P2055R1: A Relaxed Guide to `memory_order_relaxed`

Hans Boehm
hboehm@google.com

David Goldblatt
davidtgoldblatt@gmail.com

Paul E. McKenney
paulmck@kernel.org

The Indefatigible TBD

April 8, 2024

Audience: SG1 (Informational)

Abstract

The out-of-thin-air (OOTA) and read-from-untaken-branch (RFUB) properties of the specification of `memory_order_relaxed` have resulted in considerable consternation over the years. Although there are no known instances of full-blown OOTA behavior, and no known RFUB-induced failures of production code, the theoretical possibility of these properties severely complicates automated analysis of large C and C++ code bases. Thus far, attempts to eliminate OOTA and RFUB properties from the memory model have resulted in otherwise needless added overheads on weakly ordered systems on the one hand or excessive implementation complexity on the other. However, `memory_order_relaxed` never was intended to be used in arbitrary code, but rather as a part of deliberate application of specific concurrency designs. This paper forms an initial catalog of patterns underlying such designs, with an eye towards identifying bugs in existing code and preventing bugs in new code.

1 Background

1.1 Memory Ordering Strength and Expense

C++ atomic operations are sequentially consistent by default. In programs that use only sequentially consistent atomic operations, these atomics enjoy mathematically appealing interleaving semantics. However, no computer system in common use in the year 2023 provides sequential consistency at the hardware level, not even relatively strongly ordered systems such as x86. This means that sequentially consistent atomics can be quite expensive, and especially on weakly ordered systems such as ARM and IBM Power. Furthermore, a great many use cases in real-world software do not require sequentially consistent atomics, in fact, the much lower-overhead release/acquire ordering suffices in a great many situations.

For this reason, C++ provides `memory_order enum` values to allow the user to control the strength of individual atomic operations. The `memory_order_release` and `memory_order_acquire` values provide the aforementioned release-acquire ordering.

But most relevant to this document, `memory_order_relaxed` allows arbitrary reordering of accesses to distinct locations. With the exception of the unlamented Itanium CPU family, aligned and machine-word-sized `memory_order_relaxed` loads and stores compile to unadorned load and store instructions, providing complete control, excellent efficiency, and stunning scalability.

And in practice, `memory_order_relaxed` accesses provide well-understood ordering properties, which has led to them being heavily used. Unfortunately, in theory `memory_order_relaxed` accesses are subject to OOTA. RFUB can occur very rarely in practice, and there is some debate as to whether this is acceptable.

These issues are discussed in the following section.

1.2 OOTA and RFUB

There has been considerable work done on OOTA and RFUB over many years, building on prior work in the Java community, perhaps best exemplified by the infamous "Causality Test Cases".[1] There has long been hope that additional research effort will identify a model of OOTA that all can live with, for example, on the part of Paul, and that everyone would come to appreciate the relative simplicity of strengthening `memory_order_relaxed` to forbid prior reads to be reordered with later writes, for example, on the part of Hans [7, 6, 17]. And progress has been made on both fronts.

On the additional-research front, we now have methods of distinguishing between OOTA on the one hand and simple reordering on the other. Unfortunately, one method requires per-scenario creativity [21], while others have not as yet been looked upon with favor by compiler implementers [17, 27, 18, 3]. Backwards-propagating undefined behavior can be especially troublesome [14], but there proposals that this be restricted [4]. Perhaps such restrictions might eliminate some of the more troublesome examples of OOTA.

On the reads-before-writes front, there are some indications that newer weakly ordered hardware incurs reduced penalties for ordering relaxed reads before relaxed writes, at least for low-end and high-end systems. However, middle-end systems still incur significant penalties. This has led to renewed suggestions that a new `memory_order_load_store` member be added to the `memory_order enum`. It has also led to a renewed proposal that `memory_order_relaxed` be strengthened so as to prohibit reordering of prior loads and subsequent stores [12], however, this proposal was not universally loved [15].

A further complication is that although there is general agreement that OOTA behaviors must be forbidden, there is some debate on the need to forbid RFUB behaviors. Some of those who believe that RFUB behaviors should be allowed (for example, Paul) argue that the scenarios where RFUB can occur are contrived and with no known

[1] `http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html`.

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   unsigned long r1 = x.load(memory_order_relaxed);
 7   y.store(r1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   unsigned long r2 = y.load(memory_order_relaxed);
13   x.store(42, memory_order_relaxed);
14 }
```

Listing 1: Simple Reordering

production use cases. There is general agreement that simple reordering should be allowed, as it occurs in practice in scenarios that are both useful and commonplace.

1.3 OOTA Examples

This section looks at examples of simple reordering, OOTA, and RFUB.

Listing 1 shows an example of simple reordering. Both the compiler and the CPU are within their rights to reorder lines 12 and 13, in which case the following sequence of events will result in all of `x`, `y`, `r1`, and `r2` having the value 42:

1. Line 13 stores 42 to `x`.
2. Line 6 loads 42 from `x` into `r1`.
3. Line 7 stores `r1`, and thus 42, to `y`.
4. Line 12 loads 42 from `y` to `r2`.

It is strongly and generally agreed that this simple reordering be allowed.

Listing 2 shows an OOTA example that should be prohibited. where where all of `x`, `y`, `r1`, and `r2` have final values of 42: In actual hardware, this prohibition is enforced by TSO ordering in strongly ordered systems and by data dependency ordering in weakly ordered systems. Compilers cannot store something until after they load it, which results in instructions being emitted such that the hardware enforcement applies. The need to prohibit simple OOTA is one reason why compiler-based value speculation optimizations are frowned upon.

Listing 3 shows a simple example of RFUB, which again means “read from untaken branch”. The following sequnece of events will result in all of `x`, `y`, `r1`, and `r2` having the value 42:

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   unsigned long r1 = x.load(memory_order_relaxed);
 7   y.store(r1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   unsigned long r2 = y.load(memory_order_relaxed);
13   x.store(r2, memory_order_relaxed);
14 }
```

