Can Seqlocks Get Along With Programming Language Memory Models?

Hans-J. Boehm

HP Laboratories
HPL-2012-68

Keyword(s):

Memory models; seqlocks; reader-writer locks; memory fences

Abstract:

Seqlocks are an important synchronization mechanism and represent a significant improvement over conventional reader-writer locks in some contexts. They avoid the need to update a synchronization variable during a reader critical section, and hence improve performance by avoiding cache coherence misses on the lock object itself. Unfortunately, they rely on speculative racing loads inside the critical section. This makes them an interesting problem case for programming-language-level memory models that emphasize data-race-free programming. We analyze a variety of implementation alternatives within the C++11 memory model, and briefly address the corresponding issue in Java. In the process, we observe that there may be a use for "read-dont-modify-write" operations, i.e. read-modify-write operations that atomically write back the original value, without modifying it, solely for the memory model consequences, and that it may be useful for compilers to optimize such operations.

External Posting Date: June 6, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: June 6, 2012 [Fulltext]

To Be published in MSPC 2012: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness

Can Seqlocks Get Along with Programming Language Memory Models?

Hans-J. Boehm
HP Laboratories
Hans.Boehm@hp.com

Abstract

Seqlocks are an important synchronization mechanism and represent a significant improvement over conventional reader-writer locks in some contexts. They avoid the need to update a synchronization variable during a reader critical section, and hence improve performance by avoiding cache coherence misses on the lock object itself. Unfortunately, they rely on speculative racing loads inside the critical section. This makes them an interesting problem case for programming-language-level memory models that emphasize data-race-free programming. We analyze a variety of implementation alternatives within the C++11 memory model, and briefly address the corresponding issue in Java. In the process, we observe that there may be a use for "read-dont-modify-write" operations, i.e. read-modify-write operations that atomically write back the original value, without modifying it, solely for the memory model consequences, and that it may be useful for compilers to optimize such operations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.2 [*Programming Languages*]: Language Classifications—C++; D.4.1 [*Operating Systems*]: Process Management—Mutual exclusion

General Terms algorithms, languages, performance, reliability, standardization

Keywords atomic operations, C++, fences, Java, memory model, reader-writer locks, seqlocks, sequence numbers

1. Introduction

Many data structures are read more frequently than they are written. If such data structures need to be accessed concurrently by multiple threads, there are several alternatives to protecting them:

mutexes or traditional reader-writer locks These do not take advantage of infrequent writes. They further have the crucial disadvantage that even read accesses update the state of the lock. Thus if many threads try to read the data at the same time, they will still generate cache contention, as each thread in turn tries to obtain exclusive access to the cache line holding the lock. This is true even in the case of a conventional reader-writer lock, where the actual data read accesses can normally proceed in parallel on a shared cache line.

RCU An alternative mechanism is to encapsulate all the shared data in a separately allocated object, which is referenced by a globally accessible pointer p. Read accesses dereference p and access the shared data. Write accesses construct a copy of the data and atomically replace p. Reads may read slightly stale data, since they may continue to follow a previously read version of p. A nontrivial protocol [16] or a garbage collector is required to eventually reclaim the separately allocated objects.

seqlocks The data can be "protected" by a sequence number. The sequence number starts at zero, and is incremented before and after writing the object. Each reader checks the sequence number before and after reading. If both values are the same and even, then there cannot have been any concurrent increments, and the reader must have seen consistent data.

Here we focus on the last option. Seqlocks [12, 10, 17] are used, for example, in the Linux kernel to implement `gettimeofday()`. They are also available in an experimental extension of `java.util.concurrent` as `SequenceLock`. [14] Transactional memory implementations often use similar techniques (cf. [9]). We expect similar techniques are widely used elsewhere.

In order to make this as concrete as possible we will start with the code in Figure 1 which is similar to descriptions in the literature. We use C++, since it supports more expressive specifications of atomic operations, and its memory model is currently better defined. However, we argue in section 8 that

```
atomic<unsigned> seq;  // seqlock representation
         // assume big enough to ignore overflow
int data1, data2;  // protected by seq

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
      seq0 = seq;
      r1 = data1;   // INCORRECT! Data Race!
      r2 = data2;   // INCORRECT!
      seq1 = seq;
   } while (seq0 != seq1 || seq0 & 1);
   do something with r1 and r2;
}

void writer(...) {
    unsigned seq0 = seq;
    while (seq0 & 1 ||
      !seq.compare_exchange_weak(seq0, seq0+1)) {}
    data1 = ...;
    data2 = ...;
    seq = seq0 + 2;
}
```

Figure 1. Seqlocks: Initial buggy version

very similar issues apply to Java. We will see that this code is not actually correct by C++11 (or by essentially any other real) memory model rules.

