Languages and concurrency

a thorny relationship

Francesco Zappa Nardelli

Inria, France

Based on work done by or with

Vafeiadis, Sewell, Sevcik, Sarkar, Ridge, Owens, Morisset, Memar, Maranget, Chakraborty, Braibant, Balabonski, Batty, Alglave

U. Cambridge, U. Kent, MPI-SWS, Inria
In computer hardware, **shared memory** refers to a (typically) large block of **random access memory** (RAM) that can be accessed by several different **central processing units** (CPUs) in a multiple-processor computer system.

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. The issue with shared memory systems is that many CPUs need fast access to memory and will likely **cache memory**, which has two complications:

- **CPU-to-memory connection** becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.
- **Cache coherence**: Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data (see cache coherence and memory coherence). Such coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processors. On the other hand they can sometimes become overloaded and become a bottleneck to performance.
Shared memory

(according to Wikipedia)

...relatively easy to program...

...all processors share a single view of data...

...bottleneck to performance...

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. The issue with shared memory computer systems is that CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.

- Cache coherence: Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data (see cache coherence and memory coherence). Such coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processes and become a bottleneck to performance.
Imagine an ideal world
Imagine an ideal world

Programmers and compilers cooperate to make great software
Constant propagation

A simple, and *innocuous*, optimisation:

Source code

\[
\begin{align*}
  x &= 14 \\
  y &= 7 - x / 2
\end{align*}
\]

Optimised code

\[
\begin{align*}
  x &= 14 \\
  y &= 7 - 14 / 2
\end{align*}
\]

\[
\begin{align*}
  x &= 14 \\
  y &= 0
\end{align*}
\]
Shared memory concurrency

Shared memory

\[ x = y = 0 \]

\[ x = 1 \]
\[ \text{if } (y == 1) \]
\[ \text{print } x \]

\[ \text{if } (x == 1) \{ \]
\[ x = 0 \]
\[ y = 1 \} \]

Thread 1

Thread 2
Shared memory concurrency

Shared memory

\[ x = y = 0 \]

Thread 1

\[ x = 1 \]
\[ \text{if } (y == 1) \]
\[ \text{print } x \]

Thread 2

\[ \text{if } (x == 1) \{ \]
\[ x = 0 \]
\[ y = 1 \} \]

Intuitively this program always prints 0
Shared memory concurrency

But if the compiler propagates the constant $x = 1$...

$$x = y = 0$$

Thread 1

$x = 1$

if ($y == 1$)

print $x$
Shared memory concurrency

But if the compiler propagates the constant \( x = 1 \)...

\[
x = y = 0
\]

Thread 1

\[ x = 1 \]

\[ \text{if} \ (y == 1) \]

\[ \underline{\text{print } x} \]

\[ \underline{\text{print 1}} \]

Thread 2

\[ \text{if} \ (x == 1) \}

\[ x = 0 \]

\[ y = 1 \}

...the program always writes 1 rather than 0.
Shared memory concurrency

But if the compiler propagates the constant $x = 1$...

$x = y = 0$

A compiler can break your code

...the program always writes 1 rather than 0.
That pesky hardware (1)

Consider misaligned 4-byte accesses:

\[
\begin{align*}
\text{int32}_t & \quad a = 0 \\
0x4432211 & \quad \text{if } (a == 0x00002211) \\
\text{print } "\text{error}" 
\end{align*}
\]

(*Disclaimer*: compiler will normally ensure alignment)

Intel SDM x86 atomic accesses:

- \( n \)-bytes on an \( n \)-byte boundary (\( n = 1,2,4,16 \))
- P6 or later: … or if unaligned but within a cache line

*Question*: what about *multi-word high-level language values*?
That pesky hardware (2)

Initial shared memory values: \([x] = 0\) \([y] = 0\)

Per-processor registers: EAX EBX

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ([x]) ← 1</td>
<td>MOV ([y]) ← 1</td>
</tr>
<tr>
<td>MOV EAX ← ([y])</td>
<td>MOV EBX ← ([x])</td>
</tr>
</tbody>
</table>

Can you guess the final register values: EAX = ? EBX = ?
That pesky hardware (2)

Initial shared memory values: \( [x] = 0 \quad [y] = 0 \)

Per-processor registers: \( \text{EAX} \quad \text{EBX} \)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ([x]) ← 1</td>
<td>MOV ([y]) ← 1</td>
</tr>
<tr>
<td>MOV EAX ← ([y])</td>
<td>MOV EBX ← ([x])</td>
</tr>
</tbody>
</table>

Can you guess the final register values: \( \text{EAX} = 1 \quad \text{EBX} = 1 \)
That pesky hardware (2)

Initial shared memory values: \([x]=0\quad[y]=0\)

Per-processor registers: EAX EBX

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ([x]) ← 1</td>
<td>MOV ([y]) ← 1</td>
</tr>
<tr>
<td>MOV EAX ← ([y])</td>
<td>MOV EBX ← ([x])</td>
</tr>
</tbody>
</table>

Can you guess the final register values: \(EAX = 1\quad EBX = 1\)
That pesky hardware (2)

Initial shared memory values: \([x]=0 \quad [y]=0\)

Per-processor registers: EAX EBX

Can you guess the final register values: EAX = 1 EBX = 1
That pesky hardware (2)

Initial shared memory values:  \([x]=0 \quad [y]=0\)

Per-processor registers:  \texttt{EAX} \quad \texttt{EBX}

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ([x]) ← 1</td>
<td>MOV ([y]) ← 1</td>
</tr>
<tr>
<td>MOV EAX ← ([y])</td>
<td>MOV EBX ← ([x])</td>
</tr>
</tbody>
</table>

Can you guess the final register values:  \texttt{EAX = 1} \quad \texttt{EBX = 1}
That pesky hardware (2)

Initial shared memory values: \[ x = 0 \quad y = 0 \]

Per-processor registers: \( \text{EAX} \quad \text{EBX} \)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

Can you guess the final register values: \( \text{EAX} = 1 \quad \text{EBX} = 0 \)
That pesky hardware (2)

Initial shared memory values: \( [x] = 0 \quad [y] = 0 \)

Per-processor registers: \( EAX \quad EBX \)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ( [x] ) ← 1</td>
<td>MOV ( [y] ) ← 1</td>
</tr>
<tr>
<td>MOV EAX ← ( [y] )</td>
<td>MOV EBX ← ( [x] )</td>
</tr>
</tbody>
</table>

Can you guess the final register values: \( EAX = 0 \quad EBX = 1 \)
That pesky hardware (2)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

The possible outcomes should be:

- EAX : 1, EBX : 1
- EAX : 0, EBX : 1
- EAX : 1, EAX : 0
That pesky hardware (2)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x]</td>
<td>MOV [y]</td>
</tr>
<tr>
<td>MOV EAX</td>
<td>MOV EBX</td>
</tr>
</tbody>
</table>

The possible outcomes should be:

- EAX : 1, EBX : 1
- EAX : 0, EBX : 1
- EAX : 1, EAX : 0

Let's see...
That pesky hardware (2)

Thread 0

| MOV [x] ← 1 | MOV [y] ← 1 |
| MOV EAX ← [y] | MOV EBX ← [x] |

We can observe

\[ EAX = EBX = 0 \] as well
Hardware store buffering

Store buffers hide the latency of memory writes
Hardware store buffering

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

Diagram: Hardware store buffering

- Thread $1$
  - Write Buffer $1$
  - Lock
  - Shared Memory
- ... (Multiple threads)
- Write Buffer $n$
  - Shared Memory
Hardware store buffering

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ([x]) \leftarrow 1</td>
<td>MOV ([y]) \leftarrow 1</td>
</tr>
<tr>
<td>MOV EAX \leftarrow ([y])</td>
<td>MOV EBX \leftarrow ([x])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread(_1)</th>
<th>Thread(_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread(_1)</td>
<td>Thread(_n)</td>
</tr>
<tr>
<td>Write Buffer(_1)</td>
<td>Write Buffer(_n)</td>
</tr>
<tr>
<td>(W x 1)</td>
<td>(W y 1)</td>
</tr>
<tr>
<td>Lock</td>
<td>Shared Memory</td>
</tr>
</tbody>
</table>

Wednesday 5 August 15
Hardware store buffering

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

---

**Diagram:**

- Thread 0 and Thread 1
- MOV [x] ← 1
- MOV EAX ← [y]
- MOV [y] ← 1
- MOV EBX ← [x]
Hardware store buffering

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

![Diagram of hardware store buffering]
That pesky hardware (3)

...and differ between architectures...

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>print y</td>
</tr>
<tr>
<td>y = 1</td>
<td>print x</td>
</tr>
</tbody>
</table>

On x86, we only get

```
0 0
1 1
```

is printed on the screen.
That pesky hardware (3)

...and differ between architectures...

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>print y</td>
</tr>
<tr>
<td>y = 1</td>
<td>print x</td>
</tr>
</tbody>
</table>

On IBM Power or ARM

1 0

can be printed on the screen.
The fundamental problem
The fundamental problem

The programmer wants to understand the code he writes
The fundamental problem

The programmer wants to understand the code he writes

The compiler - and the hardware - try hard to optimise it
The fundamental problem

Which are the valid optimisations that the compiler or the hardware can perform without breaking the expected semantics of a concurrent program?

Which is the semantics of a concurrent program?

The programmer wants to understand the code he writes

The compiler - and the hardware - try hard to optimise it
Not new

Multiprocessors since 1964 (Univac 1108A - or Burroughs, in ‘62)

Relaxed Memory since 1972 (IBM System 370/158MP)

Eclipsed for a long time (except in high-end) by advances in performance:
  - transistor counts (continuing)
  - clock speed (hit power dissipation limit)
  - ILP (hit smartness limit?)
Mass market multiprocessors since 2005

Intel Xeon E7
up to 20 hardware threads

IBM Power 795 server
up to 1024 hardware threads

Best quad core phone: 4 contenders examined

EARLY VIEW HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad
Mass market multiprocessors since 2005

Intel Xeon E7
up to 20 hardware threads

Programming multiprocessors
no longer just for specialists
Topics

1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation
Topics

1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation
Architectures

Hardware manufacturers document architectures:

• *loose* specifications

• claimed to cover a *wide range* of past and future processor implementations.

