Reasoning with Weak Memory
An Introduction

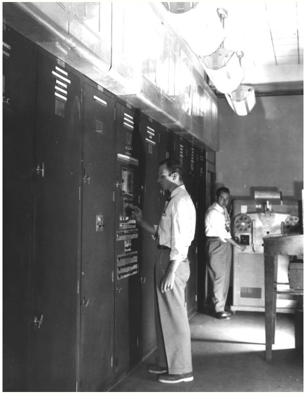
Susmit Sarkar

University of St Andrews

SPLV: July 2022

Memory: a basic abstraction

Ever since von Neumann/Turing (arguably even Babbage):


EDVAC – picture credit Wikipedia

$$\begin{tikzcd}
\boxed{\text{Processor}} \arrow[d, shift right, "\text{Store}"'] \\
\boxed{\text{Memory}} \arrow[u, shift right, "\text{Load}"']
\end{tikzcd}$$

Another old idea: Multiprocessors

Parallel hardware/concurrent programs

Another old idea: Multiprocessors

Parallel hardware/concurrent programs

BURROUGHS D825, 1962

Picture credit: Burroughs Corporation

Outstanding features include truly modular hardware with parallel processing throughout.

FUTURE PLANS

The complement of compiling languages is to be expanded.

Shared Memory

For variety of reasons, shared memory multiprocessors are now everywhere

Different threads communicate via shared memory

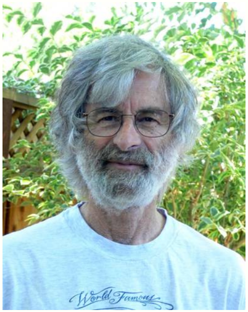
Aside: Message-passing hardware explored, but not mainstream

Key Question: What view does each thread have of shared memory?

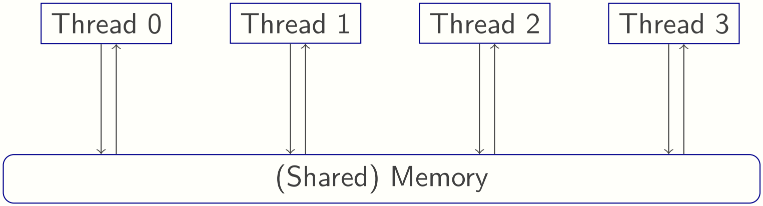
Sequential Consistency

...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program.

[Lamport, 1979]

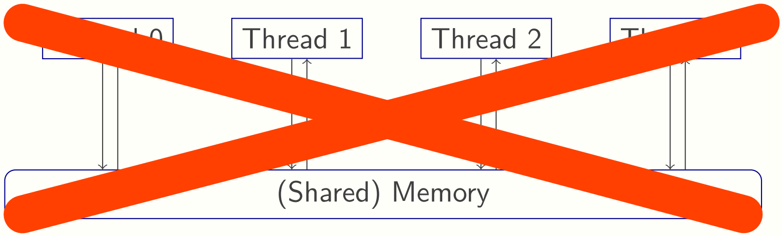


Sequential Consistency



- Traditional assumption (concurrent algorithms, semantics, verification): Sequential Consistency (SC)
- Implies: can use interleaving semantics
- Note: Optimisations allowed, as long as results “as if” linear order

Sequential Consistency



- Traditional assumption (concurrent algorithms, semantics, verification): Sequential Consistency (SC)
- Implies: can use interleaving semantics
- Note: Optimisations allowed, as long as results "as if" linear order

- **False** on modern (since 1972) multiprocessors, *or* with optimizing compilers

Our world is *not* SC

Not since IBM System 370/158MP (1972)

……Nor in x86, ARM, POWER, RISC-V, SPARC, or Itanium, …

……Nor in C, C++, Java, JavaScript, …

Example: Mutual Exclusion

At heart of mutual exclusion algorithm (Dekker's, Peterson's) there is usually code like:

Initially: t0wants = FALSE; t1wants = FALSE;	
Thread 0	Thread 1

`t0wants = TRUE;`
`if (NOT t1wants) {`
`  { …CRITICAL1 };`
	`t1wants = TRUE;`
	`if (NOT t0wants) {`
	`  { …CRITICAL2 };`

- Does it work?

