Reordering Constraints for Pthread-Style Locks

Hans-J. Boehm
HP Labs

The Problem

- Pthreads has been around for well over a decade, with many implementations. (And win32 threads are similar.)
- Most performance critical functions in pthreads are typically lock acquisition/release, e.g. `pthread_mutex_lock()`.
- Lock performance is highly dependent on what type of memory fences are included in these functions.
- It would be good to understand what fences are required by which calls.
- To get there, start with a review of pthreads rules …

Pthreads rules

No concurrent modification to shared variables (no races):

"Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. *[i.e. no data races.]* Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads…"

- Single Unix SPEC V3 & others

These functions include **`pthread_mutex_lock()`** …

- Seemingly independent of language specification.
- Problematic (see PLDI 05 paper), but …

Our (optimistic?) interpretation for this talk:

- Define two memory accesses to *conflict* if
  - They access the same *location* (i.e. variable for this talk).
  - At least one of them is a write.
  - They are executed by different threads.
- There is a *data race* if two conflicting actions can occur simultaneously in a *sequentially consistent* execution.
- Programs without data races have their sequentially consistent meaning.
- Programs with data races have undefined semantics.

Why no data races?

- Almost dodges *memory model* issues:

(Initially x = y = 0)

Thread 1	*Thread 2*
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

Can `r1` = `r2` = 0?

  - Intuitively (or under sequential consistency) no; some thread executes first.
  - In practice, yes; compilers and hardware can reorder.

- Under pthreads rules this is simply illegal.
  – We don't really get to ask the question.

Sequential consistency for race-free programs

- Similar to Ada model.
- Explored by Adve and Hill (ISCA 90).
- Essentially the basis for pthreads.
- Basis for current Java memory model.
- Likely to be the basis for C++0x memory model?

How Pthreads Implementations work (or should work)

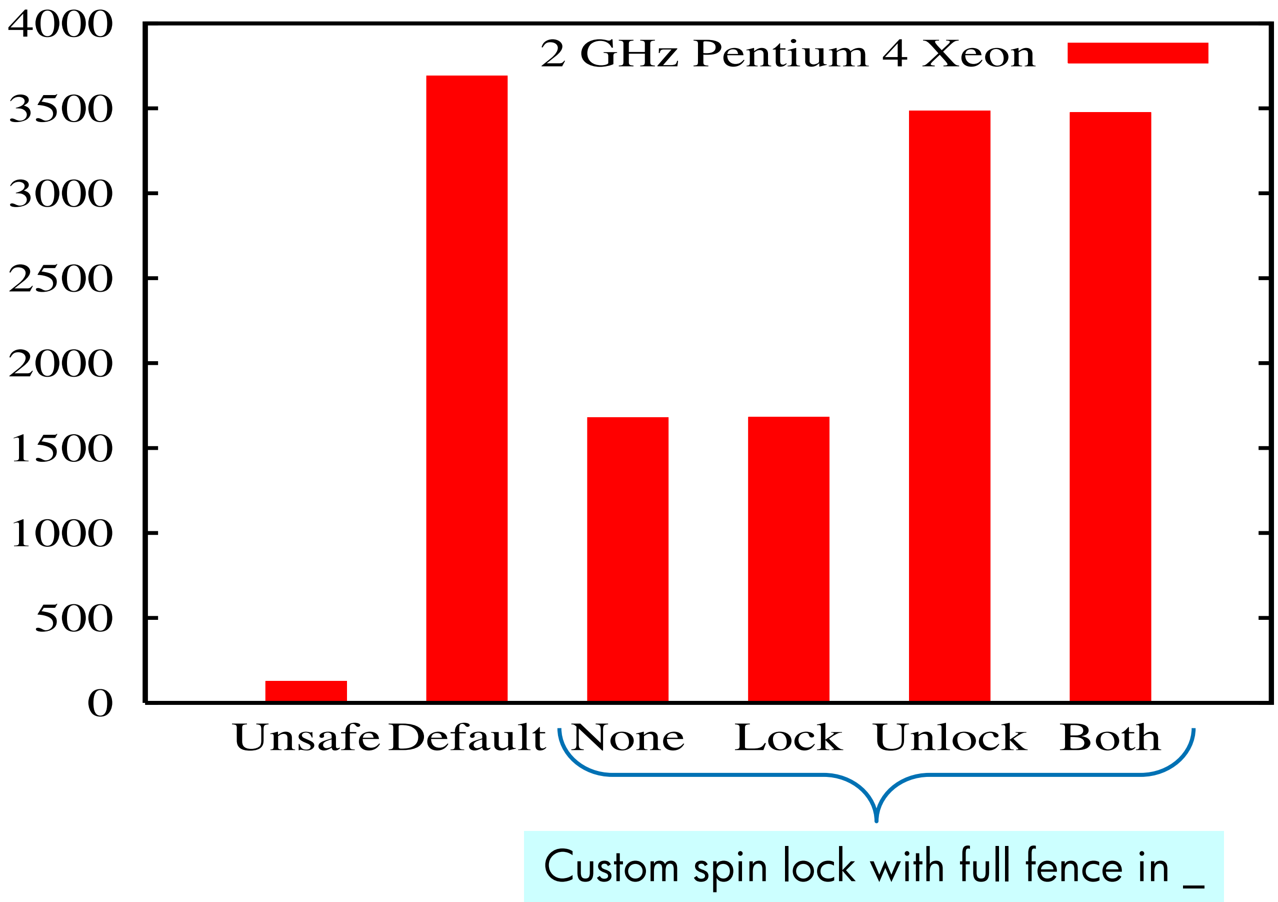
- Synchronization-free code is treated roughly as though it were single-threaded.
  - *Some optimization restrictions* (see PLDI 05 paper).
- Synchronization functions contain any needed hardware memory fences.
- Synchronization functions limit reordering with other memory operation.
  - Traditionally by viewing them as opaque
    - can potentially read or write any potentially shared variable.
  - Limits all movement.
  - But is really too strong.