Architectures should:

• *reveal enough* for effective programming;

• *without unduly constraining* future processor design.

*Examples*: Intel 64 and IA-32 Architectures SDM, AMD64 Architecture Programmer’s Manual, Power ISA specification, ARM Architecture Reference Manual, ...
In practice

Architectures described by informal prose:

In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.

(Intel SDM, November 2006, vol3a, 10-5)

As we shall see, such descriptions are:

1) vague; 2) incomplete; 3) unsound.

 Fundamental problem: prose specifications cannot be used to test programs or to test processor implementations.
1. spin_unlock() Optimization On Intel

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the 'lock: btl $0,%0' asm code, to 1 tick for a simple 'movl $0,%0' instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linux Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writing being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in some future versions of the Pentium, Manfred was right. However, he quoted in turn from the Pentium Pro Manual, Vol 3, Chapter 19.2 Memory Access Ordering, to optimize performance, the Pentium processor allows memory reads to be reordered ahead of buffered writes in most situations. Internally, CPU reads (cache hits) can be reordered around buffered writes. Memory reordering does not occur at the pins, reads (cache miss) and writes appear in-order.) He concluded from this that the second CPU would never see the spin_unlock() before the "bta" line. Linus agreed that on a Pentium, Manfred was right. However, he quoted in turn from the Pentium Pro manual, "The only enhancement in the PentiumPro processor is the added support for speculative reads and store-buffer forwarding." He explained:

A Pentium is a in-order machine, without any of the interesting speculation wrt reads etc. So on a Pentium you'll never see the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

the window is probably very very small, and you have to be unlucky to hit it.

Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "bta" may be the worst possible thing to use for this, and you might test whether a simpler "xor+xchg!" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to waste on serializing instructions rather than

... (repeated)
1. spin_unlock() Optimization On Intel

Wednesday 5 August 15

spin_unlock();

spin_lock();

a = 1;

/* cache miss satisfied, the "a" line is bouncing back and forth */

b gets the value 1

a = 0;

and it returns "1", which is wrong for any working spinlock.

Unlikely? Yes, definitely. Something we are willing to live with as a potential bug in any real kernel? Definitely not.

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the 'lock; btl $0,5/0' asm code, to 1 tick for a simple 'movl $0,5/0' instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you have to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical region (maybe it missed a cache line), and get a stale value for any of the reads that _should_ have been serialized by the spinlock.

spin_unlock();

spin_lock();

a = 1;

/* cache miss satisfied, the "a" line is bouncing back and forth */

b = a; /* cache miss, we'll delay this.. */

But a Pentium is also very uninteresting from a SMP standpoint these days. It's just too weak with too little per-CPU cache etc...

This is why the PPro has the MTRR's exactly to let the core do speculation (a Pentium doesn't need MTRR's, as it won't re-order anything external to the CPU anyway, and in fact won't even re-order things internally).

Jeff V. Merkey added:

What Linus says here is correct for PPro and above. Using a mov instruction to unlock does work fine on a 486 or Pentium SMP system, but as of the PPro, this was no longer the case, though the window was so infinitesimally small, most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is writtne to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the pipelines are no longer invalid and the buffers should be blown out.

I have seen the behavior Linux describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PROPO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, Erich replied:

It will always return 0. You don't need "spin_unlock()" to be serializing.

The only thing you need to is make sure there is a store in "spin_unlock()", and that is kind of true by the fact that you're changing something to be observable on other processors.

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed by that time), so the any loads, stores, etc absolutely have to have completed first, cache-miss or not.

He went on:

Since the instructions for the store in the spin_unlock have to have been externally observed for spin_lock to be aquired (presuming a correctly functioning spinlock, of course), then the earlier instructions to set "b" to the value of "a" have to have completed first.

In general, Intel is Processor Ordered for cacheable accesses. Speculation doesn't affect this. Also, stores are not observed speculative on other processors.

There was a long clarification discussion, resulting in a complete turnaround of Linus:

Everybody has convinced me that yes, the Intel ordering rules _are_ strong enough that all of this reality is legal, and that's what I wanted. I've gotten some explanations for why serialization (as opposed to just the simple locked access) is required for the lock() side but not the unlock() side, and that lack of symmetry was what bothered me the most.

Oliver made a stronger case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.

Thanks, guys, we'll be that much faster due to this...
Manfred Spraul thought he’d found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl $0,0,%0" asm code, to 1 tick for a simple "movl $0,0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:  

It does NOT WORK!

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren’t reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won’t be the case in the long run). The issue is that you _have_ to have a serializing instruction in order to make sure that the processor doesn’t re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical region (maybe it missed a cache line), and yet a stale value for any of the reads that _should_ have been serialized by the spinlock.

spin_unlock();

spin_lock();

a = 1; /* cache miss satisfied, the "a" line is bound to be returned */
b gets the value 1

a = 0; and it retains 0

Unlikely bug in a real case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.

We can shave spin_unlock() down from about 22 ticks for the "lock; btrl $0,0,%0" asm code, to 1 tick for a simple "movl $0,0,%0" instruction, a huge gain.

spin_unlock();

spin_lock();

a = 1; /* cache miss satisfied, the "a" line is bound to be returned */
b gets the value 1

a = 0; and it retains 0

Unlikely bug in a real case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.

The only thing you need to do is make sure there is a store in "spin_unlock()", and that is kind of true by the fact that you’re changing something to be observable on other processors.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it’s just that it no longer works reliably as an exclusion thing)

the window is probably very very small, and you have to be unlucky to hit it. Faster CPU’s, different compilers, whatever.

I might be proven wrong, but I don’t think I am.

Note that another thing is that yes, "bctl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+chg" might be better - it’s still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to waste on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imaging running the following test in a loop on multiple CPU’s:

```c
int test_locking(void){
    static int a; /* protected by spinlock */
    int b;

    a = 1; /* cache miss satisfied, the "a" line is bound to be returned */
b gets the value 1

    a = 0; and it retains 0

    Unlikely bug in a real case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.

    The only thing you need to do is make sure there is a store in "spin_unlock()", and that is kind of true by the fact that you’re changing something to be observable on other processors.
```

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed by that time), so the any loads, stores, etc absolutely have to have completed first, cache-misses or not.

He went on:

Since the instructions for the store in the spin_unlock have to have been externally observed for spin_lock to be acquired (assuming a correctly functioning spinlock, of course), then the earlier instructions to set "b" to the value of "a" have to have completed first.

In general, IA32 is Processor Ordered for cacheable accesses. Speculation doesn’t affect this. Also, stores are not observed speculatively on other processors.

There was a long clarification discussion, resulting in a complete turnaround by the following test in a loop on multiple CPU’s:

```c
int test_locking(void){
    static int a; /* protected by spinlock */
    int b;
```

```c
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```
Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed, in random ways.

Note that the fact that it does not crash now is quite possibly because of either we have a lot less contention on our spinlocks these days. That might hide the problem, because the spinlock will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing).

the window is probably very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "btcl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+exchg!" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to want to use on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imagine running the following test in a loop on multiple CPU's:

spin_lock()
a = 1;
mb();
a = 0;
mb();
spun_unlock();
return b;
}

Now, OBLVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

And yes, the above CAN return 1 with the proposed optimization. I doubt you can make it do so in real life, but hey, add another access to another variable in the same cache line that is accessed through another spinlock (to get cache-line ping-pong and timing effects), and I suspect you can make it happen even with a simple example like the above.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing any more, so there is very little effective ordering between the two actions

b = a;spin_unlock();
as they access completely different data (ie no data dependencies in sight).
So although back and forth is equivalent to

spun_lock();

We can shave about 22 ticks of asm code, to 1 tick for a simple "movl $0,%0" instruction, a huge gain.

4% speed-up in a benchmark test, making the optimization very valuable. The same optimization cropped up in the FreeBSD mailing list.

spin_unlock();

spin_lock();
a = 1;
/* cache miss satisfied, the "a" line is bound
b gets the value 1
and it reads it."

Unlike a bug in a

Unrelated

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btl $0,%0" asm code, to 1 tick for a simple "movl $0,%0" instruction, a huge gain.

Note that the fact that it does not crash now is quite possibly because of either we have a lot less contention on our spinlocks these days. That might hide the problem, because the spinlock will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing).

the window is probably very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "btcl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+exchg!" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to want to use on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imagine running the following test in a loop on multiple CPU's:

spin_lock()
a = 1;
mb();
a = 0;
mb();
spun_unlock();
return b;
}

Now, OBLVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

And yes, the above CAN return 1 with the proposed optimization. I doubt you can make it do so in real life, but hey, add another access to another variable in the same cache line that is accessed through another spinlock (to get cache-line ping-pong and timing effects), and I suspect you can make it happen even with a simple example like the above.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing any more, so there is very little effective ordering between the two actions

b = a;spin_unlock();
as they access completely different data (ie no data dependencies in sight).
So although back and forth is equivalent to

spun_lock();

We can shave about 22 ticks of asm code, to 1 tick for a simple "movl $0,%0" instruction, a huge gain.

4% speed-up in a benchmark test, making the optimization very valuable. The same optimization cropped up in the FreeBSD mailing list.

spin_unlock();

spin_lock();
a = 1;
/* cache miss satisfied, the "a" line is bound
b gets the value 1
and it reads it."

Unlike a bug in a

Unrelated

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btl $0,%0" asm code, to 1 tick for a simple "movl $0,%0" instruction, a huge gain.

Note that the fact that it does not crash now is quite possibly because of either we have a lot less contention on our spinlocks these days. That might hide the problem, because the spinlock will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing).

the window is probably very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "btcl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+exchg!" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to want to use on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imagine running the following test in a loop on multiple CPU's:

spin_lock()
a = 1;
mb();
a = 0;
mb();
spun_unlock();
return b;
}

Now, OBLVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

And yes, the above CAN return 1 with the proposed optimization. I doubt you can make it do so in real life, but hey, add another access to another variable in the same cache line that is accessed through another spinlock (to get cache-line ping-pong and timing effects), and I suspect you can make it happen even with a simple example like the above.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing any more, so there is very little effective ordering between the two actions

b = a;spin_unlock();
as they access completely different data (ie no data dependencies in sight).
So although back and forth is equivalent to

spun_lock();

We can shave about 22 ticks of asm code, to 1 tick for a simple "movl $0,%0" instruction, a huge gain.
1. **spin_unlock() Optimization On Intel**

20Nov1999-Tiomic999 (143 posts) Archive Link: "spin_unlock optimization(186)"

**People:** Linus Torvalds, Jeff V, Markus Erich Boley-Manfred Spraul-Peter Samuelson-Ingo Molnar

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the 'lock: btrl $0,%0' asm code, to 1 tick for a simple 'movl $0,%0' instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

*It is NOT WORK!* Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behavior - I bet it won't be the case in the long run).

The issue is that you _have_ to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can lerrly delay a read that happened inside the critical state value for any of the re-spinlock.