Listing 2: Simple OOTA

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   unsigned long r1 = x.load(memory_order_relaxed);
 7   y.store(r1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   bool assigned_42 = false;
13   unsigned long r2 = y.load(memory_order_relaxed);
14   if (r2 != 42) {
15     assigned_42 = true;
16     r2 = 42;
17   }
18   x.store(r2, memory_order_relaxed);
19 }
```

Listing 3: Simple RFUB

1. The compiler notices that at line 17, the value of `r2` is always 42.
2. The compiler thus substitutes the value 42 for `r2` in line 18.
3. Both the compiler and the CPU are within their rights to reorder lines 13 and 18.
4. Line 18 stores 42 to `x`.
5. Line 6 loads 42 from `x` into `r1`.
6. Line 7 stores `r1`, and thus 42, to `y`.
7. Line 13 loads 42 from `y` to `r2`.
8. Because `r2` is equal to 42, lines 15 and 16 are never executed, even though it was exactly these lines of code that justified the above compiler optimizations.

Many concurrency experts believe that RFUB should be allowed.

1.4 Classification of Patterns

Perhaps agreement on all of these points will be reached, but in the meantime, `memory_order_relaxed` use is increasing, and thus an increasing need to identify known-safe usage patterns. In the best case, these usage patterns might be automatically checked in existing code, but at a minimum we hope that this list will be useful to code reviewers. Either way, the goals are to identify bugs in existing code and to help avoid bugs in new code.

This paper is a first step toward such a set of patterns.

The term *full C++* refers to the C++20 memory model as stated in the current draft. The term *strict C++* refers to the subset of full C++ obtained by dropping the following normative encouragement from the C++20 memory model: "Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation." Some (but not all) of the proto-patterns in this document are safe in strict C++, but all of them are safe in full C++.

2 Relaxed Design Patterns

Many of these patterns are taken from Hans's `memory-model-design` posting on September 4, 2018.[2]

In the examples, `x` and `y` denote potentially shared locations, while `r1` and `r2` denote local variables ("registers") whose addresses are not taken.

[2] Message-ID: <CAMOCf+jchGw6DeE2NyCJA3wfFbNH-WFn59JruZPSWt9_jPW9NQ@mail.gmail.com>.

```
 1 int x = 0;
 2 int y = 0;
 3
 4 void thread1()
 5 {
 6   if (x)
 7     y = 1;
 8 }
 9
10 void thread2()
11 {
12   if (y)
13     x = 1;
14 }
```

Listing 4: Non-Atomic Accesses Sometimes Respect Control Dependencies

2.1 Non-Racing Accesses

Any non-racing access to an atomic object can be a relaxed access. Because the access is not concurrent with a conflicting access (store against either store or load), further ordering is unnecessary.[3] In fact, such accesses can in theory be non-atomic. In environments where atomicity is controlled by the access rather than the object definition, such accesses are often non-atomic in practice [1].

For example, given concurrent execution of `thread1()` and `thread2()` in Listing 4, the only permitted outcome results in both `x` and `y` being equal to zero in both full C++ and strict C++. Any other outcome would violate the "sequential consistency for data race free programs" principle, and must effectively be due to a compiler-created data race, which is forbidden.

In contrast, in the analogous program using C++ atomics (see Listing 5), additional behaviors are permitted by strict C++, including the one resulting in the final values of both `x` and `y` being `1`. The restriction to "strict C++" is important because this code fragment is considered to be an example of the OOTA behavior that is forbidden by the normative encouragement in that same standard.

In short, although any non-racing access to an atomic object may be relaxed, strict C++ counter-intuitively classifies many access patterns as racy.

Relaxed atomics can nevertheless be useful for non-racing accesses in real-life situations, as can be seen in the double-checked locking example shown in Listing 6. Line 1 uses an acquire load from `x_init` to check whether initialization is needed, in which case line 2 acquires `mutex` using an anonymous `lock_guard` (anonymity being designated by the underscore where a local-variable name would be expected). Once this lock is held, line 3 uses a relaxed load to recheck `x_init` to see whether initialization is still needed. A relaxed load works here because holding the lock prevents other threads from storing to `x_init`. Line 4 carries out the initialization, and line 5

[3] This covers case #8 in Hans's September 4, 2019 email.

```
 1 std::atomic<int> x = 0;
 2 std::atomic<int> y = 0;
 3
 4 void thread1()
 5 {
 6   if (x.load(memory_order_relaxed))
 7     y.store(1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   if (y.load(memory_order_relaxed)
13     x.store(1, memory_order_relaxed);
14 }
```

Listing 5: Strict C++ Does Not Require Atomics to Respect Control Dependencies