We've inlined the lock implementation to make the memory model issues more apparent, at the expense of code structure.

Writers effectively use the least significant bit of `seq` to ensure mutual exclusion among writers. If the least significant bit is set, the lock is held, and other writers wait. No writer ever modifies seq or data unless it first changed `seq` from even to odd. (The `compare_exchange_weak` call compares `seq` against `seq0`. It replaces `seq` by `seq0 + 1` only if they are still the same. Otherwise it reloads `seq` into `seq0`.) The least-significant bit of `seq` effectively acts as a simple spin-lock that prevents multiple writers from entering the critical section. At the end of the critical section the value of `seq` is incremented once more, to the next even value. Thus readers can tell whether there was an intervening update.

Readers check that the initial and final values of `seq` are identical and even. If they are not, a writer intervened, and the reader retries. If they are, then a writer can't have been active in the interim, and `r1` and `r2` must contain a consistent snapshot of `data1` and `data2`.

Unfortunately, this code, as written, is incorrect. In the next section, we review the C++11/C11 memory model, and the reason this is incorrect. Again, similar issues arise for most other real programming languages.

`int x = 0;` `atomic<int> y = 0;`	
Thread 1	*Thread 2*
a: `x = 42;`	c: `while(!y) {}`
b: `y = 1;`	d: `assert (x == 42);`

Figure 2. Atomic operations introduce happens-before relationship

2. A quick review of the C++11/C11 memory model

Here we give a very quick and informal overview of the C++11 memory model, which is essentially identical to the C11 memory model. We liberally omit details that do not concern us here. Better and more detailed descriptions can be found in [8, 11, 3].

C++11 defines a *happens-before* relation on memory accesses in a program execution. It represents enforced ordering on those accesses. Access a happens before b if either a occurs before b in the same thread (a "is sequenced before" b), or if a and b are synchronization operations such that a *synchronizes with* b. For example a lock release synchronizes with the next lock acquisition. The happens-before relation is transitively closed, i.e. a may also happen before b because there is an intermediate access c that "happens after" a and happens before b.

If two data accesses a and b in a program execution are unordered by happens-before, and one of them is a write access, then we have a *data race*. (Since no ordering between them is enforced by thread order or synchronization, they may happen concurrently.) C++11 gives *undefined* semantics to data races. Essentially they are errors analogous to out-of-bounds array references. In the absence of data races, execution is restricted to obey the happens-before relation. By default this coincides with *sequentially consistent* [13] semantics.

In order to make it easier to write high-performance data-race-free programs, C++11 also supports atomic operations. These are viewed as synchronization operations, and hence cannot themselves introduce data races. Generally they also introduce happens-before relationships sufficient to ensure expected behavior for non-racing data operations. For any data type `T`, the type `atomic<T>` holds a `T` value and supports atomic loads, stores, compare-exchange, etc. For an atomic variable `x` of type `atomic<T>` a load operation on `x` can either be written as just `x`, relying on an implicit conversion from `atomic<T>` to `T`, or explicitly as `x.load()`.

Atomic operations may specify memory ordering constraints. A `memory_order_release` store operation synchronizes with a `memory_order_acquire` operation load operation which reads that value. A `memory_order_relaxed` operation adds no happens-before ordering relations. By default, operations on atomic variables are sequentially con-

sistent, which implies `memory_order_acquire` behavior for load operations, `memory_order_release` behavior for store operations, or both for atomic read-modify-write operations such as `fetch_add` or `compare_exchange`. It also implies additional, usually more expensive, constraints to ensure full sequential consistency. For example, in Figure 2, the store to `y` by Thread 1 implicitly enforces sequential consistency, and thus, among other properties, behaves as a `memory_order_release` operation, which synchronizes with the final (also sequentially consistent) load of `y` by the while loop. Thus a happens before b (program order or "sequenced before"), which happens before c (synchronizes with), which happens before d (sequenced before). Hence a happens before d, and there is no data race on x. The same statement would hold if, for example, b were replaced with `y.store(1, memory_order_release);`

We normally encourage a programming style in which atomic operations initially rely on the default behavior, and explicit ordering constraints (e.g. `x.load(memory_order_acquire)`), which are much more difficult to reason about, are used only to address specific performance problems. In the absence of any explicit ordering constraints (and of data races), the languages promise sequential consistency.