```
spin_unlock();

spin_lock();

a = 1; /* cache miss satisfied, the */
b gets the value 1

a = 0; /* and it re-order */
```

Unlikely bug in a Pentium E633.

Manfred, in his Pentium Manual, writes in around 8 (cache misses) CPU and a Pentium manual, for speculative reads and writes etc. So on a Pentium, you ALWAYS have the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

```
spin_lock()

a = 1;
mb();
mb();

b = a;
spin_unlock();
return b;
```

Now, **OBVIOUSLY** the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

```
spin_unlock()

a = 0;
```

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to waste on serializing instructions rather than use for this, and you might test whether a simpler "xor+xchgl" might be equivalent as an example, imaging running the following test in a loop on multiple CPU's:

```
{x}

spin_lock()

spin_unlock();

b = a; /* cache miss, we’ll delay this.. */

{y}

b = a; /* cache miss satisfied, the */
```

It does NOT WORK!

```
spin_lock()

a = 1;
mb();
mb();

b = a;
spin_unlock();
return b;
```

The FreeBSD periodicity was what bothered me the most. Everybody has come to believe that only through fully synthesizing and then trying a Pentium optimizing the spinlock, a huge gain.

But a Pentium is also very uninteresting from a SMP standpoint these days.

Linus, pointing out a possible misconception in his proposed exploit. Regarding Linus posted, Erich replied:

*It is NOT WORK!* Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The FreeBSD mailing list.

```
spin_lock()

a = 1;
mb();
mb();

b = a;
spin_unlock();
return b;
```

It does NOT WORK!

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

```
spin_lock()

a = 1;
mb();
mb();

b = a;
spin_unlock();
return b;
```

Faster CPU’s, different compilers, whatever.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

the window is probably very very small, and you have to be unlucky to hit it. Faster CPU’s, different compilers, whatever.
1. `spin_unlock()` Optimization On Intel

20Nov1999-Tbe@1999 (141 posts) Archive Link: "spin_unlock optimization" (186)

Topics: BSD - FreeBSD - MP
People: Linus Torvalds; Jeff V; Marcel-Eric Beleya-Manfred Spraul; Peter Samuelson; Ingo Molnar

Manfred Spraul thought he'd found a way to shave `spin_unlock()` down from about 22 ticks for the 'lock; btl; $0,$0,$0' asm code, to 1 tick for a simple 'movl $0,$0' instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

"It does NOT WORK!"

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you _have _to have a serializing instruction immediately after store to ensure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical section for any of the re-spinlock.

```
spin_unlock();
```

...but Linus posted, Erich replied:

"The window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

```
int b; /* cache miss, we'll delay this.. */
CPU#1:
spin_lock();
a = 1;
mb();
b = a;
spin_unlock();
return b;
```

Now, _obviously_ the above always has to return 0, right? All accesses to 
"a" are inside the spinlock, and we always set it to zero before we read it into 
"b" and return it. So if we _EVER_ returned anything else, the spinlock would 
obviously be completely broken, wouldn't you say?

It seems that the issue is that Intel has a very minor bug in its spinlock function:

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
```

The only enhancement in the PentiumPro processor is the added support
for speculative reads

"Bogus optimizations can work on single machine, without any of the interesting speculation
order machine, without any of the interesting speculation
order."

Linus, pointing out a possible misconception in his proposed exploit. Regarding
serializing any more, so there is very little effective or
Linus posted, Erich replied:

"I bet it won't be the
line ping pong and timing effects), and I suspect you can make it happen even
if you up example (which will probably never show the
instructive as an example), imagine running the
CPU cache etc.."

In general, IA32 is Processor Ordered for cacheable accesses. Speculation
miss or not.

"b" and return it. So if we _EVER_ returned anything else, the spinlock would
obviously be completely broken, wouldn't you say?

Let the FreeBSD people use it, and let them get faster timings. They will crash,
eventually.

Note that another thing is that yes, "btcl" may be the worst possible thing to
"a" are inside the spinlock, and we always set it to zero before we read it into
"b" and return it. So if we _EVER_ returned anything else, the spinlock would
obviously be completely broken, wouldn't you say?

Let the FreeBSD people use it, and let them get faster timings. They will crash,
eventually.

Note that another thing is that yes, "btcl" may be the worst possible thing to
"b" and return it. So if we _EVER_ returned anything else, the spinlock would
obviously be completely broken, wouldn't you say?

Let the FreeBSD people use it, and let them get faster timings. They will crash,
eventually.

Note that another thing is that yes, "btcl" may be the worst possible thing to
"b" and return it. So if we _EVER_ returned anything else, the spinlock would
obviously be completely broken, wouldn't you say?

Let the FreeBSD people use it, and let them get faster timings. They will crash,
eventually.

Note that another thing is that yes, "btcl" may be the worst possible thing to
"b" and return it. So if we _EVER_ returned anything else, the spinlock would
obviously be completely broken, wouldn't you say?

Let the FreeBSD people use it, and let them get faster timings. They will crash,
eventually.

Note that another thing is that yes, "btcl" may be the worst possible thing to
From the Pentium Pro manual, "The only enhancement in the PentiumPro processor is the added support for speculative reads and store-buffer forwarding."

It does NOT WORK!

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is that you have to have a serializing instruction in order to make sure that the processor doesn't re-order. The issue is not writes being issued in out-of-order machine, without any of the interesting speculation order anything external to the critical region (maybe it missed a cache line), and get a speculatively ordered output.

Oliver made a strobe.

Linus posted, Erich replied:

"a" are inside the spinlock, and we always set it to zero before we read it into the pins, reads (cache miss) and writes appear in-order. The window is probably very small, and you have to be unlucky to hit it. Faster CPUs, different compilers, whatever.

Jeff V. Merkey added:

I have seen the behavior Linus describes on a hardware analyzer, BUT it will always return 0. You don't need "spin_unlock()" to be serializing.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding a simple example like the above:

let b = a;

spin_lock();

a = 1;

mb();

a = 0;

mb();

b = a;

spin_unlock();

return b;

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

It does NOT WORK!

...but hey, add another access to another variable accessed through another spinlock (to get cache-misses and other effects), and I suspect you can make it happen even more.

The reason legally is that your new "spin_unlock()" is not serializing anything external to the critical region, so it is very little effective ordering between the two spinlocks.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing anything external to the critical region, so it is very little effective ordering between the two spinlocks.

A Pentium is a in-order machine wrt reads etc. So on a Pentium you always see the problem.

Linus, pointing out a possible misconception in his proposed exploit. Regarding a simple example like the above:

let b = a;

spin_lock();

a = 1;

mb();

a = 0;

mb();

b = a;

spin_unlock();

return b;

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

It does NOT WORK!

...but hey, add another access to another variable accessed through another spinlock (to get cache-misses and other effects), and I suspect you can make it happen even more.

The reason legally is that your new "spin_unlock()" is not serializing anything external to the critical region, so it is very little effective ordering between the two spinlocks.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing anything external to the critical region, so it is very little effective ordering between the two spinlocks.

A Pentium is a in-order machine wrt reads etc. So on a Pentium you always see the problem.
I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

The issue is not writes being issued out of order, or speculative reads, but the fact that the processor doesn't know this can bite in some cases.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical section. The write will appear out-of-order, but that's okay; the CPU is allowed to do that. However, the read that happens inside the critical section will be delayed. If that read accesses a shared variable that is being written to inside the critical section, it will only get the stale value for any of the reads that _should_ have been serialized by the lock.

The issue is that you _have_ to have a serializing instruction in order to make order write behaviour observable on other processors.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed, in random ways.

Note that the fact that it does not crash now is quite possibly because of either

- the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

> It does NOT WORK!

> Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

> The issue is not writes being issued out of order, or speculative reads, but the fact that the processor doesn't know this can bite in some cases.

> For example, with a simple write, the CPU can legally delay a read that happened inside the critical section. The write will appear out-of-order, but that's okay; the CPU is allowed to do that. However, the read that happens inside the critical section will be delayed. If that read accesses a shared variable that is being written to inside the critical section, it will only get the stale value for any of the reads that _should_ have been serialized by the lock.

> The issue is that you _have_ to have a serializing instruction in order to make order write behaviour observable on other processors.

> Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed, in random ways.

> Note that the fact that it does not crash now is quite possibly because of either

- the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

> It does NOT WORK!

> Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

> The issue is not writes being issued out of order, or speculative reads, but the fact that the processor doesn't know this can bite in some cases.

> For example, with a simple write, the CPU can legally delay a read that happened inside the critical section. The write will appear out-of-order, but that's okay; the CPU is allowed to do that. However, the read that happens inside the critical section will be delayed. If that read accesses a shared variable that is being written to inside the critical section, it will only get the stale value for any of the reads that _should_ have been serialized by the lock.

> The issue is that you _have_ to have a serializing instruction in order to make order write behaviour observable on other processors.

> Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed, in random ways.

> Note that the fact that it does not crash now is quite possibly because of either

- the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

> It does NOT WORK!

> Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.

> The issue is not writes being issued out of order, or speculative reads, but the fact that the processor doesn't know this can bite in some cases.

