A Relaxed Guide to `memory_order_relaxed`

Hans Boehm
Google

Paul E. McKenney
Facebook

CPPCON 2020

`std::atomic`/`std::atomic_ref` and `memory_order_relaxed`

- C++ atomic operations are *sequentially consistent* by default.
  - Interleaving semantics is preserved if only the default is used.
- That is expensive
  - Especially on weakly-ordered architectures, such as ARM and Power
- And in many cases avoidable
  - by sacrificing the simple threads-as-interleaving semantics
- by passing `memory_order` enum values to explicit atomic operations.
- In particular, `memory_order_relaxed` allows arbitrary visibility reordering with respect to accesses to other locations.

What is Not to Like About `memory_order_relaxed`?

- Just a load, just a store: Full control, excellent efficiency and scalability!
  - Assuming aligned machine-sized atomic objects, that is…

What is Not to Like About `memory_order_relaxed`?

- Just a load, just a store: Full control, excellent efficiency and scalability!
  - Assuming aligned machine-sized atomic objects, that is…
- Small problems: Out of thin air (OOTA) & read from untaken branch (RFUB)
  - Small but persistent: Java has been attacking a similar problem for more than 20 years
  - Some progress within the past few years:
    - We now have ways of classifying OOTA vs. simple reordering!
      - One requires per-scenario creativity, others are also not suited for compilers
    - There is also one recent complication… With work towards a solution...
    - And we also have one pragmatic solution (formal variant)!
      - But incurs otherwise unnecessary overhead on architectures such as ARM
      - This refinement proposes the addition of `memory_order_load_store`

OOTA (Out-of-thin-air problem) Introduction

We contrast three examples:

- Simple reordering must be allowed. Occurs in practice.
- (OOTA) Not believed to occur in practice. Should be disallowed, but hard to do so. (2 variants)
- (RFUB) Very rarely occurs in practice. Controversial, but majority of SG1 leans towards allowing it.

Notation and conventions

- `x` and `y` denote potentially shared locations.
- `r1` and `r2` denote (not-address-taken) locals ("registers").
- All shared locations are presumed to be initially zero, null, or false, unless stated otherwise.
- $r1 =_{rlx} x$ abbreviates `r1 = x.load(std::memory_order_relaxed)`
- $x =_{rlx} r1$ abbreviates `x.store(r1, std::memory_order_relaxed)`

Must be allowed

```
atomic<int> x(0), y(0);
```

Thread 1:

// y = x;

int r1 $=_{rlx}$ x;

y $=_{rlx}$ r1;

reads from

reads from

Thread 2:

// x = 42;

int r2 $=_{rlx}$ y;

x $=_{rlx}$ 42;

r1 = r2 = 42 is fine!

OOTA: Should be disallowed

```
atomic<int> x(0), y(0);
```

Thread 1:

// y = x;

`int r1` $=_{\mathtt{rlx}}$ `x;`

`y` $=_{\mathtt{rlx}}$ `r1;`

Thread 2:

// x = y;

`int r2` $=_{\mathtt{rlx}}$ `y;`

`x` $=_{\mathtt{rlx}}$ `r2;`

reads from

reads from

r1 = r2 = 42 should be disallowed!

OOTA: Should be disallowed

```
atomic<int> x(0), y(0);
```

Thread 1:

```
// y = x;
   int r1 =rlx x; // guess 42
   y =rlx r1;
   // Confirm speculation
```

Thread 2:

```
// x = y;
   int r2 =rlx y; // guess 42
   x =rlx r2;
   // Confirm speculation
```

Formally each load observes the other thread's store, as before.

OOTA (variant B): Should be disallowed

```
atomic<unsigned int> x(0), y(0);

int a[1], b[1];
```

Thread 1:

```
int r1 =rlx x;

a[r1] = 42;
```

reads from

reads from

Thread 2:

```
int r2 =rlx y;

b[r2] = 42;
```

r1 = r2 = 42 should be disallowed!
But could have a + 42 = &y and b + 42 = &x

OOTA (variant B): Should be disallowed

```
atomic<unsigned int> x(0), y(0);

int a[1], b[1];
```

Thread 1:

```
int r1 =rlx x;  // Guesses 42

a[r1] = 42;// Out-of-bounds access
           // writes 42 to y.
```

Thread 2:

```
int r2 =rlx y;  // Guesses 42

b[r2] = 42;// Out-of-bounds access
           // writes 42 to x.
```

Assigns 42 to x and y if a + 42 = &y and b + 42 = &x

Outlawing OOTA

- Long-standing problem:
  - Correctly define and prohibit out-of-thin-air results.
  - Without prohibiting necessary reordering
- Java, C11, C++11 tried hard and failed.
- There are once again solutions on the table.
- Technically complicated.
- Under investigation by SG1 and many others.