The C++11 memory model also enforces "cache coherence" for all atomic operations, even those with a `memory_order_relaxed` constraint. Essentially updates to *a single memory location* occur in a total *modification order*, and all accesses to *a single memory location* appear to occur in a total order consistent with happens before ordering. We will occasionally need to reason about modification orders associated with individual variables.

The language also provides explicit memory fences (e.g. `atomic_thread_fence(memory_order_acquire)` with fairly complex semantics.[1] We will mention those semantics only as we need them below.

The code in figure 1 is incorrect by these rules, in that it allows data races. There is nothing to prevent `data1` and `data2` from being read while an update is in progress. This race may appear to be benign, but it is clearly disallowed by the language specification. And, as we argue in [6], in somewhat more complex cases, it can lead to subtle miscompilation inside the reader critical section. But it also leads to more blatant problems which we discuss in Section 4. These appear to be unique to explicitly speculative synchronization mechanisms like seqlocks.

3. A straightforward fix

To obtain C++ code that we can meaningfully discuss, we will have to turn `data1` and `data2` into `atomic` variables. We obtain a simple and correct version by replacing the declaration of `data1` and `data2` with

```
atomic<unsigned> seq;  // seqlock representation
         // assume big enough to ignore overflow
atomic<int> data1, data2;  // protected by seq

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
      seq0 = seq;
      r1 = data1;
      r2 = data2;
      seq1 = seq;
   } while (seq0 != seq1 || seq0 & 1);
   do something with r1 and r2;
}

void writer(...) { /* as before */
```

Figure 3. Seqlocks: Simple atomic version

```
atomic<int> data1, data2;
```

The result is repeated for reference in Figure 3. By eliminating all data races, it guarantees sequentially consistent semantics by the C++ memory model, and thus guarantees the expected semantics.

However, this is often viewed as unsatisfactory for two reasons:

1. It is unusual and feels unnatural to require that "lock protected" variables be accessed atomically.
2. The resulting code may not perform well. In real cases, there may be many loads in the reader, not just two. Each load is now required to ensure sequential consistency, and in particular it must not be visibly reordered with respect to any of the other loads. The cost associated with this can range from minor compiler reordering constraints [21, 20, 15] to very expensive fence instructions associated with each load [2].

Point 1 is arguably the result of an unreasonable expectation. It is already well-known, in at least some circles [12], that the reader "critical section" fundamentally *does not* behave entirely like a critical section. Since it can race with a writer it can see inconsistent data, which greatly restricts what can appear in the critical section. If, for example, we were to write in the reader, between the two loads of `seq`:

```
if (data2 != 0) {
    r1 = data1/data2;
} else {
    r1 = 999999999;
}
```

the code may well crash because `data2` may change between the test and the division. In this case, the `seq` test

[1] Unfortunately the complexity of the semantics has increased since they were first adopted into the language.

```
atomic<unsigned> seq;  // seqlock representation
// assume big enough to ignore overflow
atomic<int> data1, data2;  // protected by seq

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
      seq0 = seq;
      r1 = data1.load(m_o_relaxed);  // INCORRECT!
      r2 = data2.load(m_o_relaxed);  // INCORRECT!
      seq1 = seq;
   } while (seq0 != seq1 || seq0 & 1);
   do something with r1 and r2;
}

void writer(...) { ... } // unchanged
```

Figure 4. Incorrect seqlocks with relaxed atomics

```
atomic<unsigned> seq;  // seqlock representation
// assume big enough to ignore overflow
atomic<int> data1, data2;  // protected by seq

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
      seq0 = seq;
      r1 = data1.load(m_o_acquire);
      r2 = data2.load(m_o_acquire);
      seq1 = seq;  // m_o_seq_cst
   } while (seq0 != seq1 || seq0 & 1);
   do something with r1 and r2;
}

void writer(...) { ... } // unchanged
```

Figure 5. Seqlock reader with acquire atomics

would have failed and the "critical section" would have been retried, but only after it was already too late.