Our goal

- Understand the allowable reordering of memory accesses and lock operations.
- We do this by looking at program transformations.
- But we are really interested in both hardware and software reordering.
- And most of the practical impact is on fences in lock operations.

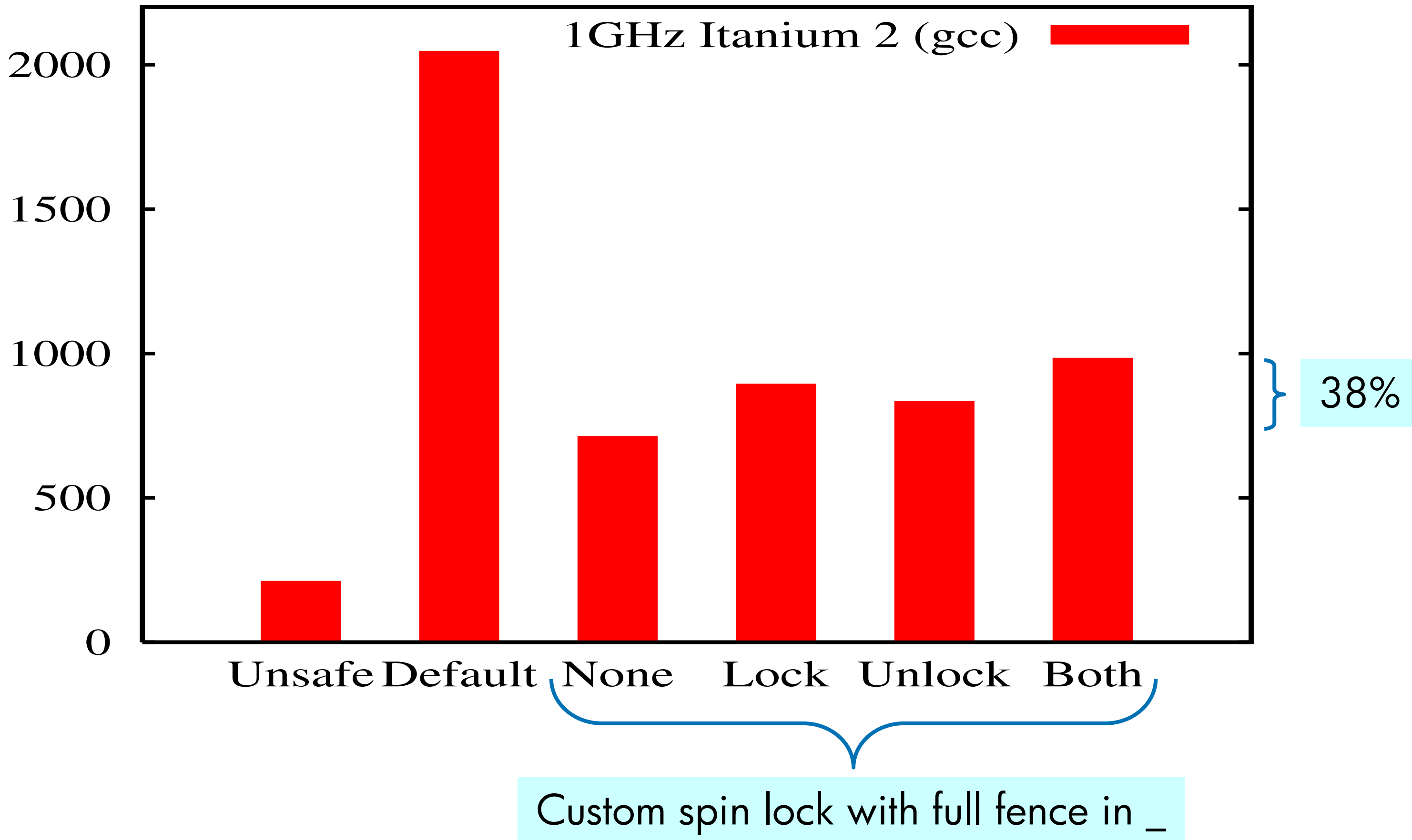
Cost of fences in lock() and/or unlock()

