

# Performance Implications of Fence-Based Memory Models

*Hans-J. Boehm*

HP Labs



# Simplified mainstream (Java, C++) memory models

- We distinguish synchronization actions
  - lock acquire/release, atomic operations, barriers, ...
- Synchronization operation *s1* *synchronizes with* *s2* in another thread if *s1* writes a value observed/acted on by *s2*. e.g.
  - `l.unlock()` synchronizes with next `l.lock()`
  - atomic store synchronizes with corr. atomic load
- The *happens-before* relation is the transitive closure of the union of
  - synchronizes-with* U *intra-thread-program-order*

# Happens-before example

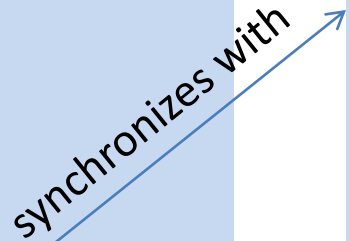
*Thread 1:*

```
l.lock();  
x = 1;  
l.unlock();
```

*Thread 2:*

```
l.lock();  
x = 2;  
l.unlock();
```

synchronizes with



`x = 1` program-ordered before `l.unlock()`  
synchronizes with `l.lock()`  
program-ordered before `x = 2`  
Therefore `x = 1` happens before `x = 2`

# Conditions on a valid execution

- Synchronization operations occur in a total order, subject to some constraints.
  - See paper for details and references.
- Happens-before must be acyclic (irreflexive).
- Every data load must see a store that happens before it.
- If two accesses to the same data are not ordered by happens-before, and one of them is a write, we have a *data race*.
- *Data-race-free executions are sequentially consistent.*
  - For the core language.
- A data race results in
  - undefined behavior (C++, C, Ada) or
  - poorly defined (Java) behavior.

# Absence of races allows reordering

```
l.lock();  
x = 1;  
l.unlock();  
r1 = y;
```

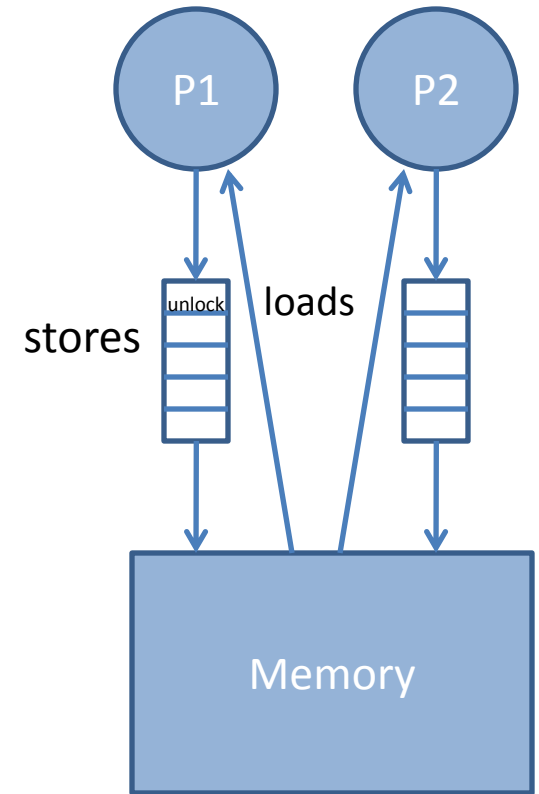


```
l.lock();  
r1 = y;  
x = 1;  
l.unlock();
```

- Independent data operations can be reordered.
  - If another thread could observe intermediate state
    - It would have to access `y` between two statements.
    - It could have exhibited a data race in original code.
- Movement into critical section (roach motel reordering) is unobservable.
- See, for example, Jaroslav Ševčík's work for details.

# Roach motel reordering supports efficient lock implementation

- Some compiler impact (Laura Effinger-Dean's talk helps you characterize this)
- Allows less expensive fences in synchronization constructs:
  - TSO hardware memory model (X86, SPARC):
    - Stores are queued before becoming visible; no other visible reordering.
    - No need to flush queue on unlock(); later reads can become visible before unlock()
    - Nearly factor of 2 for uncontended spinlocks.
  - Avoids full (expensive!) fences on PowerPC, Itanium, and the like.



# OpenMP 3.0 fence-based memory model, roughly

- Memory ordering is imposed by `flush` directives (fences).
- `flush` directives are executed in a single total order. Each `flush` synchronizes with the next one.
- `lock/unlock` implicitly include `flush`.
- These are the only synchronizes-with relationships.
- Otherwise, as before.

# OpenMP 3.0 properties, so far

- Mainstream model guarantees sequential consistency for data-race-free programs.
- OpenMP model adds synchronizes-with and happens-before constraints.
  - which are clearly already satisfied by a sequentially consistent execution

➔ so far, no real change.