Thus it is not safe to call arbitrary functions to read data structures in the reader "critical section". Aside from any memory model issues, even in a purely sequentially consistent environment, the programmer must already treat reads of shared variables in this "critical section" very specially, and be aware of the possibility of values changing asynchronously. The code must not fault or otherwise produce irreversible side effects even if it sees inconsistent values. This is precisely the case in which we would normally use atomic variables to at least ensure that we only see individual values that were actually written, and not partial stores. Thus we focus on point 2 above.

We can try to address the performance issue by explicitly using C++ `memory_order_relaxed` atomic loads, so that we still inform the compiler that races on `data1` and `data2` are expected, but refrain from enforcing additional, potentially expensive, memory ordering. Abbreviating `memory_order_relaxed` to `m_o_relaxed`, this gives us the, again subtly incorrect, version in Figure 4.

(The stores to `data1` and `data2` in the writer could also use `memory_order_relaxed` without introducing additional problems. But we want to focus on the reader.)

4. The real problem

One could be tempted to think that the above solution is correct. All accesses to `seq` use atomic operations that are implicitly "sequentially consistent", and the data races have been removed. However "sequentially consistent" atomics ensure sequential consistency only in the absence of weakly ordered, i.e. non-sequentially consistent, atomics (and of course the absence of data races). Since we are using `memory_order_relaxed`, that does not apply here.

In particular, consider the last two loads in the Figure 1 version of the reader, with `data2` not yet declared atomic:

```
int r2 = data2;
seq1 = seq;
```

The second statement performs a "sequentially consistent" atomic load from `seq`. But that is *not* required to prevent the processor or compiler from reordering the two loads. It is required to ensure that data-race-free program exhibit sequential consistency. But no data-race-free program can tell whether they have been reordered. Any thread observing such a reordering would clearly have to write `data2` between the two loads, which would unavoidable be a data race. This fact is reflected in the ordering guarantees imposed by the C++ (and Java) memory model, and gives rise to the *roach motel* reordering rules of e.g.[18]. Earlier ordinary loads can be delayed past a sequentially consistent atomic load. (The reverse is not true in general, since an earlier sequentially consistent load may affect control flow, in which case reordering may introduce a data race.)

Thus a likely way for Figure 1 to break in practice is that the load from `data2` appears to occur after the second load of `seq` and a writer executes in its entirety between the two. This allows the sequence number tests to succeed even though the loads of `data1` and `data2` were separated by a writer, and are thus likely to be inconsistent.

Similarly, changing the data load into a `memory_order_relaxed` load, as in Figure 4, does not inhibit this reordering, and it is prone to the same failure.

5. Some correct solutions

We can prevent the above reordering by turning the data loads into `memory_order_acquire` operations, giving us

```
atomic<unsigned> seq;  // seqlock representation
// assume big enough to ignore overflow
atomic<int> data1, data2;  // protected by seq

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
      seq0 = seq.load(m_o_acquire);
      r1 = data1.load(m_o_relaxed);
      r2 = data2.load(m_o_relaxed);
      atomic_thread_fence(m_o_acquire);
      seq1 = seq.load(m_o_relaxed);
   } while (seq0 != seq1 || seq0 & 1);
   do something with r1 and r2;
}

void writer(...) { ... }
    // continues to use seq_cst or
    // acquire/release operations on data
```

Figure 6. Seqlock reader with acquire fence

(again abbreviating `memory_order_...` to `m_o_...`) the code in Figure 5.

Very informally this is correct, in that `m_o_acquire` prevents reordering with later operations by the same thread, ensuring that the loads of `data1`, `data2`, and `seq` become visible in order. More formally, we can show that it is correct as follows:

As in all the other cases, if the read into `seq0` by reader r sees the final store of `seq` by a writer, then the `data` updates within that writer happen before the data accesses in the reader. Thus we only have to show that updates by a later writer, i.e. whose updates to `seq` we have not yet seen, are not visible to r. To see this assume that r's load from `data`n sees a write to $data$n by writer w. This implies the following happens-before ordering:

Initial update of `seq` by w is sequenced before
Write to `data`n by w synchronizes with
Read of `data`n by r is sequenced before
final read of `seq` by r

It follows that the initial update of `seq` in w happens before the final read of `seq` by r. Hence if a reader sees an update associated with a writer, at least the second load of `seq` must also see a `seq` update by the writer, precluding the failure from the last section.