> For example, with a simple write, the CPU can legally delay a read that happened inside the critical section. The write will appear out-of-order, but that's okay; the CPU is allowed to do that. However, the read that happens inside the critical section will be delayed. If that read accesses a shared variable that is being written to inside the critical section, it will only get the stale value for any of the reads that _should_ have been serialized by the lock.

> The issue is that you _have_ to have a serializing instruction in order to make order write behaviour observable on other processors.
**1. spin_unlock() Optimization On Intel**

It will always return 0. You don't need "spin_unlock()" to be serializing.

Linus describes on a BUT ONLY ON WHERE PPRO AND 0 people must still be aware and that's why...

processor it was on older Pentium hardware they don't know speculatively.

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

Linus Torvalds poured cold water on the whole thing, saying:

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

It's just too weak with too little per

Linux is a high performance one and that's why

A Pentium is a high performance one and that's why

A 4/5 uses this method and reads and writes. (cache)

NB: We can shave spin_unlock() down from 22 ticks for the 'lock: btl 0,%0' asm code, to 1 tick for a simple 'movl 0,%0' instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Spinning on Intel

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into the spinlock. So, the spinlock would not return anything else. But a Pentium is also very uninteresting from a SMP standpoint these days. The FreeBSD people used to have a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into the spinlock. So, the spinlock would not return anything else. But a Pentium is also very uninteresting from a SMP standpoint these days. The FreeBSD people used to have a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into the spinlock. So, the spinlock would not return anything else. But a Pentium is also very uninteresting from a SMP standpoint these days. The FreeBSD people used to have a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into the spinlock. So, the spinlock would not return anything else. But a Pentium is also very uninteresting from a SMP standpoint these days. The FreeBSD people used to have a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into the spinlock. So, the spinlock would not return anything else. But a Pentium is also very uninteresting from a SMP standpoint these days. The FreeBSD people used to have a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.

a = 0;

int test_locking(void)
{
    extern int a; /* protected by spinlock */
    static int b;

    spin_lock();
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into the spinlock. So, the spinlock would not return anything else. But a Pentium is also very uninteresting from a SMP standpoint these days. The FreeBSD people used to have a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the._spinlock_ will be fine (the cache coherence still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

It does NOT WORK!

It's just that it no longer works reliably as an exclusion thing)

Note that the fact that it does not crash now is quite possibly because of either

...wrt reads etc. So on a Pentium you'll never see the problem.
It will always return 0. You don't need "spin_unlock()" to be serializing.

Thanks, guys, we'll be that much faster due to this!

I feel comfortable again.

It's just too weak with too little performance.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

I have seen the behavior Linus describes on a hardware analyzer, BUT it will always return 0. You don't need "spin_unlock()" to be serializing.

I bet it won't be the spinlock. In general, IA32 is Processor Ordered for cacheable accesses. Speculation forwarding.

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed). The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until the earlier instructions are already committed).

Linus describes a hardware behavior where the CPU can legally delay a read that happened inside the critical section.

As an example, imaging running the following code: int b; int test_locking(void) { spin_lock(); b = a; spin_unlock(); return b; } It does NOT WORK!

It will always return 0. You don't need "spin_unlock()" to be serializing.

Thanks, guys, we'll be that much faster due to this!

I feel comfortable again.

It's just too weak with too little performance.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

I have seen the behavior Linus describes on a hardware analyzer, BUT it will always return 0. You don't need "spin_unlock()" to be serializing.

I bet it won't be the spinlock. In general, IA32 is Processor Ordered for cacheable accesses. Speculation forwarding.

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until the earlier instructions are already committed). The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until the earlier instructions are already committed).

Linus describes a hardware behavior where the CPU can legally delay a read that happened inside the critical section.

As an example, imaging running the following code: int b; int test_locking(void) { spin_lock(); b = a; spin_unlock(); return b; } It does NOT WORK!

It will always return 0. You don't need "spin_unlock()" to be serializing.

Thanks, guys, we'll be that much faster due to this!

I feel comfortable again.

It's just too weak with too little performance.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

I have seen the behavior Linus describes on a hardware analyzer, BUT it will always return 0. You don't need "spin_unlock()" to be serializing.

I bet it won't be the spinlock. In general, IA32 is Processor Ordered for cacheable accesses. Speculation forwarding.

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until the earlier instructions are already committed). The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until the earlier instructions are already committed).

Linus describes a hardware behavior where the CPU can legally delay a read that happened inside the critical section.

As an example, imaging running the following code: int b; int test_locking(void) { spin_lock(); b = a; spin_unlock(); return b; } It does NOT WORK!

It will always return 0. You don't need "spin_unlock()" to be serializing.

Thanks, guys, we'll be that much faster due to this!

I feel comfortable again.

It's just too weak with too little performance.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

I have seen the behavior Linus describes on a hardware analyzer, BUT it will always return 0. You don't need "spin_unlock()" to be serializing.
Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to compute dependencies and to define the semantics of barriers.
Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to compute dependencies and to define the semantics of barriers.

The definition of performed refers to an hypothetical store by P2.

A memory model should define if a particular execution is allowed. It is awkward to make a definition that explicitly quantifies over all hypothetical variant executions.
Power ISA

Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

The definition of performed refers to an hypothetical store by P2.

A memory model should define if a particular execution is allowed.

It is is awkward to make a definition that explicitly quantifies over all hypothetical variant executions.

See Alglave et al., PLDI, 2011.
A way out?
Way out? Create *rigorous* memory models

- Unambiguous

- Sound w.r.t. experience

- Consistent with what we know of vendor intentions
Way out? Create rigorous memory models

• Unambiguous

  mathematical language

• Sound w.r.t. experience

• Consistent with what we know of vendor intentions
Way out? Create *rigorous* memory models

• Unambiguous
  
  *mathematical language*

• Sound w.r.t. experience
  
  *rigorous testing of the model against the hardware*

• Consistent with what we know of vendor intentions
Way out? Create rigorous memory models

• Unambiguous

  mathematical language

• Sound w.r.t. experience

  rigourous testing of the model against the hardware

• Consistent with what we know of vendor intentions

  interaction with hardware developers
Mathematical language

- Operational and/or axiomatic models
- About 1k LOS, beyond comfortable pencil-and-paper math
- Events, sets, relations, partial orders
- No interesting syntax, no binding, no need for fancy types (scarcely HO)

Want reusable specifications!
LEM: a DSL for discrete-math definitions

You write:

• definitions of types, functions, inductive relations

• with quantifiers, set comprehensions, and top-level type polymorphism
  (roughly intersection of HOL4, Isabelle/HOL, and Coq)

LEM gives you:

• type-checking of the definitions

• decent typesetting

• whitespace-preserved prover definitions in HOL4, Isabelle/HOL (&Coq?)

• OCaml code (ind.rel.?) (Haskell?)
LEM: a DSL for discrete-math definitions

Example taken from the IBM POWER memory model

```ocaml
let write_reaching_coherence_point_action m s w =
  let writes_past_coherence_point' =
    s.writes_past_coherence_point union {w} in
  let coherence' = s.coherence union
    { (w,wother) | forall (wother IN (writes_not_past_coherence s)) |
      (not (wother = w)) && (wother.w_addr = w.w_addr) } in
  <| s with coherence = coherence';
    writes_past_coherence_point = writes_past_coherence_point' |>

let sem_of_instruction i ist =
  match i with
  | Padd set rD rA rB -> op3regs Add set rD rA rB ist
  | Pandi rD rA simm -> op2regi And SetCR0 rD rA (intToV simm) ist
```

* OCaml code (ind.rel.?) (Haskell?)
The ARM / IBM POWER memory model formalisation
Executing the specifications

Make the model accessible to programmers

Given a litmus test, compute the model-allowed executions:

- *operational*: search of abstract matching LTS
- *axiomatic*: enumerate all candidates, filter by axioms

Lem $\xrightarrow{Lem}$ OCaml $\xrightarrow{\text{search algorithm}}$ OCaml $\xrightarrow{\text{js_of_ocaml}}$ JavaScript

DEMO [ppcmem]
Testing the specifications

1. Systematically generate litmus tests out of the spec

2. Test them on real hardware and compare with the model

---

**WRC: Write to Read Causality**

<table>
<thead>
<tr>
<th>Test</th>
<th>Model</th>
<th>PowerG5</th>
<th>Power6</th>
<th>Power7</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRC</td>
<td>Allow</td>
<td>Ok, 44k/2.5G</td>
<td>Ok, 1.2M/13G</td>
<td>Ok, 25M/104G</td>
</tr>
<tr>
<td>WRC+data+addr</td>
<td>Allow</td>
<td>No, 0/3.3G</td>
<td>Ok, 705k/13G</td>
<td>Ok, 166k/105G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allow unseen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRC+syncs</td>
<td>Forbid</td>
<td>Ok, 0/3.3G</td>
<td>Ok, 0/17G</td>
<td>Ok, 0/157G</td>
</tr>
<tr>
<td>WRC+sync+addr</td>
<td>Forbid</td>
<td>Ok, 0/3.3G</td>
<td>Ok, 0/17G</td>
<td>Ok, 0/157G</td>
</tr>
<tr>
<td>WRC+lwsync+addr</td>
<td>Forbid</td>
<td>Ok, 0/3.3G</td>
<td>Ok, 0/17G</td>
<td>Ok, 0/137G</td>
</tr>
<tr>
<td>WRC+data+sync</td>
<td>Allow</td>
<td>No, 0/3.3G</td>
<td>Ok, 176k/13G</td>
<td>Ok, 75k/105G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allow unseen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRC+addr+ctrl</td>
<td>Allow</td>
<td>Ok, 43k/1.3G</td>
<td>Ok, 313k/4.3G</td>
<td>Ok, 4.5M/24G</td>
</tr>
<tr>
<td>WRC+addr+ctrlisync</td>
<td>Allow</td>
<td>No, 0/2.1G</td>
<td>Ok, 402k/4.3G</td>
<td>Ok, 69k/25G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allow unseen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRC+addr+isync</td>
<td>Allow</td>
<td>No, 0/2.1G</td>
<td>Ok, 403k/4.3G</td>
<td>Ok, 49k/25G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allow unseen</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testing the specifications

1. Systematically generate litmus tests out of the spec

2. Test them on real hardware and compare with the model

Rigourous testing and interaction with hardware architects to validate the formalisation of the memory models
These are abstract machines

A tool to specify exactly and only the programmer-visible behaviour, not a description of the implementation internals.

\[
\begin{align*}
\succ & \quad \text{beh} \\
\neq & \quad \text{hw}
\end{align*}
\]
Topics

1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation
Topics

1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation
The simplest memory model

*sequential consistency*
Sequential consistency

...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...