Example: Mutual Exclusion Litmus Test

Distilling that example

Initially: $x = 0;\ \ y = 0;$	
Thread 0	Thread 1

`x := 1;`
`r`$_0$` := y;`
	`y := 1;`
	`r`$_1$` := x;`

Finally: $r_0 = 0 \wedge r_1 = 0$??	

- Forbidden on SC (no interleaving allows that result)

On actual hardware?

We use the `litmus7` tool (`diy.inria.fr`, Alglave and Maranget)
`SB.litmus`

```
X86 SB
"Fre PodWR Fre PodWR"
{ x=0; y=0; }
 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;
locations [x;y;]
exists (0:EAX=0 /\ 1:EAX=0)
```

Test Results

```
$ litmus7 SB.litmus
[...]
Histogram (4 states)
14
*>0:rax=0; 1:rax=0;
499983:>0:rax=1; 1:rax=0;
499949:>0:rax=0; 1:rax=1;
54
:>0:rax=1; 1:rax=1;
[...]
Observation SB Sometimes 14 999986
[...]
```

14 in 1e6 (Intel Core i7)

Test Results

```
$ litmus7 SB.litmus
[...]
Histogram (4 states)
7136481
:> 0:X2=0; 1:X2=0;
596513783:> 0:X2=0; 1:X2=1;
596513170:> 0:X2=1; 1:X2=0;
36566
:> 0:X2=1; 1:X2=1;
[...]
Observation SB Sometimes 7136481 1193063519
[...]
```

7e6 in 1.2e9 on Apple A10 (iPhone7)

What’s going on here?

Multiprocessors (and compilers) incorporate many

performance optimisations

(local store buffers, cache hierarchies, speculative execution, common subexpression elimination, hoisting code above loops, ...)

These are:

- unobservable by single-threaded code;
- sometimes observable by concurrent (multi-threaded) code

What's going on here?

Multiprocessors (and compilers) incorporate many

(local sto
common

These ar

- uno
- som

Upshot:

No longer a *sequential consistent* memory model

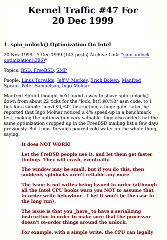
Instead, only a *weak (consistency)* (or *relaxed*) memory model

Weak Memory Consistency Models

- Real memory consistency models are **subtle**
- Real memory consistency models **differ between architectures**
- Real memory consistency models **differ between languages**
- Real memory consistency models **make SC concurrent reasoning unsound**

Research Opportunity!

Why Care? – A Motivating Tale



Kernel Traffic #47 For 20 Dec 1999

1. spin_unlock() Optimization On Intel

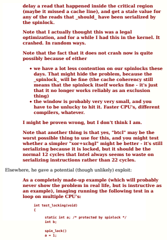
20 Nov 1999 - 7 Dec 1999 (143 posts) Archive Link: "spin_unlock optimization(i386)"

Topics: BSD: FreeBSD, SMP

People: Linus Torvalds, Jeff V. Merkey, Erich Boleyn, 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 does NOT WORK!**
>
> **Let the FreBSD 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**

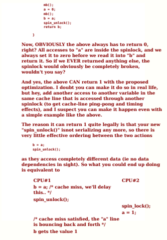


> **delay a read that happened inside the critical region (maybe it missed a cache line), and get a stale value for any of the reads that _should_ have been serialized by the spinlock.**
>
> **Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. 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, "btcl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+xchgl" 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:**

```
int test_locking(void)
{

        static int a; /* protected by spinlock */
        int b;

        spin_lock()
        a = 1;
```



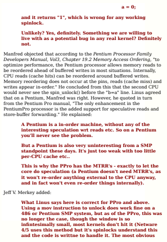
```
        mb();
        a = 0;
        mb();
        b = a;
        spin_unlock();
        return b;

}
```

> **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 "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()" isnot 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 what you could end up doing is equivalent to**

CPU#1	**CPU#2**
b = a; /* cache miss, we'll delay this.. */	
spin_unlock();	
	spin_lock();
	a = 1;
/* cache miss satisfied, the "a" line is bouncing back and forth */	
b gets the value 1	



	a = 0;

