Can Seqlocks Get Along with Programming Language Memory Models?

Hans-J. Boehm

HP Labs

The setting

- Want fast reader-writer locks
  - Locking in shared (read) mode allows concurrent access by other readers.
  - Locking in exclusive (write) mode disallows concurrent readers or writers.
- Many more readers than writers
  - We'll ignore write performance.
- Implementation language: C++11/C11, Java

Traditional reader-writer locks

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

Update lock state!

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

excl. shared shared

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

shared shared

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

Core 1: excl. | shared | shared

Core 2: shared | shared

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

shared shared

excl. shared shared

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

excl. shared shared

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

shared shared

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

excl. shared shared

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

shared shared

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

→

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

shared shared

Core 2:

→

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

excl. shared shared

Cache lines needed

Multiple readers:

Core 1:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

Core 2:

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

→

```
rwl.lock_shared();
r1 = data1;
r2 = data2;
rwl.unlock_shared();
```

→

excl. | shared | shared

shared | shared

Seqlocks

- One common solution to this problem.
- Used in Linux kernel, jsr166e `SequenceLock`.
- Similar techniques used for e.g. software transactional memory implementations.
- Readers don't update a lock data structure.
  - Check whether writer interfered.
  - If so, start over …

Seqlocks, version 0 (naïve, broken)

```
atomic<unsigned long> seq(0);
int data1, data2;
```

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

C++11 version, slightly abbrvd.
For Java, use `j.u.c.atomic`.

```
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;
}
```

Problem: Data races

```
atomic<unsigned long> seq(0);
int data1, data2;
```

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

data races!

```
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;
}
```

Probl : Data races

```
atom   un     ed long> seq(0);
int     a1,    a2;
```

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

```
  reader() {
  int r1, r2;
  unsigned seq0, seq1;
  do {
    seq0 = seq;
          data1;
          data2;
          = seq;
        e (seq0 != seq1
          || seq0 & 1);
      thing with r1 and r2;
```

Java version more subtly broken …

stay tuned …

Seqlocks, version 1 (correct)

```
atomic<unsigned long> seq;
atomic<int> data1, data2;
```

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

No data races → sequential consistency

For Java: `volatile int data1, data2;`

```
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;
}
```

Are we done?

- Bad news:
  - `atomic` annotations for *data* superficially surprising.
    - But really shouldn't be.
    - Prevents compiler misoptimization in C and C++.
    - Provides useful properties, e.g. indivisible loads of `long`.
  - Overconstrains read ordering.
    - forces *data* loads to become visible in order.
    - … and sometimes more.
  - Slows down readers on Power 7 by around a factor of 3.
- Good news:
  - Reasonably straightforward.
  - Works.
  - Essentially optimal on X86 and other TSO machines.

Better portable performance? Seqlocks version 2 (broken, again)

```
atomic<unsigned long> seq(0);
atomic<int> data1, data2;
```

(writer unchanged)

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

Seqlocks version 2 (broken, again)

```
atomic<unsigned long> seq;
atomic<int> data1, data2;
```

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

- The problem (informally):
  - m_o_seq_cst guarantees s.c. for programs using *only* m_o_seq_cst.
  - load of r2 may become visible *after* load of seq1!
  - data loads can move out of “critical section”.
  - d.r.f ➔ invisible for data loads
- Explicit ordering is tricky.

Java: Same problem with volatile `seq`, non-volatile `data`*n*.

Using C++11 fences
Seqlocks version 3 (correct)

```
atomic<unsigned long> seq;
atomic<int> data1, data2;
```

(writer unchanged)

```
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;
}
```

Advantage:
- Portable performance

Disadvantages:
- Correctness is subtle
- Fences overconstrain ordering
- Impossible in Java

Back to read-modify-write operations
Seqlocks version 4 (correct)