# The complication: weakly ordered atomic operations

- Many languages (Java, C++0x, C1x, OpenMP\*) allow atomic operations with weaker ordering.
  - Java `LazySet()`
  - C++0x/C1x `memory_order_relaxed`, etc.
  - OpenMP\* `#pragma omp atomic`
  - UPC `relaxed`
- Don't contribute to data races.
- Simplest case: Contribute no happens-before relationships or other visibility constraints.
  - Other variants also suffice.
- Load can see store that happens before it, or a racing store.
- Data-race-free programs no longer sequentially consistent.

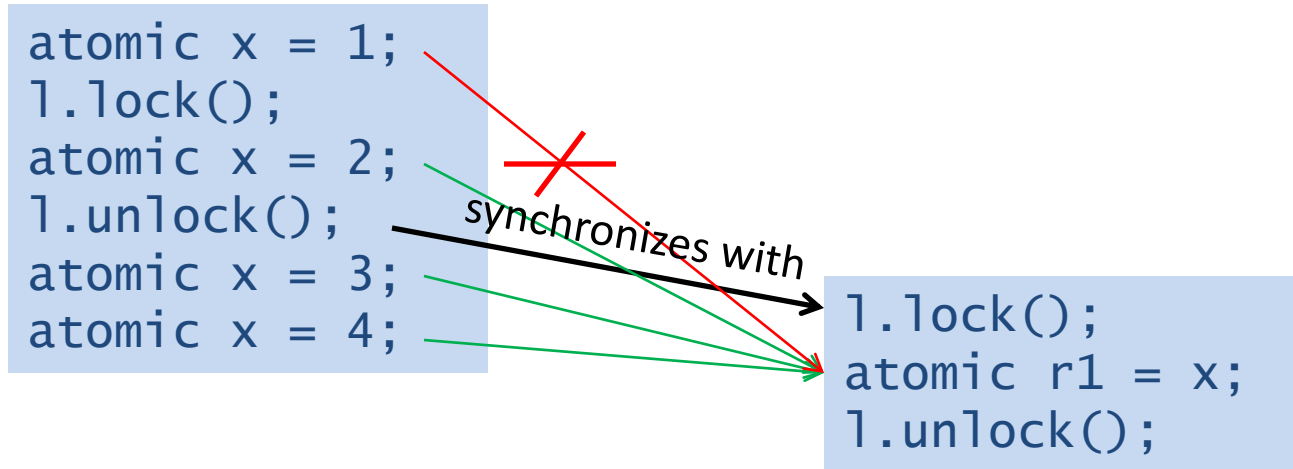
\* We assume OpenMP 3.1 atomics. The OpenMP 3.0 story is complicated ...

# Weakly ordered atomic operations

```
atomic x = 1;  
l.lock();  
atomic x = 2;  
l.unlock();  
atomic x = 3;  
atomic x = 4;
```

~~synchronizes with~~

```
l.lock();  
atomic r1 = x;  
l.unlock();
```



# Weakly ordered atomics example

“Dekker’s example”:

Everything initially zero:

*Thread 1*

```
atomic x = 1;  
atomic r1 = y;
```

*Thread 2*

```
atomic y = 1;  
atomic r2 = x;
```

- Allow  $r1 = r2 = 0!$
- Not Java `volatile` or C++0x default `atomic!`

# Dekker's example with locks, original semantics

“Dekker's example”:

Everything initially zero:

*Thread 1*

```
l1.lock();  
atomic x = 1;  
l1.unlock();  
atomic r1 = y;
```

*Thread 2*

```
l2.lock();  
atomic y = 1;  
l2.unlock();  
atomic r2 = x;
```

- No synchronizes-with relationships!
- Locks don't matter:  $r1 = r2 = 0$  still allowed.

# Dekker's example with locks, fence-based semantics

“Dekker's example”:

Everything initially zero:

<i>Thread 1</i>		<i>Thread 2</i>
<code>l1.lock();</code>		<code>l2.lock();</code>
<code>atomic x = 1;</code>		<code>atomic y = 1;</code>
<code>l1.unlock();</code>	synchronizes with →	<code>l2.unlock();</code>
<code>atomic r1 = y;</code>		<code>atomic r2 = x;</code>

- Initialization still happens before both stores.
- Assume implied flush in thread 1 `l1.unlock()` is first in flush order. (Other case is symmetric.)
- Corresponding `x = 1` store happens before load in other thread.
- Hides initialization from `r2 = x` load. Must see 1.
- `r1 = r2 = 0` disallowed.