```
1 if (!x_init.load(memory_order_acquire)) {
2   lock_guard<mutex> _(x_init_mtx);
3   if (!x_init.load(memory_order_relaxed)) {
4     initialize(&x);
5     x_init.store(true, memory_order_release);
6   }
7 }
```

Listing 6: Double-Checked Locking

updates `x_init` to indicate that initialization is complete.

When the acquire load on line 1 returns the value `true`, that load synchronizes with the release store on line 5, guaranteeing that any code following line 7 sees the results of that initialization.

These patterns can be at least partially checked using data-race detectors, both static and runtime.

2.1.1 Initialization and Cleanup

Important special cases of this pattern are the single-threaded initialization and cleanup phases of an otherwise concurrent program. These use cases are one motivation for the strong ordering guarantees of thread creation and destruction. These guarantees permit the single-threaded initialization and cleanup code to run race free, with no need to consider interference from the intervening code that runs multithreaded.

2.1.2 Lock-Based Critical Sections

Exclusive locks provide mutual exclusion, so that objects accessed only while holding a given lock may be accessed using `memory_order_relaxed` accesses, or, for that matter, using non-atomic accesses.

Reader-writer locks provide a weaker form of mutual exclusion, However, objects that are updated only while a given reader-writer lock is write-held and read only when that same lock is either read-held or write-held may also be accessed using `memory_order_relaxed` accesses, or, again, using non-atomic accesses.

Of course, non-atomic accesses are almost always used with pure locking. However, `memory_order_relaxed` accesses are sometimes quite useful, for example, in cases where objects pass through a software pipeline, where one stage uses pure locking and another stage relies on atomic operations.

2.2 Single-Location Data Structures

Relaxed atomic operations provide sequentially consistent access to a single object. This means that data structures that fit into a single object can be accessed with relaxed atomics with no possibility of OOTA or RFUB behavior.

Note well that a group of single-location data structures might well interact in a way that could raise the spectre of OOTA or RFUB. As before, design review should therefore pay careful attention to information flow.

These patterns can be checked by verifying that no store to another shared variable is affected by the value of the single-location data structure, unless that value can be shown not to affect that same single-location data structure, for example, if that other shared variable is part of a unidirectional data flow (see Section 2.6).

2.3 Shared Fences

The `atomic_thread_fence()` function can be used to order multiple accesses.

For example, consider a series of acquire loads that are intended to provide order against subsequent accesses, but not against each other. Because the compiler will normally not be able to determine that the acquire loads need not be ordered against each other, on some platforms this will result in a memory-fence instruction being emitted after each and every acquire load, when only the last fence is required. These unnecessary fences can be avoided by replacing the acquire loads with relaxed loads followed by a single `atomic_thread_fence(memory_order_acquire)` [26, Section 4.1]. This will have the desired effect of ordering all of these loads with any subsequent accesses, while also avoiding the overhead of the redundant fence instructions that would be expected from the acquire loads.

Similarly, consider a series of release stores that are intended to provide order against prior accesses, but not against each other. Again, the compiler might emit a memory-fence instruction before each of the stores, when only the first fence is required. These unnecessary fences can be avoided by replacing the release stores with relaxed stores preceded by a single `atomic_thread_fence(memory_order_release)` followed by a series of relaxed stores [26, Section 4.2]. This will have the desired effect of ordering all of these stores with any prior accesses, while also avoiding the overhead of any redundant fence instructions emitted for the release stores.

In many cases, other ordered atomic operations may be substituted for the `atomic_thread_fence()` operations. For example, `synchronize_rcu()` (also known as `rcu_synchronize()` in recent C++ working papers) implies the semantics of `atomic_thread_fence(memory_order_acqrel)`. In addition, thread creation and thread join provide the ordering semantics of:

- `atomic_thread_fence(memory_order_release)` for thread creation.
- `atomic_thread_fence(memory_order_acquire)` for the initialization of the corresponding thread.
- `atomic_thread_fence(memory_order_release)` for the termination of the corresponding thread.
- `atomic_thread_fence(memory_order_acquire)` for the join operation of the corresponding thread.

It is of course more conventional to consider thread creation to synchronize with the created thread and thread termination to synchronize with the corresponding join operation. This leads to the final example, which is that the shared-fences pattern also applies to any pair of calls to library where one synchronizes with the other.

In this design pattern, OOTA and RFUB behaviors are ruled out by the semantics of `atomic_thread_fence()` or by synchronizes-with, as the case may be.

This pattern can in theory be checked by verifying that all accesses are associated with a fence, however, current checker technology likely requires that the associated fence be explicitly called out.