Note that these arguments really only rely on consistent use of `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel` for loads, stores, and read-modify-write operations respectively. The operations on `seq` in both the reader and writer could be weakened correspondingly. (The final load of `seq` in the reader could even become `memory_order_relaxed`.)

At least in theory, the major weakness of this approach is that we are constraining all loads for `data`n to become visible in order. This is clearly not necessary; we really only care that the final load from `seq` is performed last. C++11 provides a minimal set of fence primitives designed to address issues such as this. Fences were intentionally designed to be somewhat weak, so as to avoid implementability concerns on some hardware platforms. We tend to discourage their use, since they tend to be very subtle to use correctly, and the industry already has a fairly poor track record in generating correct fence-based code [5]. However, we are already well into memory model subtleties here, and fences do allow the often better-performing solution in Figure 6.

This uses `atomic_thread_fence(m_o_acquire)` which, very informally, has the effect of turning preceding `memory_order_relaxed` operations into acquire operations, but does so without imposing an order on those preceding operations.

The correctness argument is basically as above, but the happens-before chain becomes:

Initial update of `seq` by w is sequenced before
Write to `data`n by w synchronizes with
The acquire fence in r (since preceding operation saw write, 29.8 p4 in [11]) is sequenced before
final read of `seq` by r

On TSO machines such as x86, loads are implicitly ordered by the hardware, and sequential consistency for atomic operations can be enforced without adding overhead to loads [21, 20]. All the correct seqlock implementations we have seen so far should be compiled to code that avoids all memory fence overhead in the reader. This includes Figures 3, 5, and 6. However the solutions we have presented so far do have weaknesses:

- Both Figure 5 and especially Figure 3 incur large overheads on architectures such as POWER [20, 2], which require expensive fences for such loads. Thus they're undesirable in code that has to perform well across architectures.
- The fence-based solution in Figure 6 obtains high performance only through the use of a very difficult-to-use and C++11/C11-specific construct. It cannot be used in Java, for example.
- On a more theoretical level, these solutions all over-constrain ordering. They all prevent later operations in the reader thread from moving *into* the reader critical section. (normally we allow movement of memory operations *into*, but not out of, critical sections [5, 18, 7])

Thus we explore yet another alternative.

6. Using read-dont-modify-write operations

We can avoid over-constraining the ordering by using `memory_order_relaxed` for data accesses, but enforcing

```
atomic<unsigned> seq;  // seqlock representation
// assume big enough to ignore overflow
atomic<int> data1, data2;  // protected by seq

t reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
      seq0 = seq.load(m_o_acquire);
      r1 = data1.load(m_o_relaxed);
      r2 = data2.load(m_o_relaxed);
      seq1 = seq.fetch_add(0, m_o_release);
   } while (seq0 != seq1 || seq0 & 1);
   do something with r1 and r2;
}

void writer(...) { ... } // unchanged
```

Figure 7. Seqlock with read-dont-modify-write

the necessary ordering by adding an update to the second sequence number read. We can easily ensure that the reads become visible before an update by specifying `memory_order_release` before the update. Since the update exists only for memory ordering purposes, we arrange that the update writes back the just-read value, and that the read and write are performed as a single atomic "read-modify-write" operation, though there is no actual modification involved. The resulting read-only critical section is given in Figure 7.

Here we're using `fetch_add(0, m_o_release)` as the required "read-dont-modify-write" operation; most other `fetch_op` operations can also similarly perform an identity operation with the proper arguments, as could a `compare_exchange` operation comparing against the original sequence number.[2]

Theorem The above implementation guarantees that the following statement holds for all seqlock critical sections referring to the same seqlock, i.e. to the same sequence number variable: Writer critical sections and *successful* reader critical sections can be totally ordered such that whenever *cs1* and *cs2* are two critical sections such that *cs1* precedes *cs2* in this order, and at least one of them is a writer critical section, then *cs1* happens before (in the C++11 memory model sense) *cs2*.

It follows that seqlock critical sections behave like actual critical sections; each critical section sees the changes from the latest writer critical section preceding it and no later ones.

Proof The C++11 memory model guarantees that all updates to the single location `seq` occur in a single total order, the modification order of `seq`. Thus we can order critical sections based on the appearance of the final read-modify-write operation to `seq` (readers) or write operation to `seq` (writers) in the modification order. All critical sections referring to the same sequence number variable thus occur in a well-defined sequence, and we can refer to the n^{th} critical section to be executed.