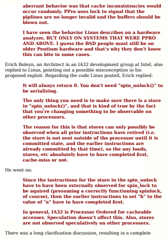
> **and it returns "1", which is wrong for any working spinlock.**
>
> **Unlikely? Yes, definitely. Something we are willing to live with as a potential bug in any real kernel? Definitely not.**

Manfred objected that according to the *Pentium Processor Family Developers Manual, Vol3, 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 "b=a" 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.**
>
> **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 is so infintesimally 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 piplines are no longer invalid and the buffers should be blown out.**
>
> **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.**

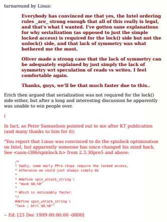
Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, Erich replied:

> **It will always return 0. You don't need "spin_unlock()" to be serializing.**
>
> **The only thing you need is to 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, 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 Linus:

> **Everybody has convinced me that yes, the Intel ordering rules _are_ strong enough that all of this really is legal, and that's what I wanted. I've gotten sane 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 strong 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..**

Erich then argued that serialization was not required for the lock() side either, but after a long and interesting discussion he apparently was unable to win people over.

(

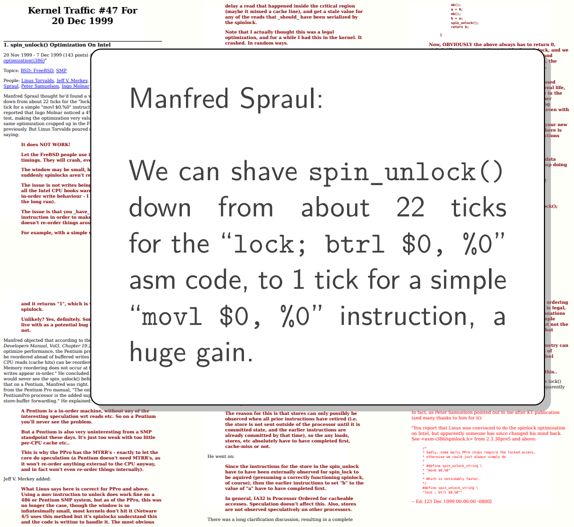
In fact, as Peter Samuelson pointed out to me after KT publication (and many thanks to him for it):

"You report that Linus was convinced to do the spinlock optimization on Intel, but apparently someone has since changed his mind back. See <asm-i386/spinlock.h> from 2.3.30pre5 and above:

```
/*
 * Sadly, some early PPro chips require the locked access,
 * otherwise we could just always simply do
 *
 * #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale



Kernel Traffic #47 For 20 Dec 1999

1. spin_unlock() Optimization On Intel

20 Nov 1999 - 7 Dec 1999 (143 posts)
optimization(i386)"

Topics: BSD: FreeBSD, SMP

People: Linus Torvalds, Jeff V. Merkey,
Spraul, Peter Samuelson, Ingo Molnar

Manfred Spraul thought he'd found a w
down from about 22 ticks for the "lock;
tick for a simple "movl $0,%0" instruct
reported that Ingo Molnar noticed a 4%
test, making the optimization very valu
same optimization cropped up in the F
previously. But Linus Torvalds poured
saying:

> **It does NOT WORK!**
>
> **Let the FreBSD people use i
> timings. They will crash, eve**
>
> **The window may be small, b
> suddenly spinlocks aren't re**
>
> **The issue is not writes being
> all the Intel CPU books warn
> in-order write behaviour - I
> the long run).**
>
> **The issue is that you _have_
> instruction in order to make
> doesn't re-order things arou**
>
> **For example, with a simple v**

> **and it returns "1", which is
> spinlock.**
>
> **Unlikely? Yes, definitely. Som
> live with as a potential bug
> not.**

Manfred objected that according to the
Developers Manual, Vol3, Chapter 19.2
optimize performance, the Pentium pro
be reordered ahead of buffered writes
CPU reads (cache hits) can be reorder
Memory reordering does not occur at t
writes appear in-order." He concluded
would never see the spin_unlock() befo
that on a Pentium, Manfred was right.
from the Pentium Pro manual, "The onl
PentiumPro processor is the added sup
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.**
>
> **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 is so infintesimally 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**

> **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.**
>
> **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.**

> **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, 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