Compilers, programmers & sequential
Compilers, programmers & sequential

Simple and intuitive programming model
Compilers, programmers & sequential

Expensive to implement

Simple and intuitive programming model
An SC-preserving compiler, obtained by restricting the optimization phases in LLVM, a state-of-the-art C/C++ compiler, incurs an average slowdown of 3.8% and a maximum slowdown of 34% on a set of 30 programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites.
An SC-preserving compiler, obtained by restricting the optimization phases in LLVM, a state-of-the-art C/C++ compiler, incurs an average slowdown of 3.8% and a maximum slowdown of 34% on a set of 30 programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites.

This study assumes that the hardware is SC: these numbers are optimistic lower bounds.
The layman solution

*forbid data-races*
Data-race freedom

Our examples again:

- the problematic transformations (e.g. swapping the two writes in thread 0) do not change the meaning of single-threaded programs

- the problematic transformations are detectable only by code that allows two threads to access the same data simultaneously in conflicting ways (e.g. one thread writes the datas read by the other).

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ast y = 1 )</td>
<td>if ( \ast x == 1 )</td>
</tr>
<tr>
<td>( \ast x = 1 )</td>
<td>then print ( \ast y )</td>
</tr>
</tbody>
</table>

Observable behaviour: 0
Data-race freedom

Our examples again:

- the problematic transformations (e.g. swapping the two writes in thread 0) do not change the meaning of single-threaded programs
- the problematic transformations are detectable only by code that allows two threads to access the same data simultaneously in conflicting ways (e.g. one thread writes the data read by the other).

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y = 1 if *x == 1</td>
<td></td>
</tr>
<tr>
<td>*x = 1</td>
<td></td>
</tr>
</tbody>
</table>

Observable behaviour:

...intuition...

Programming languages provide synchronisation mechanisms

if these are used (and implemented) correctly, we might avoid the issues above...

*conflicting ways* (e.g. one thread writes the data read by the other).
The basic solution

Prohibit data races

Defined as follows:

- two memory operations conflict if they access the same memory location and at least one is a store operation;

- a SC execution (interleaving) contains a data race if two conflicting operations corresponding to different threads are adjacent (maybe executed concurrently).

Example: a data race in the example above:

\[
W_{t_1} y=1, W_{t_1} x=1, R_{t_2} x=1, R_{t_2} y=1, P_{t_2} 1
\]
The basic solution

Prohibit data races

Defined as follows:

The definition of data race quantifies only over the sequential consistent executions executed concurrently).

Example: a data race in the example above:

\[ W_{t_1} y = 1, W_{t_1} x = 1, R_{t_2} x = 1, R_{t_2} y = 1, P_{t_2} 1 \]
How do we avoid data races? (high-level languages)

• **Locks**
  No `lock(l)` can appear in the interleaving unless prior `lock(l)` and `unlock(l)` calls from other threads balance.

• **Atomic variables**
  Allow concurrent access “exempt” from data races (called `volatile` in Java).

**Example:**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y = 1</td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>lock();</td>
<td><code>tmp = *x;</code></td>
</tr>
<tr>
<td>*x = 1</td>
<td><code>unlock();</code></td>
</tr>
<tr>
<td>unlock();</td>
<td><code>if tmp = 1</code></td>
</tr>
<tr>
<td></td>
<td>then print *y</td>
</tr>
</tbody>
</table>
How do we avoid data races? (high-level languages)

This program is data-race free:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y = 1</td>
<td>lock();</td>
</tr>
<tr>
<td>lock();</td>
<td>tmp = *x;</td>
</tr>
<tr>
<td>*x = 1</td>
<td>unlock();</td>
</tr>
<tr>
<td>unlock();</td>
<td>if tmp = 1 then print *y</td>
</tr>
</tbody>
</table>

*y = 1; lock();*x = 1;unlock();

*y = 1; lock(); tmp = *x; unlock();

*y = 1; lock(); tmp = *x; unlock(); if tmp=1;

lock();tmp = *x;unlock();

*y = 1; lock(); *x = 1; unlock(); if tmp=1;

lock();tmp = *x;unlock();

*y = 1; lock(); *x = 1; unlock();

*y = 1; lock(); tmp = *x; unlock(); if tmp=1;

lock();tmp = *x;unlock();

*y = 1; lock();tmp = *x;unlock(); if tmp=1; *x = 1;unlock();

lock();tmp = *x;unlock();

*y = 1; lock();*x = 1;unlock();

lock();tmp = *x;unlock();

*y = 1; if tmp=1; lock(); *x = 1; unlock();

lock();tmp = *x;unlock();

*y = 1; if tmp=1; lock();*x = 1;unlock();
**How do we avoid data races?** (high-level languages)

- `lock()`, `unlock()` are opaque for the compiler: viewed as potentially modifying any location, memory operations cannot be moved past past them.

- `lock()`, `unlock()` contain "sufficient fences" to prevent hardware reordering across them and global ordering.

```c
*y = 1; lock();*x = 1;unlock();
```
```c
lock();tmp = *x;unlock(); if tmp=1 then print *y
```
```c
*y = 1; lock(); tmp = *x; unlock();
```
```c
lock(); *x = 1; unlock(); if tmp=1
```
```c
*y = 1; lock(); tmp = *x; unlock(); if tmp=1;
```
```c
lock(); *x = 1; unlock();
```
```c
lock();tmp = *x;unlock();
```
```c
*y = 1; lock(); *x = 1; unlock(); if tmp=1
```
```c
lock(); tmp = *x; unlock(); if tmp=1;
```
```c
*y = 1; lock();*x = 1;unlock();
```
```c
lock();tmp = *x;unlock();
```
```c
*y = 1; if tmp=1; lock(); *x = 1; unlock();
```
```c
lock();tmp = *x;unlock();
```
```c
*y = 1; if tmp=1; lock(); *x = 1; unlock();
```
Compiler/hardware can continue to reorder accesses

**Intuition:**

compiler/hardware do not know about threads but only racing threads can tell the difference!

- `lock()`, `unlock()` contain "sufficient fences" to prevent hardware reordering across them and global orderering.

*y = 1; lock();*x = 1;unlock();*y = 1; lock();*x = 1;unlock(); if tmp=1 then print *y
*y = 1; lock(); tmp = *x; unlock(); if tmp=1
*y = 1; lock(); tmp = *x; unlock(); if tmp=1; lock(); *x = 1; unlock();
lock();tmp = *x;unlock(); *y = 1; lock();*x = 1;unlock(); if tmp=1
lock(); tmp = *x; unlock(); if tmp=1; *y = 1; lock();*x = 1;unlock();
lock();tmp = *x;unlock(); *y = 1; if tmp=1; lock(); *x = 1; unlock();
Validity of compiler optimisations,

<table>
<thead>
<tr>
<th>Transformation</th>
<th>SC</th>
<th>DRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory trace preserving transformations</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant read after read elimination</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant read after write elimination</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Irrelevant read elimination</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant write before write elimination</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redundant write after read elimination</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Irrelevant read introduction</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Normal memory accesses reordering</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Roach-motel reordering</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>External action reordering</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

* Optimisations legal only on adjacent statements.
Validity of compiler optimisations,

<table>
<thead>
<tr>
<th>Transformation</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory trace preserving transformations</td>
<td>✓</td>
</tr>
</tbody>
</table>

Jaroslav Sevcik

Safe Optimisations for Shared-Memory Concurrent Programs

* Optimisations legal only on adjacent statements.
Compilers, programmers & data-race
Compilers, programmers & data-race

Can be implemented efficiently
Compilers, programmers & data-race

Can be implemented efficiently

Intuitive programming model (but detecting races is tricky!)
Another example of DRF program

Exercise: is this program DRF?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>if *x == 1</td>
<td>if *y == 1</td>
</tr>
<tr>
<td>then *y = 1</td>
<td>then *x = 1</td>
</tr>
</tbody>
</table>
Another example of DRF program

*Exercise:* is this program DRF?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if *x == 1</code></td>
<td><code>if *y == 1</code></td>
</tr>
<tr>
<td><code>then *y = 1</code></td>
<td><code>then *x = 1</code></td>
</tr>
</tbody>
</table>

*Answer:* yes!

The writes cannot be executed in any SC execution, so they cannot participate in a data race.
Another example of DRF program

Exercise: is this program DRF?

Data-race freedom is not the ultimate panacea

- the absence of data-races is hard to verify / test (undecidable)
- imagine debugging...

  my program ended with a wrong result:
  my program has a bug OR it has a data-race

  my program ended with a correct result:
  my program is correct OR it has a data-race
Defining programming language memory models
Option 1

Don't.

No concurrency.

Implemented by highly-successful programming languages (OCaml)

Poor match for current trends
Option 2

Don't.

No shared memory

A good match for some problems (see Erlang, MPI, …)
Option 3

Don't.

But language ensures data-race freedom

Possible:
- syntactically ensuring data accesses protected by associated locks
- fancy effect type systems (**don’t miss Pottier’s lecture on Friday**)

*Not suitable for general purpose programming.*
Option 4

Don't.

Leave it (sort of) up to the hardware

Example:

**MLton**, a high performance ML-to-x86 compiler with concurrency extensions

Accesses to ML refs exhibit the underlying x86-TSO behaviour (atomicity is guaranteed though)
Option 5

Do.

Use data race freedom as a definition

1. Programs that race-free have only sequentially consistent behaviours
2. Programs that have a race in some execution can behave in any way

*Sarita Adve & Mark Hill, 1990*
Option 5

Do.

