed);
while (x.compare_exchange_weak(old, foo(old)) {}
```

This would remain correct if we replaced the first line by `old = 42;`
We're clearly not relying on the load value to tell us anything about the state.

Non-racing accesses

Some accesses to `atomic<T>` just don't need to be atomic.
Example: Double-checked locking:

```
atomic<boolean> x_init(false);
```

```
if (!x_init) {
    lock_guard<mutex> _(x_init_mtx);
    if (!x_init.load(memory_order_relaxed)) {
        initialize x;
        x_init = true;
    }
}
```

Simple one-way communication

```
int x;
atomic<bool> x_init(false);
```

Thread 1:

```
x = 17;
x_init.store(true,
             memory_order_release);
```

Thread 2:

```
if (x_init.load(memory_order_acquire) {
    assert(x == 17);
}
```

- Requires only that `x = 17`, from before the store, is visible after the `load`.
- No presumption that store must happen before something else.

Double-checked locking again

```
atomic<boolean> x_init(false);
```

```
if (!x_init.load(memory_order_acquire)) {
    lock_guard<mutex> _(x_init_mtx);
    if (!x_init.load(memory_order_relaxed)) {
        initialize x;
        x_init.store(true, memory_order_release);
    }
}
```

Requires only that the correct value of x be seen after an execution that sees `x_init == true`.

Implementing simple locks

Use standard lock if available!

If lock entry and exit require a single atomic operation, then it suffices to prevent critical section operations from moving *out* of the critical section:

(Warning: world's dumbest spin-lock!)

Lock acquisition:

```
while (lock.exchange(true, memory_order_acquire)) {}
```

Lock release:

```
lock.store(false, memory_order_release);
```

Some complex examples

- The next two require more complex reasoning, but are also fairly common.
- Deriving solutions like this requires thorough memory model understanding.
- Copying them not so much.

Reference counting

You should probably use `shared_ptr`, but …

Increment:

```
count.fetch_add(1, memory_order_relaxed);
```

Decrement:

```
if (count.fetch_sub(1, memory_order_acq_rel) == 1) deallocate;
```

`unique()` : memory_order_acquire ?

- Client ensures all objects are visible before final decrement.
- Each decrement ensures that prior operations are visible to next decrement.
- All operations happen before the final deallocation.

Note on reference counting

The usual recommendation is to instead use:

```
if (count.fetch_sub(1, memory_order_release) == 1) {
    atomic_thread_fence(memory_order_acquire);
    deallocate;
}
```

- This is correct for similar, but even more subtle reasons.
- A bit faster on ARMv7 and Power.
- Roughly the same on x86 and ARMv8.
- I just dislike fences.

Reading with a version counter (seqlocks)

Scenario: Data is written very rarely, but read frequently. Avoid locking for readers.

```
atomic<unsigned int> version(0);
```

Writer: `{ lock_guard _(m); version++;` *write data;* `version++; }`

Unoptimized reader:

```
do {
    unsigned int v1 = version; read data; unsigned int v2 = version;
} while ((v1 & 1) != 0 || v1 != v2);
```

Seqlocks continued

Problem:

- Data accesses race!
- Read values are discarded when they do.
- But still introduces undefined behavior. ⇒ Data must also be atomic!

Optimized reader:

```
do {
    unsigned int v1 = version.load(memory_order_acquire);
    foo_1 my_data_1 = data_1.load(memory_order_relaxed); ...
    foo_n my_data_n = data_n.load(memory_order_relaxed);
    atomic_thread_fence(memory_order_acquire);
    unsigned int v2 = version.load(memory_order_relaxed); // corrected after talk.
} while ((v1 & 1) != 0 || v1 != v2);
```

Conclusions

Weakly ordered atomics are hard!

Avoid when possible!

A small number of recipes seem to cover a significant fraction of use cases.

For anything else, you need to fully understand the memory model!

References:

- Fairly quick and slightly dated overview: WG14/N1479.
- Thorough mathematical discussion: Chapter 3 in Mark Batty's thesis.
- And there's always the standard itself.
  - But it's not clear this is best expressed in "standardese".