```
1 unsigned long expected = x.load(memory_order_relaxed);
2 while (!x.compare_exchange_weak(expected, f(expected))
3   continue;
```

Listing 7: Untrusted Load Checked by CAS

2.4 Atomic Reference-Count Updates

In certain reference-count use cases, the ordering of the increments and decrements is irrelevant. One common case is where it is only legal to increment the reference count when the incrementing thread already holds a reference, in which case the count cannot possibly decrease to zero in the meantime. Because only the one-to-zero transition requires ordering, reference-count increments can be relaxed in cases where another reference is guaranteed to be held throughout.

Similarly, reference-count decrements can also be relaxed, but only if the thread will still hold at least one reference after the decrement. In other words, a thread releasing its last reference is forbidden from using a relaxed operation to do so, because in that case there is no guarantee that another reference is guaranteed to be held throughout.[4]

This pattern can be checked in conjunction with lock dependency checkers such as the Linux kernel's lockdep facility [11].

We suspect that this is an example of a more general class of patterns, but other examples of such a class do not immediately come to mind. One can of course imagine things like preprocessed sensor values where these values are irrelevant except in their relation to cutoff values. We would welcome examples used in actual code.

2.5 Untrusted Loads

In many cases, it is acceptable for a load from an atomic shared variable to occasionally return random bits because the value is checked by some later operation. In such cases, the load can be a relaxed load.

Listing 7 demonstrates this pattern. If misordering, OOTA, or RFUB were to cause line 1 to return a bogus value, then the `compare_exchange_weak()` on line 2 would fail, implicitly re-loading the value and retrying.

2.5.1 Pre-Load for Compare and Swap

Perhaps the most well-known later checking operation is a non-relaxed compare-and-swap (CAS). The `atomic_compare_exchange_*()` family of read-modify-write CAS operations are typically used in a loop, and often require an initial load prior to the first pass through that loop. For non-relaxed CAS operations, this initial load can typically be a relaxed load, with the CAS operation's ordering preventing OOTA and RFUB behaviors. Relaxed CAS operations need to be part of some other design pattern

[4] More elaborate variants of this pattern allow these rules to be relaxed. For example, if a parent thread is guaranteed not to release its last reference until after joining with its child threads, then those child threads may use relaxed decrements to release their final reference.

(for example, the shared fences pattern called out in Section 2.3) if cycles containing them are to be guaranteed to be OOTA/RFUB-free in conjunction with an initial relaxed load. One common design pattern is the single-location data structure discussed in Section 2.2.

Additional examples are presented by Sinclair et al. [27].

This pattern can be checked by verifying that the values from the relaxed loads propagate only to a CAS operation.

2.5.2 Sequence Locking

The accesses within a sequence-locking read-side critical section can used relaxed loads because any concurrency with the corresponding update will result in a retry, thus discarding any loaded values. Assuming that sequence-locking readers never store to shared memory, this not only prevents the surfacing of any OOTA or RFUB cycles, but also of any other non-SC behaviors.

Note that a proposal[5] provides an `atomic_load_per_byte_memcpy()` that allows safe non-atomic access to data for sequence-lock readers, as well as an `atomic_store_per_byte_memcpy()` to update that same data by sequence-lock updaters. It is nevertheless quite possible that some sequence-lock readers might continue to use relaxed atomics in order to permit reliable computations within readers in the presence of data objects having trap representations.

Furthermore, sophisticated sequence-locking use cases may need to use relaxed accesses for other reasons. For example, the Linux kernel's lockless path-to-`inode` traversal uses the closely related sequence counters to detect large-scale changes to the filesystem tree that would otherwise confuse this traversal [9, 10]. Such confusion could result is a number of anomalies, including successful lookup of paths that never actually existed.

This pattern can be checked by enlisting the aid of lock dependency checkers to verify that the access is within the scope of a sequence lock reader. Checking that the value does not leak out of that sequence lock is more difficult.

2.6 Unidirectional Data Flow

If data flows only in one direction, then OOTA cycles cannot form. The following sections give several examples of this general design pattern.

2.6.1 Independent Input Data

Input data consisting of independent objects may be read using relaxed accesses because these objects are not affected by downstream computations. Here input data is defined broadly, including:

1. Measurements of outside environmental conditions.
2. Device configuration data.

[5] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1478r0.html`

```
 1 atomic<int> s1(0);
 2 atomic<int> s2(0);
 3
 4 void thread1()
 5 {
 6   int s1 = get_ext_state(1);
 7   int s2 = get_ext_state(2);
 8   cs1.store(reduce_state(s1), memory_order_relaxed);
 9   cs2.store(reduce_state(s2), memory_order_relaxed);
10 }
11
12 void thread2()
13 {
14   int c = compute_ctl(cs1.load(memory_order_relaxed,
15                        cs2.load(memory_order_relaxed));
16   set_ext_ctl(c);
17 }
```

Listing 8: Unidirectional I/O Data Flow

3. Software configuration data.
4. Security policies.
5. Network routing information.

The key point is that the concurrent-computation portion of application references but does not modify this data, and that there are no object-to-object consistency constraints.

2.6.2 Independent Output Data

Similarly, output data consisting of independent objects may be written using relaxed accesses because these objects do not affect upstream computations. As before, output data is defined broadly, including:

1. Control of objects external to the computer.
2. Many classes of debug output.
3. Some use cases involving video frame buffers.
4. Some use cases involving communication to a later stage of a software pipeline.

Similar to the independent input data discussed in the preceding section, the key point is that the concurrent-computation portion of application modifies but does not reference this data, and that there are no object-to-object consistency constraints.