Use data race freedom as a definition

Pro:
- simple
- strong guarantees for most code
- allows lots of freedom for compiler and hardware optimisations

Cons:
- undecidable premise
- can't write racy programs (escape mechanisms?)
Data-races are errors

[ANSI-STD-1815A-1983, 9.11] For the actions performed by a program that uses shared variables, the following assumptions can always be made:

- If between two synchronization points in a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.
- If between two synchronization points in a task, this task updates a shared variable whose task type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.

The execution of the program is erroneous if any of these assumptions is violated.
Data-races are errors

[IEEE 1003.1-2008, Base Definitions 4.11] 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.

Posix Threads Specification
Data-races are errors
Les data-races sont des erreurs

[C++ 2011 FDIS (WG21/N3290) 1.10p21] The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

How to use C/C++ to implement low-level system code?

Data-races are errors
Escape lanes
for expert programmers
Low-level atomics in C11/C++11

std::atomic<int> flag0(0), flag1(0), turn(0);

void lock(unsigned index) {
    if (0 == index) {
        flag0.store(1, std::memory_order_relaxed);
        turn.exchange(1, std::memory_order_acq_rel);
        while (flag1.load(std::memory_order_acquire) && 1 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    } else {
        flag1.store(1, std::memory_order_relaxed);
        turn.exchange(0, std::memory_order_acq_rel);
        while (flag0.load(std::memory_order_acquire) && 0 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    }
}

void unlock(unsigned index) {
    if (0 == index) {
        flag0.store(0, std::memory_order_release);
    } else {
        flag1.store(0, std::memory_order_release);
    }
}
The qualifiers

MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED

LESS RELAXED

MORE RELAXED
The qualifiers

- **MO_SEQ_CST**
- **MO_RELEASE / MO_ACQUIRE**
- **MO_RELEASE / MO_CONSUME**
- **MO_RELAXED**

Sequential consistent accesses

LESS RELAXED

MORE RELAXED
The qualifiers

MO_SEQ_CST

Sequential consistent accesses

MO_RELEASE / MO_CONSUME

Efficient implementation of message passing

MO_RELEASE / MO_ACQUIRE

LESS RELAXED

MO_RELAXED

MORE RELAXED
The qualifiers

- **MO_SEQ_CST**
  - Sequential consistent accesses

- **MO_RELEASE**
  - Efficient implementation of message passing

- **MO_RELAXED**
  - Efficient implementation of message passing on ARM/Power

**LESS RELAXED**

**MORE RELAXED**
The qualifiers

MO_SEQ_CST
Sequential consistent accesses

MO_RELEASE
Efficient implementation of message passing

MO_RELEASE / MO_ACQUIRE
Efficient implementation of message passing on ARM/Power

MO_RELEASE / MO_CONSUME
No synchronisation; direct access to hardware

LESS RELAXED

MORE RELAXED
The compiler must ensure that `MO_SEQ_CST` accesses have sequentially consistent semantics.

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.store(1,MO_SEQ_CST)</code></td>
<td><code>y.store(1,MO_SEQ_CST)</code></td>
</tr>
<tr>
<td><code>r1 = y.load(MO_SEQ_CST)</code></td>
<td><code>r2 = x.load(MO_SEQ_CST)</code></td>
</tr>
</tbody>
</table>

The program above cannot end with `r1 = r2 = 0`.

Sample compilation on x86:

- **store:** `MOV; MFENCE`
- **load:** `MOV`

Sample compilation on Power:

- **store:** `HWSYNC; ST`
- **load:** `HWSYNC; LD; CMP; BC; ISYNC`
MO_RELAXED

MO_RELAXED accesses can be reordered by compiler/hardware

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.store(1, MO_RELAXED)</code></td>
<td><code>y.store(1, MO_RELAXED)</code></td>
</tr>
<tr>
<td><code>r1 = y.load(MO_RELAXED)</code></td>
<td><code>r2 = x.load(MO_RELAXED)</code></td>
</tr>
</tbody>
</table>

The program above can end with `r1 = r2 = 0`.

Sample compilation on x86:

- `store: MOV`
- `load: MOV`

Sample compilation on Power:

- `store: ST`
- `load: LD`
MO_RELEASE / MO_ACQUIRE

Supports a fast implementation of the message passing idiom:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.store(1,MO_RELAXED)</td>
<td>r1 = y.load(MO_ACQUIRE)</td>
</tr>
<tr>
<td>y.store(1,MO_RELEASE)</td>
<td>r2 = x.load(MO_RELAXED)</td>
</tr>
</tbody>
</table>

The program above cannot end with \( r1 = 1 \) and \( r2 = 0 \).

Accesses to the data structure can be reordered/optimised (MO_RELAXED).

Sample compilation on x86:  
store: MOV  
load: MOV

Sample compilation on Power:  
store: LWSYNC; ST  
load: LD; CMP; BC; ISYNC
MO_RELEASE / MO_CONSUME

Supports a fast implementation of the message passing idiom on Power:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x.store(1,MO_RELAXED)}</td>
<td>\texttt{r1 = y.load(x,MO_CONSUME)}</td>
</tr>
<tr>
<td>\texttt{y.store(&amp;x,MO_RELEASE)}</td>
<td>\texttt{r2 = (*r1).load(MO_RELAXED)}</td>
</tr>
</tbody>
</table>

The program above cannot end with \( r1 = 1 \) and \( r2 = 0 \).

The two loads have an address dependency, Power won't reorder them.

Sample compilation on x86:  
store: \texttt{MOV}  
load: \texttt{MOV}

Sample compilation on Power:  
store: \texttt{LWSYNC; ST}  
load: \texttt{LD}
The C11/C++11 memory model formalisation
Enough about formalising...

...what about reasoning?
Topics

1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compilers and optimisations: proof and/or validation
Topics

1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compilers and optimisations: proof and/or validation
Idea: the programming language *faithfully* mimics the processor model.

The C-TSO programming language: a C-like language with a TSO semantics for memory accesses.

A semantic preserving compiler *CompCertTSO* building on CompCert 1.5

Intel processors implement the x86-TSO MM
CompCert 1.5 proves that all behaviours of the source program are behaviours of the compiled program (building simulation relations).

The converse follows from determinacy of the semantics.

Problem: in CompCertTSO the semantics is not deterministic...
**Semantic engineering**

*Proof sketch*

*Want:* whole-system *upward* simulation

*Have:* Leroy's per-thread *downward* simulations

1. replace implicit memory accesses with explicit labels
2. port Leroy's proof to the labellised semantics
   - surprisingly easy for many phases
   - tedious for explicitly small-stepped phases (could not reuse CompCert's proof)
3. Turn per-thread downward simulations to per-thread upward simulations
4. Turn per-thread upward simulations to whole-system upward simulations
5. Compose the whole system upward simulations.

*If* $R$ is a threadwise downward simulation from $S$ to $T$, $S$ is receptive, and $T$ is determinate, *then there is a threadwise upward simulation that contains* $R$. 

Wednesday 5 August 15
Semantic engineering

Want: whole-system upward simulation
Have: Leroy’s per-thread downward simulations

1. replace implicit memory accesses with explicit labels
2. port Leroy’s proof to the labelled semantics
   • surprisingly easy for many phases
   • tedious for explicitly small-stepped phases (could not reuse CompCert’s proof)
3. Turn per-thread downward simulations to per-thread upward simulations
4. Turn per-thread upward simulations to whole-system upward simulations
5. Compose the whole system upward simulations.

ClightTSO small-step semantics has about 90 reduction rules

How to formalise programming language definitions?

If R is a threadwise downward simulation from S to T, S is receptive, and T is determinate, then there is a threadwise upward simulation that contains R.
The Ott tool

Complement to LEM, specialised for formalising programming language definitions and semantics.

\[
\begin{align*}
\text{type_to_chunk} & \quad ty_1 = \text{Some } c \\
\text{cast_value_to_chunk} & \quad c \ v_1 = v_2
\end{align*}
\]

\[
\frac{v_1 \cdot [p_1^\{ty_1\} = \_].\ k \ | \ \text{env} \quad \text{--mem}(\text{write } p_1 \ c \ v_2) \rightarrow \text{skip}. \ k \ | \ \text{env}}{	ext{StepAssign}}
\]

| StepAssign : \forall (v_1: \text{val}) (p_1: \text{pointer}) (ty_1: \text{type}) (k: \text{cont}) (env: \text{env}) \\
(c: \text{memory_chunk}) (v_2: \text{val}), \\
\text{type_to_chunk} \quad ty_1 = \text{Some } c \rightarrow \\
\text{cast_value_to_chunk} \quad c \ v_1 = v_2 \rightarrow \\
\text{cl_step} \quad (SKval \ v_1 \ | \ \text{env} \ (EKassign2 \ (Vptr \ p_1) \ ty_1 \ k) ) \ (TEmem \ (MWrite \ p_1 \ c \ | \ \text{env} \ k) \skip \ | \ \text{env} \ k)
The Ott tool

Complement to LEM, specialised for formalising programming language definitions and semantics.