Msecs to copy 10MB with `putc`()/`getc`() (1 thread)



And on Itanium

Msecs to copy 10MB with `putc`()/`getc`() (1 thread)



Basic reordering rules as generally believed:

- Compiler/hardware can reorder non-locking instructions, so long as this is correct for 1 thread:

```
x = 1;                 r1 = y;
r1 = y;      ——→       x = 1;
```

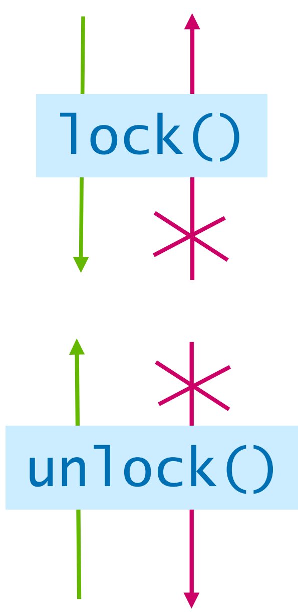
- Moving code out of critical sections is bad:

```
pthread_mutex_lock(…);              pthread_mutex_lock(…);
x++;                    ——✗——→      pthread_mutex_unlock(…);
pthread_mutex_unlock(…);            x++;
```

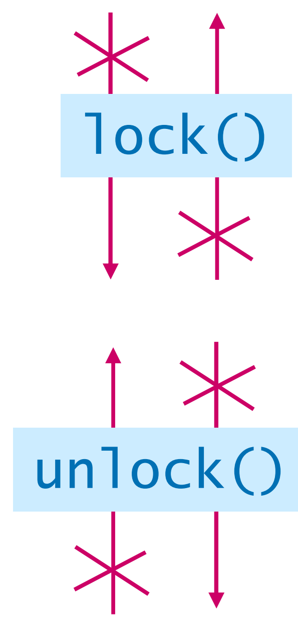
Movement of memory operations into critical sections is more interesting

- The obvious possibilities:

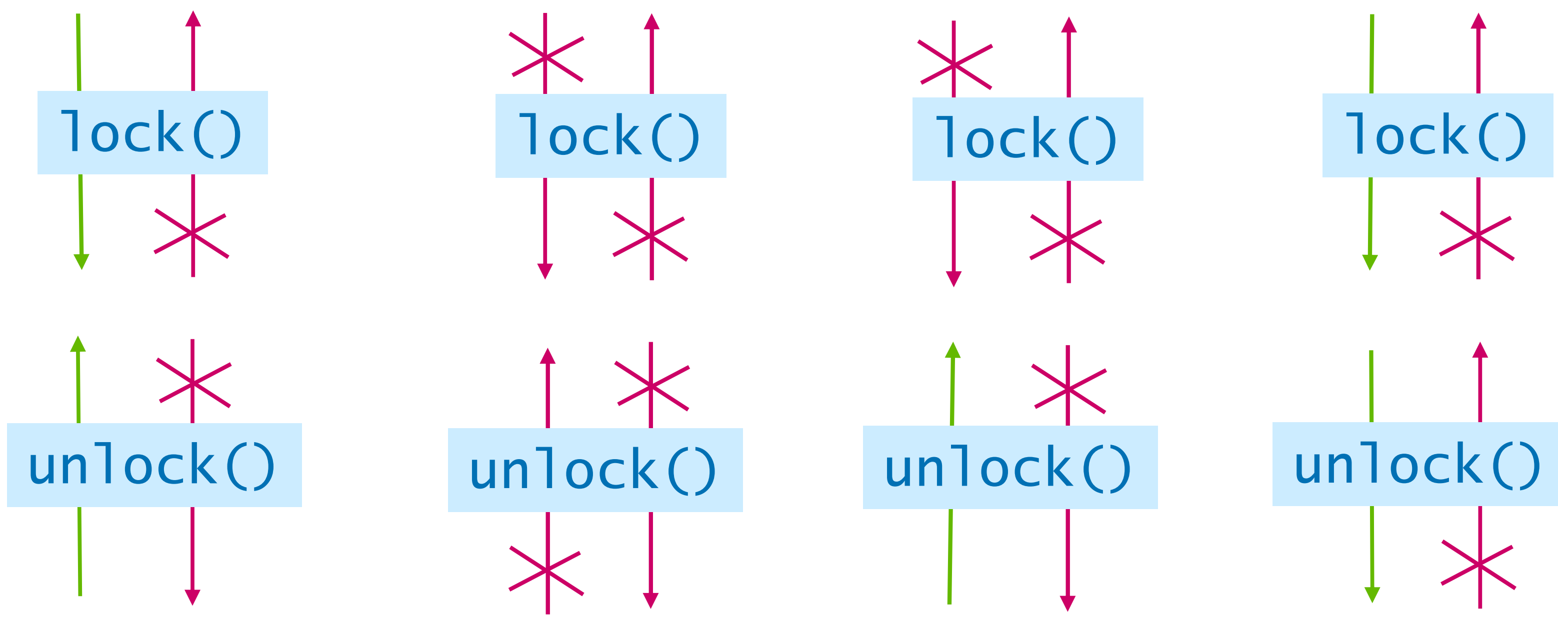
Java



Naïve pthreads ("synchronize memory")
Really required? Observable?



Some open source pthread lock implementations (2006):