```
                mb();
                a = 0;
                mb();
                b = a;
                spin_unlock();
                return b;
        }
```

> **Now, OBVIOUSLY the above always has to return 0,**
> **ock, and we**
> **and**
> **, the**
>
> **osed**
> **real life,**
> **in the**
> **her**
> **ng**
> **even with**
>
> **your new**
> **here is**
> **ctions**
>
> **data**
> **up doing**
>
> **ck();**
>
> **ordering**
> **is legal,**
> **anations**
> **nple**
> **ut not the**
> **hat**
>
> **netry can**
> **of**
> **feel**
>
> **this..**

e lock()
pparently

In fact, as Peter Samuelson pointed out to me after KT publication (and many thanks to him for it):

"You report that Linus was convinced to do the spinlock optimization on Intel, but apparently someone has since changed his mind back. See <asm-i386/spinlock.h> from 2.3.30pre5 and above:

```
/*
 * Sadly, some early PPro chips require the locked access,
 * otherwise we could just always simply do
 *
 * #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Manfred Spraul:

We can 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.

Why Care? – A Motivating Tale

Ingo Molnar:

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

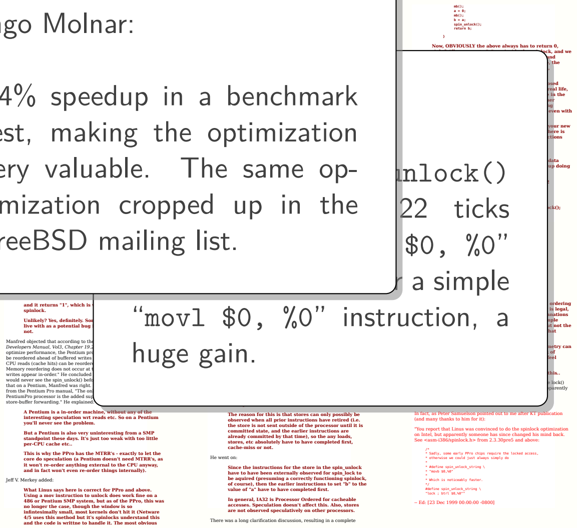
unlock()

22 ticks

$0, %0”

a simple

“movl $0, %0” instruction, a huge gain.



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

}
```

Now, OBVIOUSLY the above always has to return 0,

lock, and we

and

, the

osed

real life,

in the

her

ng

even with

your new

here is

ctions

data

up doing

ock();

and it returns "1", which is

spinlock.

Unlikely? Yes, definitely. So

live with as a potential bug

not.

Manfred objected that according to the
Developers Manual, Vol3, Chapter 19.2
optimize performance, the Pentium pro
be reordered ahead of buffered writes
CPU reads (cache hits) can be reorder
Memory reordering does not occur at t
writes appear in-order." He concluded
would never see the spin_unlock() befo
that on a Pentium, Manfred was right.
from the Pentium Pro manual, "The on
PentiumPro processor is the added sup
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.

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 is so infintesimally 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

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, 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

ordering

is legal,

anations

nple

ut not the

hat

metry can

of

feel

this..

e lock()

pparently

In fact, as Peter Samuelson pointed out to me after KT publication (and many thanks to him for it):

"You report that Linus was convinced to do the spinlock optimization on Intel, but apparently someone has since changed his mind back. See <asm-i386/spinlock.h> from 2.3.30pre5 and above:

```
/*
* Sadly, some early PPro chips require the locked access,
* otherwise we could just always simply do
*
* #define spin_unlock_string \
* "movb $0,%0"
*
* Which is noticeably faster.
*/
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale

Ingo Molnar:

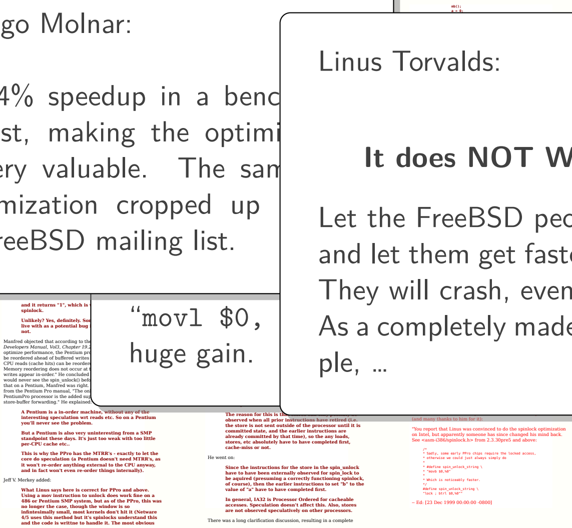
...4% speedup in a benc
test, making the optimi
very valuable. The sam
timization cropped up
FreeBSD mailing list.