ClightTSO is formalised in Ott
we get an interpreter as a biproduct

\[
\begin{align*}
\text{type_to_chunk} & \quad ty_1 = \text{Some } c \\
\text{cast_value_to_chunk} & \quad c \ v_1 = v_2
\end{align*}
\]
CompCertTSO
CompCertTSO + fence optimisations

- ClightTSO
  - simplify
  - local vars
  - instruction selection
  - CFG generation

- C#minor
  - Cstacked
    - simplify
    - local vars

- Cminor
  - CminorSel
    - simplify
    - instruction selection
    - CFG generation

- Const prop.
  - CSE
    - FE1
    - PRE
    - FE2

- Branch tunnelling
  - linearize
    - reload/spill
    - act.records
    - Machabstr
      - Machconc
      - x86

Register allocation

SAS 2012

Wednesday 5 August 15
Example of fence elimination in action

PRE

FE2

Wednesday 5 August 15
Example of fence elimination in action

Proof of correctness requires a novel bisimulation-based proof technique (need to guess if “in the future” a fence instruction will be executed).
What about C11?
Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);

Can you guess the output?
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>26; ++b)
    ;
}

Thread 2

b = 42;
printf("%d\n", b);
### Thread 1

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

### Thread 2

```c
b = 42;
printf("%d\n", b);
```
int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 1

Thread 2

b = 42;
printf("%d\n", b);
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}

Thread 2

int a = 1;
int b = 0;

b = 42;
printf("%d\n", b);
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}

Thread 1

Thread 2

int a = 1;
int b = 0;
b = 42;
printf("%d\n", b);

Thread 1 returns without modifying b
Shared memory

int a = 1;
int b = 0;

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 1 returns without modifying b

Thread 2

b = 42;
printf("%d\n", b);

Thread 2 is not affected by Thread 1 and vice-versa
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
;
}

b = 42;
printf("%d\n", b);

Thread 1

Thread 2

Thread 1 returns without modifying b

Thread 2 is not affected by Thread 1 and vice-versa

C11 states that this program must print 42
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 1

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

Thread 2

b = 42;
printf("%d\n", b);
int a = 1;
int b = 0;

**Thread 1**

```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

**Thread 2**

```c
b = 42;
printf("%d\n", b);
```
int a = 1;  
int b = 0;  

Thread 1  
int s;  
for (s=0; s!=4; s++) {  
  if (a==1)  
    return NULL;  
  for (b=0; b>=26; ++b)  
    ;  
}  

Thread 2  
b = 42;  
printf("%d\n", b);  
gcc 4.7 -O2  

...sometimes we get 0 on the screen
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    {
    }
}

movl a(%rip), %eax    # load a into eax
movl b(%rip), %ebx    # load b into ebx
testl %eax, %eax      # if a==1
jne .L2               # jump to .L2
movl $0, b(%rip)       # store 0 into eax
ret

.L2:
    movl %ebx, b(%rip)   # store ebx into b
    xorl %eax, %eax     # store 0 into eax
    ret                 # return
```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

The outer loop can be (and is) optimised away

```asm
movl a(%rip), %eax  # load a into eax
movl b(%rip), %ebx  # load b into ebx
testl %eax, %eax    # if a==1
jne  .L2            # jump to .L2
movl $0, b(%rip)    # store 0 into eax
ret
.L2:
movl %ebx, b(%rip)   # store ebx into b
xorl %eax, %eax     # store 0 into eax
ret                  # return
```

gcc 4.7 -O2

The outer loop can be (and is) optimised away
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

movl  a(%rip), %eax  # load a into eax
movl  b(%rip), %ebx  # load b into ebx
testl %eax, %eax     # if a==1
jne   .L2            #   jump to .L2
movl  $0, b(%rip)    # store 0 into eax
ret                   # return

.L2:
    movl  %ebx, b(%rip)  # store ebx into b
    xorl  %eax, %eax    # store 0 into eax
    ret                  # return

gcc 4.7 -O2
```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
    ;
}
```

```assembly
movl a(%rip), %eax    # load a into eax
movl b(%rip), %ebx    # load b into ebx
testl %eax, %eax      # if a==1
jne   .L2             # jump to .L2
movl $0, b(%rip)      # store 0 into ebx
ret                   # return
.L2:
movl %ebx, b(%rip)    # store ebx into b
xorl %eax, %eax       # store 0 into eax
ret                   # return
```

```bash
gcc 4.7 -O2
```
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
       ;
}

movl a(%rip), %eax    # load a into eax
movl b(%rip), %ebx    # load b into ebx
testl %eax, %eax      # if a==1
jne .L2              # jump to .L2
movl $0, b(%rip)      # load 0 into b
ret                   # return

.L2:
movl %ebx, b(%rip)    # store ebx into b
xorl %eax, %eax      # store 0 into eax
ret                   # return
```c
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

```
gcc 4.7 -O2
```

```
movl a(%rip), %eax  # load a into eax
movl b(%rip), %ebx  # load b into ebx
testl %eax, %eax    # if a==1
jne .L2             #   jump to .L2
movl $0, b(%rip)    # store ebx into b
ret                   # return
```

```
.L2:
movl %ebx, b(%rip)  # store ebx into b
xorl %eax, %eax    # store 0 into eax
ret                   # return
```
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

movl  a(%rip), %eax   # load a into eax
movl  b(%rip), %ebx   # load b into ebx
testl %eax, %eax      # if a==1
jne   .L2             #   jump to .L2
movl  $0, b(%rip)     # store 0 into b
ret                   # return

.L2:
movl  %ebx, b(%rip)    # store ebx into b
xorl  %eax, %eax      # store 0 into eax
ret                   # return
The compiled code saves and restores b

Correct result in a sequential setting

```
movl a(%rip), %eax  # load a into eax
movl b(%rip), %ebx  # load b into ebx
testl %eax, %eax    # if a==1
jne  .L2            # jump to .L2
movl $0, b(%rip)    # store 0 into eax
ret
.L2:
movl %ebx, b(%rip)  # store ebx into b
xorl %eax, %eax     # store 0 into eax
ret                 # return
```
**Shared memory**

```c
int a = 1;
int b = 0;
```

**Thread 1**

```asm
movl   a(%rip),%eax
movl   b(%rip),%ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret
.L2:
    movl   %ebx, b(%rip)
xorl   %eax, %eax
ret
```

**Thread 2**

`b = 42;
printf("%d\n", b);`
Thread 1

```
        movl   a(%rip),%eax
        movl   b(%rip),%ebx
        testl  %eax, %eax
        jne    .L2
        movl   $0, b(%rip)
        ret
.L2:
        movl   %ebx, b(%rip)
        xorl   %eax, %eax
        ret
```

Thread 2

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
Shared memory

```c
int a = 1;
int b = 0;
```

### Thread 1

```assembly
movl   a(%rip),%eax
movl   b(%rip),%ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret
```

### Thread 2

```c
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
- Read b (0) into ebx
Shared memory

```c
int a = 1;
int b = 0;
```

Thread 1

```assembly
movl a(%rip),%eax
movl b(%rip),%ebx
testl %eax, %eax
jne .L2
movl $0, b(%rip)
ret
```

.L2:

```assembly
movl %ebx, b(%rip)
xorl %eax, %eax
ret
```

Thread 2

```assembly
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
- Read b (0) into ebx
- Store 42 into b
```
int a = 1;
int b = 0;
```

**Thread 1**

```
movl   a(%rip),%eax
movl   b(%rip),%ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret
.L2:
```

**Thread 2**

```
b = 42;
printf("%d\n", b);
```

- Read a (1) into eax
- Read b (0) into ebx
- Store 42 into b
- Store ebx (0) into b
Shared memory

```
int a = 1;
int b = 0;
```

Thread 1

```
movl   a(%rip), %eax
movl   b(%rip), %ebx
testl  %eax, %eax
jne    .L2
movl   $0, b(%rip)
ret
.L2:
    movl   %ebx, b(%rip)
xorl   %eax, %eax
ret
```

Thread 2