Out-of-thin-air in C++14/17/20 standard

Defined only by example!

"Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation."

We knew this was really way too imprecise ...

Strict C++20 $=_{\mathrm{def}}$ C++20 without this hand-waving

Read-from unexecuted branch (RFUB)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
int r1 =rlx x;
y =rlx r1;
```

Thread 2:

```
bool assigned_42 = false;
r2 =rlx y;
if (r2 != 42) {
  assigned_42 = true;
  r2 = 42;
}
x =rlx r2;
```

Can result in
`x` = `y` = 42 and
`assigned_42` = false!

wg21.link/p1217 has details.

What is Not to Like About `memory_order_relaxed`?

- Just a load, just a store: Full control, excellent efficiency and scalability!
  - Assuming aligned machine-sized atomic objects, that is…
- Small problems: Out of thin air (OOTA) & read from untaken branch (RFUB)
  - Small but persistent: Java has been attacking a similar problem for more than 20 years
  - Some progress within the past few years:
    - We now have ways of classifying OOTA vs. simple reordering!
      - One requires per-scenario creativity, others are not suited for compilers
    - There is also one recent complication… With work towards a solution...
    - And we also have one pragmatic solution (formal variant)!
      - But incurs otherwise unnecessary overhead on architectures such as ARM
      - This refinement proposes the addition of `memory_order_load_store`
- In the meantime, where can `memory_order_relaxed` be used???

In the Meantime, Use Known Good Patterns

- If you randomly generate code using `memory_order_relaxed`, you will get what you deserve
- The same is also true of the other `memory_order` values, but with fewer counter-intuitive pitfalls
- Use `memory_order_relaxed` only in the context of known-good patterns!

When is memory_order_relaxed “fully safe”?

By “fully safe”, we mean that it’s correct in strict C++20.

This means:

- We can precisely reason about it, potentially even formally, without resulting to weasel-wording.
- Adding unexecuted code won’t enable RFUB behavior.

Some common memory_order_relaxed idioms are “fully safe”; other important ones require the weasel words.

Strict C++20: Simple Safe Use Cases

A great many memory_order_relaxed usages are safe even in Strict C++.

Easy examples:

- Atomic counters that are only read after all other threads have terminated.
    - At which time there are no racing stores, so the canonical OOTA pattern cannot form.
- Ordering is provided by atomic_thread_fence().
    - The atomic_thread_fence() calls prevent the canonical OOTA pattern from forming.
- There is only one shared object
    - The canonical OOTA pattern requires at least two shared objects

Safe in strict C++20: Unidirectional Data Flow

1. Get data from outside world

Thread 1:

```
s1 = get_ext_state(1);
s2 = get_ext_state(2);
cs1_rlx = reduce_state1(s1);
cs2_rlx = reduce_state2(s2);
```

2. Pure function numerically conditions data

3. Pure function computes control output

Thread 2:

```
c = compute_ctl(cs1_rlx, cs2_rlx);
set_ext_ctl(c);
```

4. Send control signal to outside world

Other examples include software pipelines and per-thread statistical counters.
But “unidirectional” can be a slippery concept: See upcoming reentrant mutexes.

What makes an idiom potentially unsafe?

Loading an atomic value with a relaxed load, and relying on that value for correctness, e.g. by

- Using it to determine the value stored into another atomic, or
- Relying on it to avoid disastrous misbehavior
  - … Which usually involves generating “undefined behavior"

This is surprisingly common!

Mostly due to relaxed loads that can lead to undefined behavior

… if they read a bad value

… which they shouldn’t

…

Canonical potentially unsafe pattern

Thread 1:

```
int r1 =rlx x;
if (r1 is bad)
   bad_behavior();
```

Thread 2:

```
int r2 =rlx y;
if (r2 is bad)
   bad_behavior();
```

reads from

reads from

Thread 1 and 2 speculatively read bad values for `r1` and `r2`.
Thread 1's bad behavior stores the value read by Thread 2 into `y`.
Thread 2's bad behavior stores the value read by Thread 1 into `x`.

Formally. each thread's relaxed load reads from a value written by `bad_behaviour()`.

Unsafe in strict C++20: Lazy idempotent scalar initialization

Thread 1:

```
long r1 =rlx x;
if (r1 == 0) {
  r1 = pure_function();
  x =rlx r1;
}
// r1 is now trusted.
if (r1 is bad)
  a[rand()] = rand();
```

Thread 2:

```
long r2 =rlx y;
if (r2 == 0) {
  r2 = pure_function();
  y =rlx r2;
}
// r2 is now trusted.
if (r2 is bad)
  b[rand()] = rand();
```

Unsafe in strict C++20: reentrant mutexes

```
class my_reentrant_mutex {
  std::mutex m;
  // Writes of owner are protected by m.
  // Only owner writes or clears its id.
  std::atomic<std::thread::id owner>; // id() if not held
  // Protected by m.
  int count;  // Held count-1 times by owner
  …
}
```

Unsafe in strict C++20: reentrant mutexes(2)

```
void my_reentrant_mutex::lock() {
  std::thread::id me = std::this_thread::get_id();
  // No other thread can change whether owner == me.
  if (owner.load(memory_order_relaxed) == me) {
    ++mutex.count; // Done; reacquired the lock.
  } else {
    ... // Acquire m, leaving count == 0
    owner.store(me, memory_order_relaxed);
  }
}
```

Unsafe in strict C++20: reentrant mutexes(2)

1. Already holds lock; enters c.s.

Thread 1:

```
if (owner_rlx == me) {
    ++mutex.count;
} else {
    ...
}
// Critical section:
++x;
if (x >= 2) a[rand()] = rand();
--x;
```

4. Enters c.s.

5.Overwrites owner, satisfying speculation.

2. Speculates it holds lock

Thread 2:

```
if (owner_rlx == me) {
    ++mutex.count;
} else {
    ...
}
// Critical section:
++x;
if (x >= 2) b[rand()] = rand();
--x;
```

3. Runs c.s. to here

Other important strict-C++20-unsafe examples

Implementing a concurrent non-conservative (e.g. Java) garbage collector.

- Client (“mutator”) accesses must be memory_order_relaxed for performance.
- Collector relies on pointer validity to avoid undefined-behavior.

“Chaotic relaxation” numerical algorithms with no synchronization between successive iterations.

- OOTA might create a NaN, which could propagate through the computation.

Safe in strict C++: Non-racing access

Double-checked locking

```
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);
    }
}
```

I hold the same lock as the only writer. No concurrent access.

Safe in strict C++: untrusted accesses

Initial load for compare-exchange

Result does not affect correctness!

```
unsigned long expected = x.load(memory_order_relaxed);
while (!x.compare_exchange_weak(expected, f(expected)) {}
```

Summary of Categories of Patterns (P2055R0)

	Multiple Threads	Concurrent WW	Concurrent RW	But Checked	But Discarded	But Fungible	Unordered Cycle	Strict C++ Safe
Non-Racing Accesses (Section 2.1)	Y							Y
Single-Location Data Structures (Section 2.2)	Y	Y	Y					Y
Shared Fences (Section 2.3)	Y	Y	Y					Y
Atomic Reference-Count Updates (Section 2.4)	Y	Y	Y			Y		Y
Untrusted Loads (Section 2.5)	Y		Y	S	S	S		Y
Unidirectional Data Flow (Section 2.6)	Y	Y	Y					Y
Reentrant Mutex (Section 2.6.5)	Y		Y				Y	
Java-Style Lazy Scalar Initialization (Section 2.7)	Y	Y	Y			Y		
Chaotic Relaxation (Section 2.8)	Y	Y	Y	Y			Y	S
Garbage Collection (Section 2.9)	Y	Y	Y				S	S

“Yes”

“No”

“Sometimes”

Summary

Using `memory_order_relaxed` can be tricky because we don't yet know an efficient way to formally define the boundaries of OOTA and RFUB.

- Important use cases of memory_order_relaxed work in practice, but some resist attempts at a precise correctness argument.
- We define "strict C++20" as that portion of C++20 that excludes the vague normative encouragement to avoid OOTA.
- Good news: Many common `memory_order_relaxed` usage patterns are demonstrably correct even in strict C++20.

Questions?

References:

Boehm & McKenney, "P2055: A Relaxed Guide to memory_order_relaxed"

We recklessly assumed: Boehm, ""P2215: Undefined behavior" and the concurrency memory model"

OOTA, RFUB background: Boehm, "P1217: Out-of-thin-air, revisited, again", and David Goldblatt, "P1916: There might not be an elegant OOTA fix"

Closely related: "quantum atomics" in Sinclair, Alsop, and Adve, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems"

A proposal to fully define memory_order_relaxed: Batty et al, "P1780: Modular Relaxed Dependencies: A new approach to the Out-Of-Thin-Air Problem", "Lee et al, Promising 2.0: global optimizations in relaxed memory concurrency"