"movl $0,
huge gain.

Linus Torvalds:

It does NOT WORK!

Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually. As a completely made up example, ...



```
mb();
a = 0;
```

and it returns "1", which is
spinlock.

Unlikely? Yes, definitely. So
live with as a potential bug
not.

Manfred objected that according to the
Developers Manual, Vol3, Chapter 19.2
optimize performance, the Pentium pro
be reordered ahead of buffered writes
CPU reads (cache hits) can be reorder
Memory reordering does not occur at t
writes appear in-order." He concluded
would never see the spin_unlock() befo
that on a Pentium, Manfred was right.
from the Pentium Pro manual, "The on
PentiumPro processor is the added sup
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.

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 is so infintesimally 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

The reason for this is tha
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, 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

(and many thanks to him for it):

"You report that Linus was convinced to do the spinlock optimization on Intel, but apparently someone has since changed his mind back. See <asm-i386/spinlock.h> from 2.3.30pre5 and above:

```
/*
 * Sadly, some early PPro chips require the locked access,
 * otherwise we could just always simply do
 *
 * #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale

Manfred Spraul:

...according to the Pentium Processor Family Developers Manual, Vol3, 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."

Your example cannot happen.

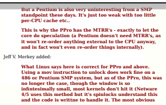
ORK!

ple use it,

r timings.

ually.

up exam-

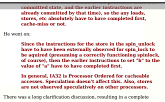


But a Pentium is also very uninteresting from a SMP standpoint these days. It's just too weak with too little per-CPU cache etc..

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 is so infintesimally 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



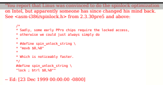
committed state, and the earlier instructions are already committed by that time), so the any loads, stores, etc absolutely have to have completed first, cache-miss or not.

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, 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



"You report that Linus was convinced to do the spinlock optimization on Intel, but apparently someone has since changed his mind back. See <asm-i386/spinlock.h> from 2.3.30pre5 and above:

```
/*
* Sadly, some early PPro chips require the locked access,
* otherwise we could just always simply do
*
* #define spin_unlock_string \
* "movb $0,%0"
*
* Which is noticeably faster.
*/
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale

Manfred Spraul:

...according to the Pentium Processor Family Developers Manual, Vol3, Chapter 19.2 Memory Access Ordering, "to optimize performance, the Pe

reordered ahead of b

CPU reads (cache h

Memory reordering c

and writes appear in

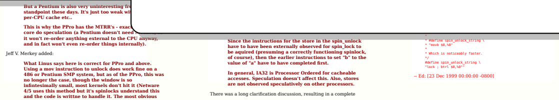
Your

ORK!

Linus Torvalds:

from the Pentium Pro manual, "The only enhancement in the PentiumPro processor is the added support for speculative reads and store-buffer forwarding."

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.**



But a Pentium is also very uninteresting fr
standpoint these days. It's just too weak wit
per-CPU cache etc..

This is why the PPro has the MTRR's - exact
core do speculation (a Pentium doesn't need
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 is so infintesimally 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

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, 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

```
 *
 * #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale

Manfred Spraul:

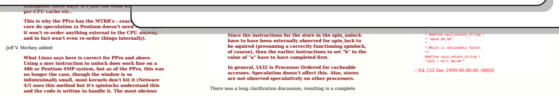
ORK!

ly enhancement
ded support for
rding."

nout any of the
o on a Pentium

Jeff V. Markey:

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.



standpoint these days. It's just too weak wi
per-CPU cache etc..

This is why the PPro has the MTRR's - exac
core do speculation (a Pentium doesn't need
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 is so
infintesimally 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

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, 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

```
 *
 * #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale

Manfred Spraul:

ORK!