```
b = 42;
printf("%d
", b);
```

- Read a (1) into eax
- Read b (0) into ebx
- Store 42 into b
- Store ebx (0) into b
- Print b: 0 is printed
The horror, the horror... a subtle compiler bug!
Compiler testing: state of the art
Yang, Chen, Eide, Regehr - PLDI 2011

![Diagram showing the flow of compiler testing process]

- Random Generator
  - C program
  - clang -O0
  - clang -O3
  - gcc -O
  - ... 

- Results
  - Majority
  - Vote
  - Minority
Compiler testing: state of the art
Yang, Chen, Eide, Regehr - PLDI 2011

Reported hundreds of bugs on various versions of gcc, clang and other compilers
Reported hundreds of bugs on various versions of gcc, clang and other compilers.

Cannot catch concurrency compiler bugs.
Hunting concurrency compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours: how to test for correctness?

limit case: two compilers generate correct code with disjoint final states
Idea

C/C++ compilers support separate compilation
Functions can be called in arbitrary non-racy concurrent contexts

\[ \downarrow \]

C/C++ compilers can only apply transformations sound with respect to an arbitrary non-racy concurrent context

Hunt concurrency compiler bugs

= search for transformations of sequential code not sound in an arbitrary non-racy context
Random Generator  →  SEQUENTIAL PROGRAM

reference semantics

REFERENCE MEMORY TRACE

optimising compiler under test

EXECUTABLE

MEMORY TRACE

tracing

Check: only transformations sound in any concurrent non-racy context
Soundness of compiler optimisations in the C11/C++11 memory model
Elimination of overwritten writes

Under which conditions is it correct to eliminate the first store?
A **same-thread release-acquire pair** is a pair of a release action followed by an acquire action in program order.

An action is a *release* if it is a possible source of a synchronisation

\[ \text{unlock mutex, release or seq_cst atomic write} \]

An action is an *acquire* if it is a possible target of a synchronisation

\[ \text{lock mutex, acquire or seq_cst atomic read} \]
Elimination of overwritten writes

It is safe to eliminate the first store if there are:

1. no intervening accesses to $g$
2. no intervening same-thread release-acquire pair
The soundness condition

Shared memory

g = 0; atomic f1 = f2 = 0;

Thread 1

g = 1;
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;
The soundness condition

Shared memory

g = 0; atomic f1 = f2 = 0;

Thread 1

candidate overwritten write

\[\begin{align*}
g &= 1; \\
f1.\text{store}(1, \text{RELEASE}); \\
\text{while}(f2.\text{load}(\text{ACQUIRE}) == 0); \\
g &= 2;
\end{align*}\]
The soundness condition

Shared memory

\[ g = 0; \text{ atomic } f1 = f2 = 0; \]

**Thread 1**

- candidate overwritten write
- same-thread release-acquire pair

\[
\begin{align*}
g &= 1; \\
f1.\text{store}(1,\text{RELEASE}); \\
\text{while}(f2.\text{load}(\text{ACQUIRE}) == 0); \\
g &= 2;
\end{align*}
\]
The soundness condition

Shared memory

\[
g = 0; \text{ atomic } f_1 = f_2 = 0;
\]

**Thread 1**

\[
g = 1; \\
f_1.\text{store}(1, \text{RELEASE}); \\
\text{while}(f_2.\text{load}(\text{ACQUIRE}) == 0); \\
g = 2;
\]

**Thread 2**

\[
\text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \\
\text{printf("%d", g); \\
f_2.\text{store}(1, \text{RELEASE});
\]

Wednesday 5 August 15
The soundness condition

Shared memory

g = 0; atomic \ f1 = f2 = 0;

Thread 1

\begin{align*}
g &= 1; \\
f1 \text{.store}(1, \text{RELEASE}); \\
\text{while}(f2 \text{.load}(\text{ACQUIRE}) == 0); \\
g &= 2;
\end{align*}

Thread 2

\begin{align*}
\text{while}(f1 \text{.load}(\text{ACQUIRE}) == 0); \\
\text{printf}(\text{"%d"}, g); \\
f2 \text{.store}(1, \text{RELEASE});
\end{align*}

Thread 2 is non-racy
The soundness condition

\[ g = 0; \text{atomic } f_1 = f_2 = 0; \]

**Shared memory**

\[ g = 1; \]
\[ f_1.\text{store}(1, \text{RELEASE}); \]
\[ \text{while}(f_2.\text{load}(\text{ACQUIRE})==0); \]
\[ g = 2; \]

**Thread 1**

**Thread 2**

\[ \text{while}(f_1.\text{load}(\text{ACQUIRE})==0); \]
\[ \text{printf}("%d", g); \]
\[ f_2.\text{store}(1, \text{RELEASE}); \]

Thread 2 is non-racy

The program should only print 1
The soundness condition

Shared memory

g = 0; atomic f1 = f2 = 0;

Thread 1

\[
g = 1;
\]
f1.store(1,RELEASE);
while(f2.load(ACQUIRE)==0);
g = 2;

Thread 2

\[
\text{while}(f1.load(ACQUIRE)==0);
\]
\[
\text{printf} \("%d", \ g);
\]
f2.store(1,RELEASE);

Thread 2 is non-racy
The program should only print 1
If we perform overwritten write elimination it prints 0
The soundness condition

Shared memory

\[ g = 0; \text{atomic } f_1 = f_2 = 0; \]

**Thread 1**

\[ g = 1; \]
\[ f_1.\text{store}(1, \text{RELEASE}); \]
\[ \text{while}(f_2.\text{load}(\text{ACQUIRE}) == 0); \]
\[ g = 2; \]

**Thread 2**

\[ \text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \]
\[ \text{printf}("%d", g); \]
\[ f_2.\text{store}(1, \text{RELEASE}); \]
The soundness condition

**Shared memory**

\[ g = 0; \text{atomic } f1 = f2 = 0; \]

**Thread 1**

\[ g = 1; \]
\[ f1.\text{store}(1,\text{RELEASE}); \]
\[ g = 2; \]

**Thread 2**

\[ \text{while}(f1.\text{load}(\text{ACQUIRE})==0); \]
\[ \text{printf}("%d", g); \]
\[ f2.\text{store}(1,\text{RELEASE}); \]
The soundness condition

**Shared memory**

\[ g = 0; \text{atomic } f_1 = f_2 = 0; \]

**Thread 1**

\[ g = 1; \]
\[ f_1.\text{store}(1, \text{RELEASE}); \]
\[ g = 2; \]

**Thread 2**

\[ \text{while}(f_1.\text{load}(\text{ACQUIRE}) == 0); \]
\[ \text{printf}("%d", g); \]
\[ f_2.\text{store}(1, \text{RELEASE}); \]

If only a release (or acquire) is present, then all discriminating contexts are racy.

It is sound to optimise the overwritten write.
Eliminations: bestiary

Overwritten-Write Write-after-Write
Read-after-Read Read-after-Write
Write-after-Read

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (irrelevant reads).
Also correctness statements for reorderings, merging, and introductions of events.

Overwritten-Write  Write-after-Write  Read-after-Read  Read-after-Write  Write-after-Read

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (irrelevant reads).
From theory to the Cmmtest tool
Check: only transformations sound in any concurrent non-racy context
CSmith extended with locks and atomics

REFERENCE

reference semantics

SEQUENTIAL PROGRAM

optimising compiler under test

EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context
CSmith extended with locks and atomics

SEQUENTIAL PROGRAM

optimising compiler under test

EXECUTABLE

binary instrumentation

REFERENCE MEMORY TRACE

REFERENCE semantics

EXECUTABLE MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context
CSmith extended with locks and atomics

gcc/clang -O0

EXECUTABLE

binary instrumentation

REFERENCE
MEMORY
TRACE

Check: only transformations sound in any concurrent non-racy context
CSmith extended with locks and atomics → SEQUENTIAL PROGRAM → optimising compiler under test

gcc/clang -O0

EXECUTABLE

binary instrumentation

REFERENCE MEMORY TRACE

1. analyse the traces to detect eliminable actions
2. match reference and optimised traces

EXECUTABLE

binary instrumentation

MEMORY TRACE

OCaml tool
const unsigned int g3 = 0UL;
long long g4 = 0x1;
int g6 = 6L;
volatile unsigned int g5 = 1UL;

void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Start with a randomly generated well-defined program
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

const unsigned int g3 = 0UL;
long long g4 = 0x1;
int g6 = 6L;
volatile unsigned int g5 = 1UL;
unsigned int l107 = 0xAA37C3ACL;
g4 &= g3;
g5++;
int *l102 = &l36;
for (g6 = 4; g6 < (-3); g6 += 1);
l102 = &g6;
*l102 = ((*l8) && (l107 << 7)*(*l102));
void func_1(void) {
    int *l8 = &g6;
    int l36 = 0x5E9D070F;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Init g3 0
Init g4 1
Init g5 1
Init g6 6
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)>(*l102));
}
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```c
void func_1(void)
{
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xA9A37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1)
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```
```c
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}

Can match applying
only correct eliminations and reorderings

reference semantics

gcc -O2 memory trace
int a = 1;
int b = 0;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

If we focus on the miscompiled initial example...
int a = 1;
int b = 0;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
int a = 1;
int b = 0;

int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}

reference semantics

Load a 1
```c
int a = 1;
int b = 0;
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```
int s;
for (s=0; s!=4; s++) {
if (a==1)
return NULL;
for (b=0; b>=26; ++b);
}

int a = 1;
int b = 0;

Load a 1
Load b 0
Store b 0

Cannot match some events → detect compiler bug

reference semantics

gcc -O2 memory trace

Load a 1

Load a 1

Applications
1. Testing C compilers (GCC, Clang, ICC)

Some concurrency compiler bugs found in the latest version of GCC.

Store introductions performed by loop invariant motion or if-conversion optimisations.

*Remark*: these bugs break the Posix thread model too.

All promptly fixed.
2. Checking compiler invariants

GCC internal invariant: never reorder with an atomic access

Baked this invariant into the tool and found a counterexample...

...not a bug, but fixed anyway

```c
atomic_uint a;
int32_t g1, g2;
int main (int, char *[]) {
    a.load() & a.load ();
    g2 = g1 != 0;
}
```

```
ALoad  a   0  4
ALoad  a   0  4
Load   g1  0  4
Store  g2  0  4
Load   g1  0  4
ALoad  a   0  4
ALoad  a   0  4
Store  g2  0  4
```
3. Detecting unexpected behaviours

```
uint16_t g
for (; g==0; g--);
```

Correct or not?

```
uint16_t g

g=0;
```
3. Detecting unexpected behaviours

```c
uint16_t g
for (; g==0; g--);
g=0;
```

If `g` is initialised with `0`, a load gets replaced by a store:

```c
Load g 0
```

The introduced store cannot be observed by a non-racy context. Still, **arguable if a compiler should do this or not.**
3. Detecting unexpected behaviours

```c
uint16_t g
for (; g==0; g--);
g=0;
```

If `g` is initialised with 0, a load gets replaced by a store:

```
Load g 0 Store g 0
```

*False positives in Thread Sanitizer*
The formalisation of the C11 memory model enables compiler testing... what else?
Proving the correctness of mappings for atomics

https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

<table>
<thead>
<tr>
<th>C/C++11 Operation</th>
<th>ARM implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Relaxed:</td>
<td>ldr</td>
</tr>
</tbody>
</table>
| Load Consume:                     | ldr + preserve dependencies until next kill_dependency  
                                         | OR                  
                                         | ldr; teq; beq; isb  
                                         | OR                  
                                         | ldr; dmb            |
| Load Acquire:                     | ldr; teq; beq; isb  
                                         | OR                  
                                         | ldr; dmb            |
| Load Seq Cst:                     | ldr; dmb            |
| Store Relaxed:                    | str                 |
| Store Release:                    | dmb; str            |
| Store Seq Cst:                    | dmb; str; dmb       |
| Cmpxchg Relaxed (32 bit):         | _loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop |
| Cmpxchg Acquire (32 bit):         | _loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb |
| Cmpxchg Release (32 bit):        | dmb; _loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb |
| Cmpxchg AcqRel (32 bit):          | dmb; _loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb |
| Cmpxchg SeqCst (32 bit):          | dmb; _loop: ldrex roldval, [rptr]; mov res, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; dmb |
| Acquire Fence:                    | dmb                 |
| Release Fence:                    | dmb                 |
| AcqRel Fence:                     | dmb                 |
| SeqCst Fence:                     | dmb                 |
Inform new optimisations

e.g. the work by Robin Morisset on the Arm LLVM backend

while (flag.load(acquire))
{}

.loop
  ldr r0, [r1]
  dmb ish
  bnz .loop

.loop
  ldr r0, [r1]
  bnz .loop
  dmb ish
Take-up in Industrial Concurrency Community?

handled the real behaviour - found some bugs - published some papers

• Fixed up ISO C/C++11 Standard
  standard text and our maths in sync

• Fixed and verified C/C++11 to POWER compilation scheme
  compilers have to agree on this

• Clarified POWER and ARM architectural intent
  ongoing dialogues with the architects

• Found concurrency bugs in gcc, proposing optimisation schemes
  ongoing dialogue with gcc developers
The memory models of modern hardware are better understood.

Programming languages attempt to specify and implement reasonable memory models.

Researchers and programmers are now interested in these problems.
The memory models of modern hardware are better understood. Still, many open problems...
The memory models of modern hardware are better understood

Still, many research opportunities!
Thank you! Questions?