# Roach-motel semantics:

```
l.lock();  
atomic x = 1;  
l.unlock();  
atomic r1 = y;
```



```
l.lock();  
atomic r1 = y;  
atomic x = 1;  
l.unlock();
```

- Transformation still allowed w. original semantics.
- Racing accesses may see state inconsistent with sequentially consistent interleaving semantics.
- **Disallowed by implicit flush in unlock.**

# Consequences

- Weakly-ordered atomics distinguish traditional happens-before and fence-based semantics.
- Fence-based semantics → potentially *much* more expensive **lock/unlock**.
  - Rarely optimizable.
- Incorrect OpenMP 3.0 implementations can support much faster uncontended locks.
  - And probably nobody will notice.
- Sequentially consistent atomics don't expose issue:
  - Slows down atomics.
  - Potentially less than lock/unlock slowdown.
  - May be a faster way to implement OpenMP 3.0 spec!

# How does this impact real implementations?

- We suspect proprietary implementations ignore the rules where it matters.
  - Which is probably what users want!
- Inspection of gcc4.4 showed:
  - OpenMP critical section entry on PowerPC did not include full fence.
  - The corresponding Itanium code didn't guarantee proper lock semantics (since fixed).
  - Critical section exit code had full fences.
  - This all appeared to be fairly accidental.
    - We really need to make this less confusing!



# Implications for OpenMP specification

- This was discussed in OpenMP ARB meetings, resulting in:
  - Various memory model clarifications in the OpenMP 3.1 draft.
  - Informal wording in the 3.1 draft allowing roach-motel reordering.
  - Ongoing discussion about a revised memory model, and sequentially consistent atomic operations in 4.0.

# Implications for UPC

- Much more precise memory model in the spec, but:
  - strict accesses have flush-like semantics.
  - “A null strict access is implied before a call to `upc_unlock()`”
  - relaxed shared accesses are essentially weakly ordered atomic accesses.

→ Same problem!

Questions?

# Backup slides

# OpenMP 3.0 atomics example

- Only RMW operations are allowed

- Initially  $x = y = 1$ ;

```
x *= 0;
```

```
y++;
```

```
l.lock();
```

```
l.lock();
```

```
y *= 0;
```

```
x++;
```

- after join, can  $x = 1$  and  $x = 2$ ?
- I believe isync-based PowerPC lock() allows this.
- Dekker's with these primitives is an Itanium example.

# A performance measurement

```
#include <stdlib.h>
```

```
int main()  
{
```

```
    int i;  
    for (i = 0; i < 100*1000*1000; ++i) {  
        free(malloc(8));  
    }  
    return 0;
```

```
}
```

```
> gcc -O2 -lpthread malloc.c
```

```
> time ./a.out
```

```
3.965u 0.001s 0:03.96 100.0%    0+0k 0+0io 0pf+0w
```

Intel Xeon E7330@2.4GHz  
(Core2 / Tigerton)

gcc 4.1.2

RHEL 5.1

# Another one

```
#include <stdio.h>
#include <pthread.h>

void * child_func(void * arg)
{
}

int main()
{
    pthread_t t;
    int code;

    if ((code = pthread_create(&t, 0, child_func, 0)) != 0) {
        printf("pthread creation failed %u\n", code);
    }
    if ((code = pthread_join(t, 0)) != 0) {
        printf("pthread join failed %u\n", code);
    }

    return 0;
}
```

```
> gcc -O2 -lpthread create_join.c
> time ./a.out
0.000u 0.000s 0:00.00 0.0%          0+0k 0+0io 0pf+0w
```

# Both combined

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void * child_func(void * arg)
{
}

int main()
{
    int i;
    pthread_t t;
    int code;

    if ((code = pthread_create(&t, 0, child_func, 0)) != 0) {
        printf("pthread creation failed %u\n", code);
    }
    if ((code = pthread_join(t, 0)) != 0) {
        printf("pthread join failed %u\n", code);
    }
    for (i = 0; i < 100*1000*1000; ++i) {
        free(malloc(8));
    }
    return 0;
}

> gcc -O2 -lpthread both.c
> time ./a.out
9.880u 0.000s 0:09.88 100.0%    0+0k 0+0io 0pf+0w
```



# Where is the time spent:

10%:

0x3b9a47213f <\_int\_free+1023>: lock andl \$0xfffffffffffffffe,0x4(%r15)

9%:

0x3b9a472172 <\_int\_free+1074>: lock cmpxchg %rbx, (%rcx)

10%:

0x3b9a472a80 <\_int\_malloc+128>: lock cmpxchg %rdx,0x8(%rsi)

11%:

0x3b9a474e16 <malloc+86>: lock cmpxchg %edx, (%rbx)

40% of time in fence + RMW instructions