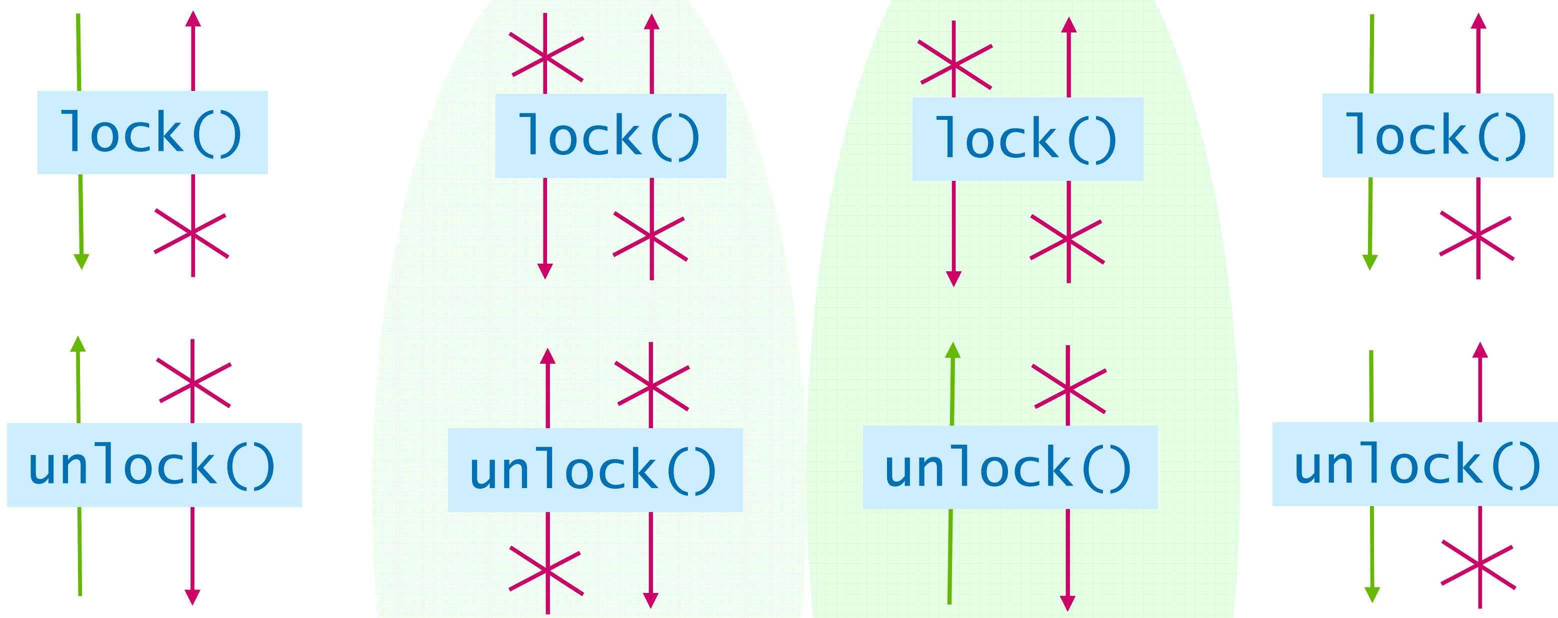
NPTL
{Alpha, PowerPC}
{mutex, spin}

NPTL
Itanium (&X86)
mutex

NPTL
{ Itanium, X86 }
spin

[Incorrect]
FreeBSD
Itanium
spin

And the right answer is:



[Technically incorrect]
NPTL
{Alpha, PowerPC}
{mutex, spin}

NPTL
Itanium (&X86)
mutex

NPTL
{ Itanium, X86 }
spin

[Incorrect]
FreeBSD
Itanium
spin

What this means:

- Moving memory operations into a critical section past `pthread_mutex_lock()` is observable.
- Moving memory operations into a critical section past `pthread_mutex_unlock()` is not observable.

Contributions of the paper:

- Set up a framework in which these questions can be analyzed.
- Prove some of the boring theorems that we all assume:
  - Reordering of independent memory operations is safe.
  - performing later memory operations before `unlock` is safe.
  - And hence `unlock` does not need a full fence.
- Show that performing earlier memory operations after lock leads to non-sequentially-consistent executions of race-free programs.

Formal Setting

- We phrase everything in terms of source transformations.
  - In a highly simplified source language.
- We reason in terms of sequentially consistent executions, i.e. interleavings of individual thread executions.
- To prove the validity of a transformation T, we need to show:
  - T preserves data-race-freedom
    - Doesn't generate undefined behavior.
  - For every sequentially consistent execution of the transformed program, there is an equivalent execution of the original program.
- By reasoning about source reorderings, we dodge architecture-dependent issues of fence semantics.

Reordering past lock: A counterexample

- *Insight:* In the presence of `try_lock`, e.g. `pthread_mutex_trylock()`, it is possible to invert the sense of a lock:
  - We can wait for a lock to be *acquired*, not released.

```
Thread 1:  v = 42;
           lock(l);
Thread 2:  while (try_lock(l))
               lock(l);
           r2 = v;  // No race!
                    // Must be 42
```

Cannot be reordered! (Thread 1: `v = 42;` and `lock(l);`)

Why Java is different (kind of)

- Java
  - Always allows movement *into* locked regions.
  - Still claims sequential consistency for race-free programs.
- The difference is in the definition of "data race":
  - Java requires conflicting operations to be "happens-before" ordered to avoid race.
  - We simply require no concurrent (or adjacent) execution.
  - Accesses to shared variable in last example are not happens-before ordered!

Practical implications

- We need agreement on fence implications of locks for performance comparisons to be meaningful.
- The strict pthreads requirements
  - Appear to have been accidental.
  - Do lead to slightly simpler programming rules.
    - But only when you use `try_lock`.
  - Result in an otherwise needless performance penalty.
- Currently it looks like C++0x will follow the Java model here.

invent