The first successful reader critical section after a writer critical section w reads (with a `memory_order_acquire` load) the sequence number written at the end of the immediately preceding writer critical section, and hence happens after w. If there are multiple successful readers before the next writer in the sequence, each subsequent successful reader critical section r reads either the sequence number written at the end of w, or the identical value written by an earlier reader that also follows w. In the first case, the final write of w directly synchronizes with the initial read of r and hence happens before r. In the latter case w happens before the earlier reader, which synchronizes with r, and thus transitively happens before the reader. Similarly, the next writer in the sequence reads the sequence number written by w or by one of the intervening readers, and hence must again happen after w.

It remains to be shown that successful reader critical sections happen before the next writer w', and thus can't see effects of w'.

No update to `seq` of any kind (including a reader `fetch_add(0, ...)`) from a successful critical section can appear in `seq`'s modification order between the successful `compare_exchange_weak` and store operations corresponding to w'. An intervening writer attempt to enter its critical section with `compare_exchange_weak` will always fail. Similarly another writer can't possibly exit its critical section in the middle, since w' would have failed to enter its critical section. The `fetch_add` of a successful reader can't occur in the middle, since it would yield an odd value of `seq`, which would cause the reader to fail. It follows that if we order critical sections based on appearance in the modification order of `fetch_add` for readers and successful `compare_exchange_weak` for writers (instead of the final store), we get the same order as before.

Each of the reader `fetch_add` operations, as well as the successful writer `compare_exchange_weak` operations, are atomic operations, meaning they must read the preceding write in modification order. It follows that each of these `fetch_add` and `compare_exchange_weak` operations (though not necessarily the entire critical sections) happen before the next one, and hence each reader critical section happens before the next writer critical section w'. •

We do not ensure that readers are totally ordered by happens-before, but since they are presumed to only read shared variables, that should not be observable.

Unlike the fence-based approach, the `fetch_add` with release ordering does not unnecessarily prohibit reordering of other operations with the end of a reader critical section.

[2] The last option was pointed out by Vitaly Davidovich and Ruslan Cheremin on Doug Lea's `concurrency-interest` mailing list.

Thread 1
a: `x.store(1,m_o_relaxed);` b: `r1 = y.fetch_add(0,m_o_release);`
Thread 2
c: `y.fetch_add(1,m_o_acquire);` d: `r2 = x.load(m_o_relaxed);`

Figure 8. `Fetch_add(0)` needs a memory fence

Subsequent memory operations, including stores, can still move *into* the critical section.

7. Optimizing read-dont-modify-write operations

The above "solution" has the huge disadvantage that the final load from `seq` has been turned into an operation that normally requires exclusive access to the cache line. This reintroduces coherence misses as ownership of the cache line is transferred between threads executing read-only critical sections.

Ideally, it would be possible to address this by having the compiler treat read-dont-modify-write operations, such as `fetch_add(0, memory_order_release)` specially, to precisely an atomic load operation plus whatever memory fences etc. are required to enforce the required ordering. Unfortunately, that is not correct.[3] To illustrate, consider the code in Figure 8 (where once again `memory_order` is abbreviated `m_o`).

Assume `x` and `y` are both initially zero. We claim that `r1 = r2 = 0` is impossible. If `r1` were zero, then (b) must have preceded (c) in `y`'s modification order. Thus (c) must see the result of (b). Hence (b) synchronizes with (c) and (a) happens before (d), ensuring r2 = 1.

Now consider how this code is compiled to x86, using the mapping in [20]. The store (a) is compiled using an ordinary store (`MOV`) instruction. Atomic loads are always compiled to a simple store (also `MOV`), even for `memory_order_seq_cst`. If the `fetch_add(0, ...)` instruction (b) were also compiled to an ordinary load instruction, as would be desirable for the `seqlock` implementation, then Thread 1's operations could become visible out of order, since the store to `x` could remain in the store buffer during the load of `y`. Hence it is not in general safe to compile (b) to a simple `MOV` instruction.

In the seqlock case, there is no relevant store preceding the `fetch_add(0, ...)` operation. Hence a `MOV` would be an acceptable translation. However, the compiler normally has no way to determine the absence of such an atomic store operation, especially since it could occur much earlier, in a different compilation unit. Hence it appears impossible to avoid generating a fence as part of the `fetch_add(0, ...)` operation.