```
 1 StatCounter<unsigned long> a;
 2 StatCounter<unsigned long> b;
 3
 4 void thread1()
 5 {
 6   unsigned long r1 = a.readout();
 7   b.increase(r1);
 8 }
 9
10 void thread2()
11 {
12   unsigned long r2 = b.readout();
13   a.increase(r2);
14 }
```

Listing 9: Statistical-Counter Abuse and OOTA

Listing 8 combines item 2 from the lists in Sections 2.6.2 and 2.6.1. The `thread1()` input data flows from the `get_ext_state()` functions through the `reduce_state()` functions into the `cs1` and `cs2` shared variables. The `thread2()` output data flows from these same `cs1` and `cs2` shared variables though `compute_ctl()` and finally is output by `set_ext_ctl()`. The data flow is unidirections from input to output, so no OOTA cycles can form.

2.6.3 Statistical Counters

The canonical instance of a unidirectional data-flow pattern is the statistical counter, in which each thread (or CPU, as the case may be) updates its own counter, and the aggregate value of the counter is read out by summing all threads' counters [20, Section 5.2].

Statistical counters do have concurrent updates and reads, and thus must use atomics. However, the concurrent reads can be modeled as returning approximate results (for example, for monitoring or debugging), and can in fact be modeled as sequentially consistent approximate operations. But more to the point, data flow in real use cases is always unidirectional, proceeding from the updater responding to an event and flowing through the counter to some reader displaying or logging statistics. This unidirectional data flow precludes the cycles required for OOTA or RFUB behavior to manifest.

An example abuse is shown in Listing 9. Lines 1 and 2 define a pair of statistical counters `a` and `b`. The `thread1()` and `thread2()` functions form a classic data-dependent OOTA cycle. Assuming both statistical counters start out with all counters zero, we could in theory see the following OOTA sequence of events:

1. Line 6 sums `a`'s counters, obtaining the sum 42.
2. Line 7 increases the current component of `b`'s counter by 42.

3. Line 12 sums `b`'s counters, obtaining the sum 42 due to the increase from line 7.

4. Line 13 increases the current component of `a`'s counter by 42, thus justifying the sum of 42 obtained by line 6.

Of course, the code in Listing 9 is complete nonsense: Counters should count events, not each others's cumulative values. The code as written is about as useful as the proverbial screen door in a submarine. Problems of this sort should be located in a code review, or better yet during the preceding design review.[6]

Exact values are sometimes obtained from statistical counters in stop-the-world situations, such as checking for consistent results at the end of a stress test or benchmarking run [20, Sections 5.3 and 5.4]. Alternatively, counter updates might be carried out while read-holding a given reader-writer lock and counter reads while write-holding that same lock. In all of these cases, OOTA and RFUB behaviors are additionally avoided due to the fully synchronized nature of the readout.

2.6.4 Software Pipelines

Software pipelines break computation up into stages that might proceed concurrently. If the interface between a consecutive pair of stages is simple enough, relaxed accesses might be used for the corresponding communication of data. Pipelines are not necessarily strictly linear, in fact it can be quite advantageous to have concurrent stages feeding into a single subsequent stage via a reduction step. If the output of the concurrent stages is sufficiently simple, the reduction step might be a simple relaxed atomic fetch-and-op operation to a single scalar object. An example of a sufficiently simple output is event counts emanating from concurrent stream processing feeding into later sequential logic.

Note that the independent input and output data patterns discussed in Sections 2.6.1 and 2.6.2 might be endpoints of a software pipeline.

2.6.5 Owner Field for Re-Entrant Mutex

This pattern is first analyzed for full C++, and then for strict C++. Spoiler warning: There is reason to believe that this pattern works in full C++, but not in strict C++.

Pseudo-code for a re-entrant exclusive mutex is shown in Listing 10. Each mutex must track its owner (`owner` on line 5) in order to avoid self-deadlock when the owner re-acquires a mutex that it already holds. This owner field is updated only while the mutex is held (line 18), and its value is used only to compare for equality to the current thread's ID. Before releasing the mutex, the owner writes a special ID to the owner field that is guaranteed not to match the ID of any thread. Other threads can access the owner concurrently with the owner's update, so the owner field must be atomic in order to avoid data races. Of course, a nesting counter (`count` on line 7) must also be used in order identify the outermost lock-release operation, however this counter is accessed

[6] Yes, this could be considered analogous to a difference-equation control system. But in that case, the system being controlled is part of the loop, and proper synchronization must be used when communicating with that system. In addition, the actual difference-equation computation will normally be single-threaded. More importantly, if the system being controlled might pose a threat to life and limb, the design review had jolly well better be sufficiently well-informed and thorough as to avoid this sort of problem.