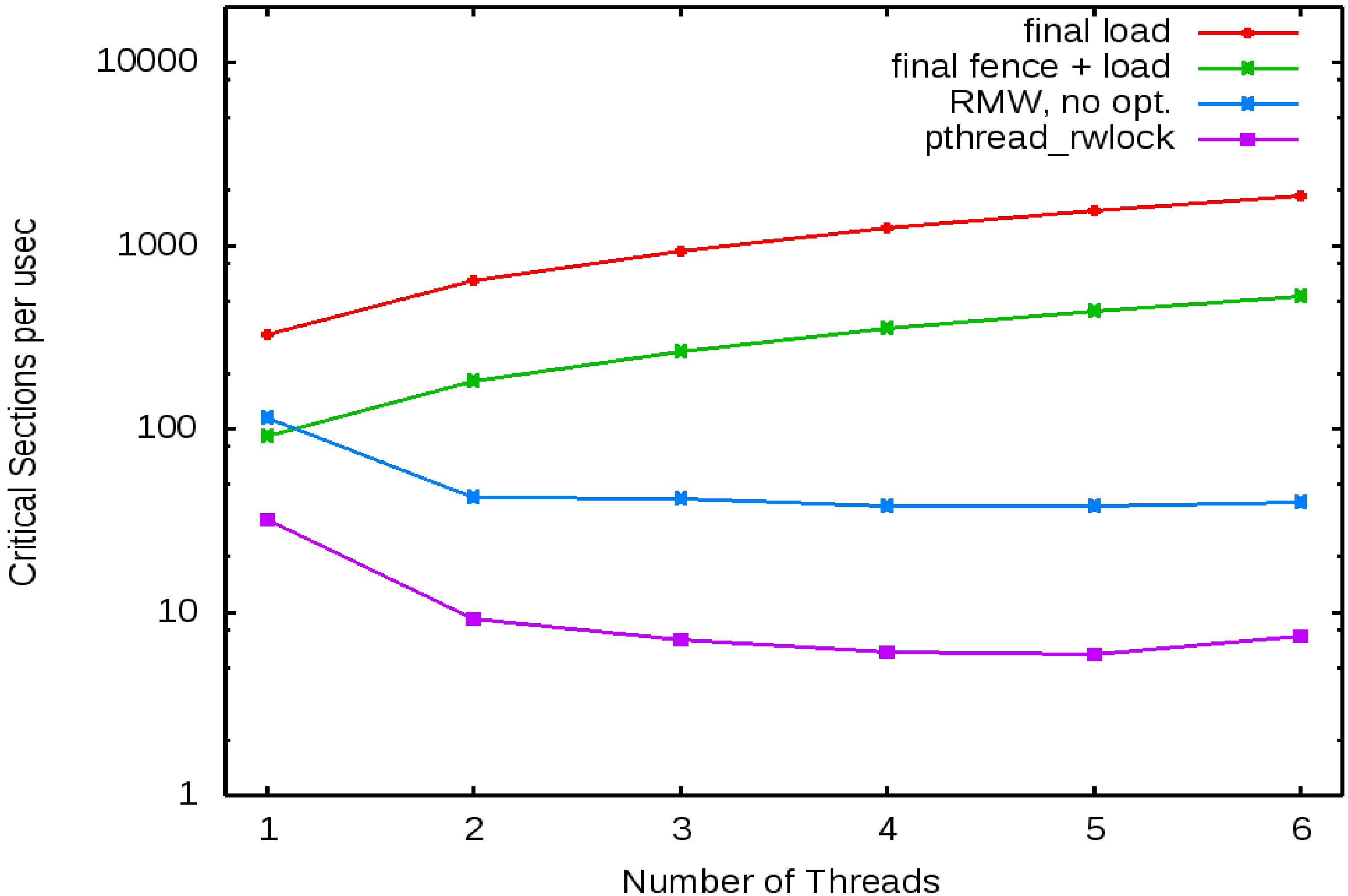
```
atomic<unsigned long> seq;
atomic<int> data1, data2;
```

(writer unchanged)

```
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_and_add(0, m_o_release);
  } while (seq0 != seq1
         || seq0 & 1);
  do something with r1 and r2;
}
```

Read-don't-modify-write operations

- Advantages
  - Seems much more natural: `m_o_acquire` to acquire "lock", `m_o_release` to release lock.
  - Works with Java and ordinary variables in "critical section".
- Disadvantage:
  - Reintroduces store to lock and cache-line ping-ponging.
- But:
  - *Store can be optimized out,* at least on x86, probably on POWER.
  - Unfortunately, an extra fence remains (see paper).
  - Probably the best we can do for Java on POWER.



X86 reader performance

final load ~ seq_cst or fence version

final fence + load ~ optimized RMW (better than seq.cst. on Power)

Bottom line:

- **Version 1** (seq. cst. atomics for data) is easy to write, works with C++ and Java, performs well on some platforms, not others.
- **Version 3** (fences) is very tricky to write correctly. Should perform well everywhere. Only for C & C++.
- **Version 4** (read-don't-modify-write) works everywhere. Scalability depends on currently unimplemented compiler optimization. With optimization: Worse than version 1 on X86, better on POWER.
- **Version 2** (plain relaxed data) may be quite popular in Java, but is undeserving of its popularity.

Questions?

Backup slides

Seqlocks, version 0 (naïve, broken)

```
atomic<unsigned long> seq;
int data1, data2;
```

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

```
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;
}
```

C++ version, slightly abbrvd.
For Java, use `j.u.c.atomic`.