[3] ... in spite of my earlier claims in the previously mentioned mailing list discussion.

We can however safely compile `x.fetch_add(0, memory_order_release)` to an `MFENCE` followed by a `MOV` instruction. This is easy to see based on the x86 memory model as presented in [21]. In this model, both the normal compilation of `fetch_add(0, ...)` as an atomic add operation with an implicit fence e.g. `lock add`, and our proposed compilation as `MFENCE; MOV` have exactly the same effect: The store buffer is flushed, memory is unchanged, and the value at the effected memory location is returned.

We conjecture that similar optimizations to remove the need for exclusive cache line access are possible on other architectures. But extra fence semantics are also likely to be required.

8. Java

Java provides neither weakly ordered atomic loads nor fences. Thus the only available options are the algorithm from Figure 3 using `volatile` operations in place of sequentially consistent atomics, and the version from Figure 7, this time using ordinary non-volatile operations on `data`n, and a sequentially consistent `fetch_add`. The former is much simpler, and not subject to current uncertainties about the treatment of data races in the Java memory model [1, 19]. But in the presence of appropriate, but currently hypothetical, compiler optimizations, the latter is likely to perform significantly better on architectures with expensive `volatile` load implementations, such as POWER.

9. Conclusions and performance comparison

Seqlocks may well be the most challenging of common programming idioms to implement with modern programming language memory models. We presented a number of reasonable ways to implement them.

1. We can constrain data loads to not be reordered. This includes Figure 3 as well as Figure 5. The first one of these is the only viable solution for existing Java implementations. On TSO-based architectures, such as x86, even sequentially consistent loads are inexpensive, and these solutions result in essentially optimal code. Unfortunately, on architectures such as POWER and pre-v8 ARM, such loads require fences, making them relatively expensive.
2. The fence-based solution in Figure 6, though it abstractly over-constrains the problem, appears to be a good match to all current architectures, and should generate nearly optimal code. But it is not expressible in Java, and appears exceptionally unnatural to us.
3. Solutions based on a trailing read-dont-modify-write operation are not viable without compiler support, since they introduce writes into reader critical sections. But the right compiler optimization can make them viable across all languages and architectures, though not optimal.

The simple atomic solution from Figure 3 follows from normal C++11 programming rules: If a variable is intention-

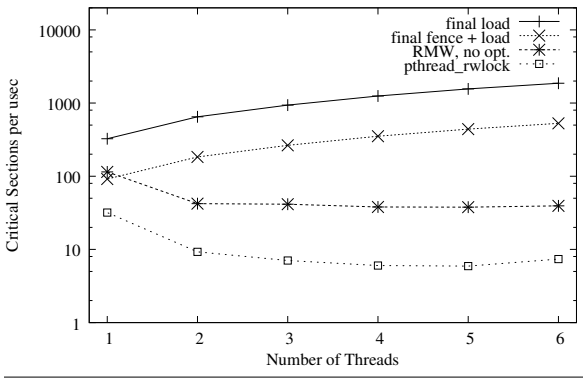

Figure 9. Seqlocks: Millions of reads per second

ally accessed in a racy fashion, make it `atomic`. It is easy to obtain, once the race is identified. And unless the race is identified, more realistic code is unlikely to be correct even with a hypothetical fully sequentially consistent, implementation: The body of the reader "critical section" is unlikely to be robust against concurrent updates.

Unfortunately, implementations that lead to more portable good performance still require a deep understanding of memory model intricacies.

The performance of various options on a TSO machine is illustrated in Figure 9. Measurements were obtained on a machine containing a single Xeon X5670 processor with six cores and "hyperthreading" disabled. (This is a recent "Westmere" processor with relatively fast synchronization and fence support.) We ran a microbenchmark with 1 to 6 threads at a time, each alternating a million read accesses with one write access. We measured the total number of reader critical sections performed in 10 seconds, and report the total number of reader critical sections, in millions, executed per second. Each critical section reads two integer variables as in our examples.[4] Note the log scale on the y-axis and the huge performance differences.