```
 1 class my_reentrant_mutex {
 2   std::mutex m;
 3   // Writes of owner are protected by m.
 4   // Only owner writes or clears its id.
 5   std::atomic<std::thread::id> owner; // id() if not held
 6   // Protected by m.
 7   int count;  // Held count-1 times by owner
 8   . . .
 9 }
10
11 void my_reentrant_mutex::lock() {
12   std::thread::id me = std::this_thread::get_id();
13   // No other thread can change whether owner == me.
14   if (owner.load(memory_order_relaxed) == me) {
15     ++mutex.count; // Done; reacquired the lock.
16   } else {
17     ... // Acquire m, leaving count == 0
18     owner.store(me, memory_order_relaxed);
19   }
20 }
```

Listing 10: Re-Entrant Mutex Owner Field

only by the thread currently holding the lock (line 15). Therefore, if the lock works correctly, exclusive access will be provided to the nesting counter, as is required. Those wishing to produce a proof of correctness are encouraged to try induction.

However, the only time that the owner field can be equal to the thread ID is when that thread carried out the last update to the owner field and still holds the mutex:

1. Each thread writes only its ID or the special ID.
2. Because `memory_order_relaxed` loads are single-variable SC, and because each thread sets the owner field to the special ID before releasing the mutex, a given thread cannot see its own ID unless it still holds the mutex.
3. Because atomic accesses forbid load tearing, each load from the owner field will return either the special ID or the thread ID corresponding to some thread that recently held the mutex.
4. Therefore, when a thread is not holding the mutex, it is guaranteed not to load its own ID from the owner field.

No other thread is allowed to write to the owner field while the mutex is held, so it is impossible to form the cycles required for OOTA or RFUB behavior to manifest. Therefore, both the reads from and the writes to the owner field may use `memory_order_relaxed`.

This is a special case of unidirectional data flow, with the data flowing from the mutex holder to threads not holding the mutex. The mutual exclusion provided by the mutex prevents any OOTA or RFUB cycles from forming.

However, things might well be more difficult in strict C++.

These potential difficulties stem from the possibility of undefined behavior (UB) back-propagating through a cycle so as to justify the OOTA behavior [13]. To see the rationale for this back-propagating self-justifying UB (BPSJUB?), consider a pair of threads each concurrently attempting to acquire a mutex, but where (incorrectly and inconsistently) each see that the owner field matches their respective thread IDs. Both threads would then simultaneously execute within their respective critical sections, which could result in UB. UB can back-propagate in time, which could quite possibly result in the threads each seeing their own values in the owner field, which is what instigated the UB in the first place.[7]

Therefore, developers and reviewers should assume that owner fields for re-entrant mutexes require full C++ in order to operate correctly.

2.6.6 One-Way Memory Allocation

One-way memory allocation provides fresh memory that is never deallocated, or that is deallocated using a heavy weight one-sided mechanism, for example, a stop-the-world deallocation phase. Such an allocator might use a relaxed atomic fetch-and-add operation on a shared pointer to allocate memory from a contiguous buffer. This pointer would be initialized to reference the beginning of the buffer, and each fetch-and-add operation would add the desired allocation size (perhaps rounded up to meet alignment constraints), returning the initial value of the pointer and leaving the pointer referencing the portion of the buffer following the just-completed allocation.

The semantics of the C++ fetch-and-add operation guarantees that data flows from one runtime operation to the next, acyclicly. Therefore, OOTA cycles cannot be formed on this type of allocator's pointer manipulation alone.[8] However, OOTA cycles can form based on the pointer values returned from such an allocator and from dereferences of these pointers. Adventurous readers can find an early drafty draft analysis of this situation (but on a more complicated allocator) in Appendix A.

In the meantime, code reviewers should view relaxed stores of pointers to newly allocated objects with great suspicion.

2.6.7 Relaxed Consumption

In cases where a full-speed `memory_order_consume` is needed on a weak-memory system and where the developers are willing to live within strict coding standards [19], `memory_order_relaxed` may be used to head dependency chains. In many (but not all!) use cases, the data flow is also unidirectional, proceeding from the thread installing the new object to the threads consuming it.

[7] Full disclosure: Paul wrote this paragraph, and he is not completely sold on back-propagating self-justifying UB. The critical reader should therefore review both David Goldblatt's working paper [13] and Hans Boehm's recent proposal [5].

[8] This pattern can also be considered to be a single-location data structure, as discussed in Section 2.2.

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   long r1 = x.load(memory_order_relaxed);
 7   if (r1 == 0) {
 8     r1 = pure_function();
 9     x.store(r1, memory_order_relaxed);
10   }
11   // r1 is now trusted
12   if (is_bad(r1))
13     bad_behavior();
14 }
15
16 void thread2()
17 {
18   long r2 = y.load(memory_order_relaxed);
19   if (r2 == 0) {
20     r2 = pure_function();
21     y.store(r2, memory_order_relaxed);
22   }
23   // r2 is now trusted
24   if (is_bad(r2))
25     bad_behavior();
26 }
```

Listing 11: Lazy Scalar Initialization

Note well that this design pattern is outside of the current standard.

All of these patterns can be checked by looking for cycles in a dataflow that has been marked unidirectional.

2.7 Java-Style Lazy Scalar Initialization

Given the "Java" in the title, it is only natural to ask why this applies to C or C++. The answer is simple: It applies because this portion of Java is written in C++.[9]

Java-style lazy scalar initialization can be used to track values that are expensive to compute and not known until runtime on the one hand, but immutable and deterministic on the other. A natural way to handle this situation is to have each access check to see if the desired value has already been computed, and, if not, compute the value and store it for later use, as shown in Listing 11. Of course, it is possible that two threads might concurrently load the initial not-yet-computed value, in which case, both threads will

[9] Documented here: `https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`.

compute the value and store it. This does waste CPU time, but this waste is often greatly outweighed by reduced synchronization cost. This reduced synchronization cost is due to the `memory_order_relaxed` loads and stores used to access the value. After all, the fact that the values are deterministic means that the two threads will be storing the same value, so strongly ordered stores provide no benefit. Furthermore, any other thread is guaranteed to see either the before-computation initial value or the exact same computed value.

Given that each value is deterministic, there cannot be a cyclic chain of interdependent values, so this pattern works in both full C++ and in strict C++.

This pattern must be used with caution in cases where the value is a pointer to allocated memory, especially if that memory is allocated and initialized at runtime. First, racing initializations will result in one-time memory leaks. Second, a store of a pointer to recently initialized memory should be a `memory_order_release` store or stronger, and the corresponding loads should be `memory_order_consume` loads or stronger. Therefore, this pattern might not be helpful when the immutable and deterministic values are linked data structures, but it is often used for scalar values.

This pattern can be checked by looking for the check-compute-store pattern of accesses.

2.8 Chaotic Relaxation

There are a number of iterative numerical algorithms for which unsynchronized access does not slow convergence as much as waits for barrier synchronization. These algorithms can use relaxed loads and stores to update the numerical data [2]. The idea is that the iterative convergence tests correct any small errors due to accessing data from some other iteration.

In theory, these algorithms are subject to OOTA and RFUB behaviors. For example, one thread might speculate a NaN[10], which might result in a store of that NaN which might in turn justify some other thread's speculation of a NaN, which could finally justify the first thread' initial speculation. However, in practice, current implementations avoid such behaviors.

How to check this pattern?

2.9 Garbage Collection

By definition, concurrent garbage collectors read pointers in th user's heap while the application is running. On weakly ordered machines, such accesses *must* use `memory_order_relaxed` accesses, since anything else would require all pointer accesses by the user program to be ordered, which is usually far too expensive. Since such collectors often both read and write heap pointers, it is currently difficult or impossible to strictly preclude OOTA in strict C++. However, neither OOTA nor RFUB behavior has been observed in practice.

Note that although such garbage collection for C++ is rare, such garbage collectors, e.g. for Java, are often implemented in C++.

[10] IEEE floating point "not a number".

	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

Table 1: Attributes of Categories of Relaxed Design Patterns

How can this pattern be checked?

3 Attributes of Relaxed Design Patterns

Table 1 shows attributes of design patterns. Cell with "Y" indicate "yes", with "S" indicate "sometimes", and otherwise indicate "no". Please note that this table assumes that back-propagating self-justifying undefined behavior is prevented. The attributes are as follows:

1. *Multiple Threads:* "Y" indicates that the design pattern uses multiple threads in and of itself. Note that ostensibly single-threaded patterns often interact with other patterns extending across multiple threads. For example, the allocator caches discussed in Section A operate within a single thread, but the resulting memory blocks and associated pointers might be passed to other threads using some other pattern such as release-acquire or release-consume.

2. *Concurrent WW:* "Y" indicates that the design pattern involves concurrent relaxed writes to a given object.

3. *Concurrent RW:* "Y" indicates that the design pattern involves concurrent relaxed reads and writes to a given object, but not necessarly concurrent relaxed writes.

4. *But Checked:* "Y" indicates that the values from the concurrent reads are checked if there might have been a concurrent write. See for example Section 2.5.1.

5. *But Discarded:* "Y" indicates that the values from the concurrent reads are discarded if there might have been a concurrent write. See for example Section 2.5.2.

6. *But Fungible:* "Y" indicates that set of writers are fungible if a reader running concurrently with those writers will exhibit the same behavior regardless of which of those writes' values that read returns. An important special case is when all the writers are storing the same value, as discussed in Section 2.7. Another important special case is where readers take one action if the value is (say) zero and another for any non-zero value, and all concurrent writers will write a non-zero value, as discussed in Section 2.4.

   In all of these "But" columns, "S" indicates that some examples in the category might possess the corresponding attribute.

7. *Unordered Cycle:* "Y" indicates that the design pattern can produce an unordered cycle in and of itself. Of course, a combination of design patterns that individually exclude the possibility of an unordered cycle might nevertheless produce an unordered cycle when used in combination. Design and code reviews should therefore carefully consider ordering at the intersection of multiple design patterns.

8. *Strict C++ Safe:* "Y" indicates that the design pattern is expected to work correctly in strict C++ as well as in strict C++. "S" indicates that only some examples in the category are agreed to be safe for strict C++, in which case the examples that are believe to be unsafe for strict C++ are called out in their respective sections.

The key attribute that renders an idiom potentially unsafe is the loading of an atomic value with a relaxed load, and then relying on that value for correctness, for example, by:

1. Using it to determine the value stored into another atomic, or

2. Relying on it to avoid disastrous misbehavior.

In this last, the potential unsafety normally stems from having generated undefined behavior. But this requires the loads to read a bad value, which in practice normally cannot happen in the absence of that bad value actually being explicitly stored somewhere. The exception occurs in case of backwards-propagating undefined behavior. If the `bad_behavior()` functions on lines 8 and 15 invoke undefined behavior, and if that undefined behavior can back-propagate, then that undefined behavior can justify the value loaded that resulted in the call to the `bad_behavior()` function in the first place.

4 Marking of Relaxed Design Patterns

It is currently believed that these design patterns will need to be explicitly marked in order for code reviewers and automatic verifiers to recognize them and validate their

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   unsigned long r1 = x.load(memory_order_relaxed);
 7   if (is_bad(r1)
 8     bad_behavior();
 9 }
10
11 void thread2()
12 {
13   unsigned long r2 = y.load(memory_order_relaxed);
14   if (is_bad(r2)
15     bad_behavior();
16 }
```

Listing 12: Canonical Unsafe Pattern

usage, although the authors would love to be proven wrong on this point. Here are some candidate marking strategies that have been discussed within the C++ standards committee:

1. Create new `memory_order` `enum` members for each new design pattern. This has the benefit of calling out the pattern in an unmistakable way that is visible to the compiler, but requires that each new design pattern be standardized. It also does not support the case where a given access plays a role in multiple overlapping design patterns.

2. Use structured comments to mark each design pattern. This avoids the time delays and administrative overhead inherent in standardization, and could potentially allow multiple comments to handle a given access that plays a role in multiple overlapping design patterns.

3. Use structured comments with a per-instance identifier for a given use of a pattern. The idea here is to enable tools to more easily spot unintended interactions between different design patterns being applied to a given group of objects. On the other hand, this raises the issue of namespace management.

4. Define C++ `template` types for each design pattern. This is an excellent idea where it applies, as it might well for the statistical counters discussed in Section 2.6. However, we have reason to doubt that `template` types can be reasonably created for all possible relaxed-access design patterns.

More ideation and discussion is needed on this topic.

5 Concluding Remarks

Use of `memory_order_relaxed` can be tricky because we do not yet have an efficient way to formally define the boundaries of OOTA and RFUB. Important `memory_order_relaxed` use cases work in practice, but some have no known precise correctness argument. We define "strict C++" as that portion of the standard excluding the vague normative encouragement to avoid OOTA. The good news is that many common `memory_order_relaxed` use cases are demonstrably correct even in strict C++.

This paper starts the work of classifying known-safe design patterns involving `memory_order_relaxed`. It is hoped that this work will be of use in design and code reviews, and that it might eventually lead to improved theoretical models of `memory_order_relaxed` accesses.

A Allocator Caches

This appendix expands on the one-way memory allocator discussed in Section 2.6.6 by way of a multithreaded allocator with caches. Once agreement is reached on the simpler one-way case, this appendix might be promoted to the main paper.

Allocator caches provide per-CPU or per-thread pools of free memory in order to provide high-performance scalable memory allocation in the common case [22, 8]. Accesses to these pools are normally single-threaded by design for reasons of performance and scalability. However, objects are often allocated for concurrent algorithms, It may be helpful to list phases of a dynamically allocated object's typical lifetime in a concurrent context:

1. Allocation.
2. Initialization, including construction.
3. Use. This might include subphases, but given that any such subphases are defined by the user, safely transitioning between them is the user's responsibility. This is usually the only phase that permits concurrent access.
4. Cleanup, including destruction.
5. Deallocation.

Note the possibility of memory reuse means that this is a cycle rather than a sequence.

The key point is that there must be a happens-before edge for each phase transition. In the case of the cleanup to deallocation to allocation to initialization transitions, this happens-before edge is frequently supplied by sequenced-before, courtesy of the fact that allocator caches cause all of those transitions to occur within a single thread in the common case. However, some sort of happens-before edge is required for each phase transition regardless of which thread is executing any given phase.

In the common case, the transitions requiring other-thread-visible ordering are those to and from the "Use" phase. In particular, the complexities of transitioning from the "Use" phase to the "Cleanup" phase has inspired safe memory reclamation schemes, including reference counting, hazard pointers [24, 16, 25], and RCU [23].

In less-common cases where inter-thread transitions occur between other phases, proper synchronization must be provided. For example, the earlier phase might use a release store and the later phase might use an acquire or consume load.

Proper phase-transition synchronization rules out the infamous RFUB cycle shown in Listing 13.[11] This is because the allocation phase on line 15 is required to happen before any later phase, a requirement that is violated by the relaxed accesses on lines 5, 12, and 17.

Again, code reviewers should view relaxed stores of pointers to newly allocated objects with great suspicion.

[11] Adapted from Boehm and Demsky[7, Figure 5].