Neither pthread reader-writer locks ("pthread_rwlock" in Figure 9), nor seqlocks with an unoptimized read-dont-modify-write operation ("RMW, no opt") scale with processor count. This is expected for the tight reader loop we tested, since exclusive access to the cache line containing the lock data structure is required to execute a reader critical section. Critical sections with either a plain load instruction ("final load", as would be generated from sequentially consistent (Figure 3) or acquire data loads (Figure 5), or from the explicit fence version (Figure 6)) or a fence followed by a load ("final fence + load", the optimized code generated from the read-dont-modify-write code in Figure 7) scale essentially perfectly with processor count. (Linear speedup doesn't result in a straight line, due to the log scale.) Note that optimizing `fetch_add(0, ...)` to a fence followed by a load gains more than a factor of 13 on a 6 core machine (a factor of 74 on a separate 32 core and 32 thread experiment). But even on a modern processor such as this, the fence still imposes a significant slowdown (more than a factor of 3.5), though one that would be far less noticeable in a real application.

Unfortunately we were not able to obtain a fully trustworthy set of POWER measurements for comparison. We were able to run our microbenchmark on a POWER 7 machine shared with several other users, to see general trends. The scalability trends for the different solutions were similar to the x86 machine. However all correct solutions on POWER result in at least two weak fences in the reader critical section. Those already slowed down the code by roughly a factor of three relative to the fence-less version. (The fence-less version ran correctly on only a single thread.) The fully-fenced version corresponding to Figure 3 was roughly another factor of three slower, though it again scaled well. The unoptimized read-modify-write implementation beat the fully-fenced version on a single thread but, as expected, its throughput decreased as threads were added, instead of scaling with the number of threads.

[4] The experiments were performed on a desktop Linux machine, where we found that we needed to bind threads to cores to get reasonable scheduling behavior and reproducible results. This may have been slightly unfair to pthread reader-writer locks. We later reproduced very similar results, with continued near-linear scaling for write-free readers, to 32 threads, on a 32-core, 8-socket Linux server with or without threads bound to cores. Gcc 4.6 did not generate sufficiently performing code for the required atomic operations, so we used `libatomic_ops` [4] and manually ensured the proper assembly code was generated.

Acknowledgments

This paper benefited from several discussions of seqlock memory model issues with various people, including Sarita Adve, Pramod Joisha, Doug Lea, and Jeffrey Yasskin, as well as from the reviewer comments and from a public online discussion on Doug Lea's concurrency-interest mailing list. We were able to obtain access to a shared Power 7 with the help of Paul McKenney, OSU OSL, and the gcc Compile Farm.

References

[1] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, August 2010.

[2] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *POPL'12*, 2012.

[3] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL'11*, pages 55–66, 2011.

[4] H.-J. Boehm. The atomic_ops atomic operations package. `http://www.hpl.hp.com/research/linux/atomic_ops/`, 2005.

[5] H.-J. Boehm. Reordering constraints for pthread-style locks. In *Proc. 12th Symp. Principles and Practice of Parallel Programming*, pages 173–182, 2007.

[6] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

[7] H.-J. Boehm. Performance implications of fence-based memory models. In *MSPC*, 2011.

[8] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. Conf. on Programming Language Design and Implementation*, pages 68–78, 2008.

[9] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI*, 2009.

[10] S. Hemminger. Fast reader/writer lock for gettimeofday 2.5.30. Linux kernel mailing list August 12, 2002, `http://lwn.net/Articles/7388/`.

[11] ISO JTC1/SC22/WG21. ISO/IEC 14882:2011, information technology — programming languages — C++. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372` or a close approximation at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf`.

[12] C. Lameter. Effective synchronization on Linux/NUMA systems. Proceedings of the May 2005 Gelato Federation Meeting (`http://www.lameter.com/gelato2005.pdf`), 2005.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[14] D. Lea. jsr166e: Class SequenceLock. `http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/SequenceLock.html`, retrieved Mar. 8, 2012.

[15] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasami. A case for an sc-preserving compiler. In *PLDI*, 2011.

[16] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Engineering at Oregon Health and Science University, 2004.

[17] T. Nkaike and M. Michael. Lock elision for read-only critical sections in java. In *PLDI*, 2010.

[18] J. Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[19] J. Sevcik and D. Aspinall. On validity of program transformations in the java memory model. In *ECOOP 2008*, pages 27–51, 2008.

[20] J. Sevcik and P. Sewell. C/C++11 mappings to processors. `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html"`, retrieved Mar. 3, 2012, 2011.

[21] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocesors. *Communications of the ACM*, 53(7):89–97, July 2010.