```
 1 void *heap;
 2
 3 void thread1()
 4 {
 5   r1 = x.load(memory_order_relaxed);
 6   y.store(r1, memory_order_relaxed);
 7 }
 8
 9 void thread2()
10 {
11   bool allocated(false);
12   r1 = y.load(memory_order_relaxed);
13   if (r1 != heap) {
14     allocated = true;
15     r1 = heap;
16   }
17   x.store(r1, memory_order_relaxed);
18   assert_not(allocated);
19 }
```

Listing 13: RFUB Allocator-Like Example

B History

This is a revision of "P2055R0: A Relaxed Guide to `memory_order_relaxed`" based on discussions at the 2020 Prague meeting and also a presentation[12] to CPPCON 2020.

References

[1] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. Who's afraid of a big bad optimizing compiler? Linux Weekly News, July 2019.

[2] Gregory R. Andrews. *Concurrent Programming, Principles, and Practices*. Benjamin Cummins, 1991.

[3] Mark Batty, Simon Cooksey, Scott Owens, Anouk Paradis, Marco Paviotti, and Daniel Wright. D1780R0: Modular relaxed dependencies: A new approach to the out-of-thin-air problem. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1780r0.html`, June 2019.

[12] `https://docs.google.com/presentation/d/1HT-Gaj5oRT5VArx_4bVyos3y6vCkxubx1rF4Nqo4n64/edit?usp=sharing`

[4] Hans Boehm. "undefined behavior" and the concurrency memory model. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2215r0.pdf, August 2020.

[5] Hans Boehm. "Undefined behavior" and the concurrency memory model. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2215r0.pdf, August 2020.

[6] Hans-J. Boehm. P1217R2: Out-of-thin-air, revisited, again. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1217r2.html, June 2019.

[7] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM.

[8] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Technical Conference*, pages 87–98, 1994.

[9] Neil Brown. Pathname lookup in Linux. https://lwn.net/Articles/649115/, June 2015.

[10] Neil Brown. RCU-walk: faster pathname lookup in Linux. https://lwn.net/Articles/649729/, July 2015.

[11] Jonathan Corbet. The kernel lock validator. Available: https://lwn.net/Articles/185666/ [Viewed: March 26, 2010], May 2006.

[12] Luke Geeson. A proposal fix for c/c++ relaxed atomics in practice. http://lukegeeson.com/blog/2023-10-17-A-Proposal-For-Relaxed-Atomics/, November 2023.

[13] David Goldblatt. P1916R0: There might not be an elegant OOTA fix. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf, October 2019.

[14] David Goldblatt. There might not be an elegant OOTA fix. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf, October 2019.

[15] Richard Grisenthwaite. Views on relaxed atomics in C++ from Arm's technical leadership team. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-technical-view-on-relaxed-atomics, November 2023.

[16] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing*, pages 339–353, October 2002.

[17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. ACM.

[18] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 362–376, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Paul E. McKenney. Proper care and feeding of return values from `rcu_dereference()`. `https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt`, February 2014.

[20] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (2018.12.08a Release)*. kernel.org, Corvallis, OR, USA, 2018.

[21] Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. Out-of-thin-air execution is vacuous. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html`, July 2016.

[22] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, pages 295–306, Berkeley CA, February 1993. USENIX Association. Available: `http://www.rdrop.com/users/paulmck/scalability/paper/mpalloc.pdf` [Viewed January 30, 2005].

[23] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[24] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, August 2002.

[25] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[26] Raúl Silvera, Michael Wong, Paul E. McKenney, and Bob Blainey. N2153: A simple and efficient memory model for weakly-ordered architectures. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2153.pdf`, January 2007.

[27] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 161–174, New York, NY, USA, 2017. ACM.