Jeff V. Markey:

I have seen the behavior Linus describes on a hard-ware analyzer, BUT O

WERE PPRO AND A

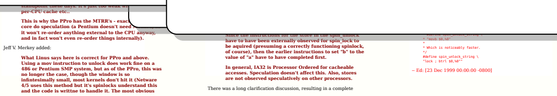
ple must still be on c

that's why they don't

cases.

Erich Boleyn (**Architect, Intel**):

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


standpoint these days. It's just too weak wi
per-CPU cache etc..

This is why the PPro has the MTRR's - exac
core do speculation (a Pentium doesn't need
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 is so
infintesimally 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

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, 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

```
 #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0""
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Why Care? – A Motivating Tale

Linus Torvalds:

I feel comfortable again.

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

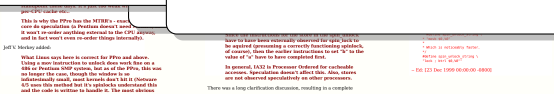
ORK!

n a hard-

ple must still be on c
that's why they don't
cases.

Erich Boleyn (**Architect, Intel**):

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



per-CPU cache etc..

This is why the PPro has the MTRR's - exac
core do speculation (a Pentium doesn't need
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 is so
infintesimally 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

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, 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

```
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0"
```

-- Ed: [23 Dec 1999 00:00:00 -0800]

Lecture Plan

Recall: Dekker/SB test

Initially: $x = 0; \ y = 0;$	
Thread 0	Thread 1

`x := 1;`
`r`$_0$` := y;`
	`y := 1;`
	`r`$_1$` := x;`

Finally: $r_0 = 0 \wedge r_1 = 0$??	

- Forbidden on SC
- Observed on x86

- How come?

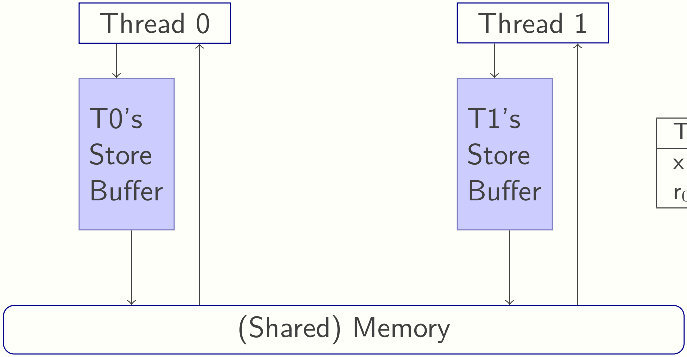
Microarchitecture Interlude: Store Buffering

- Storing to memory is expensive
- Thread has to gain exclusive ownership of location

- In practice, thread buffers stores
- ...letting the thread go ahead if it can

- We *think* x86 has FIFO store buffers

An Explanation



x = y = 0 initially

Thread 0	Thread 1
x := 1	y := 1
r_0 := y	r_1 := x

Another Test: SB+rfi-pos

Initially: $x = 0; \; y = 0;$	
Thread 0	Thread 1
`x := 1;` r_2 `:= x;` r_0 `:= y;`	`y := 1;` r_3 `:= y;` r_1 `:= x;`
Finally: $r_0 = 0 \land r_1 = 0 \land r_2 = 0 \land r_3 = 0$??	

- Not observed on x86
- Threads *required* to read from local store buffer

Regaining order when needed

- Suppose you wanted to program, e.g. mutual exclusion
- Need to regain strong ordering
- `MFENCE` memory barrier

Fences: SB+mfences

Initially: $x = 0;\ y = 0;$	
Thread 0	Thread 1

`x := 1;`
`MFENCE();`
$\texttt{r}_0$ `:= y;`
	`y := 1;`
	`MFENCE();`
	$\texttt{r}_1$ `:= x;`

Finally: $r_0 = 0 \wedge r_1 = 0$??	

- Not observed on x86
- Store buffer must be emptied (flushed) at fence

Atomics

- x86 is very much not RISC

- Instructions like `INC`rement
- Can be made atomic by using a `LOCK` prefix
- ...in the early days, literally locked the bus

Atomics

- x86 is very much not RISC

- Instructions like `INC`rement
- Can be made atomic by using a `LOCK` prefix
- ...in the early days, literally locked the bus

- Instructions like `LOCK CMPXCHG`
- Atomic (either exchanges if memory as expected, or not)

Atomics

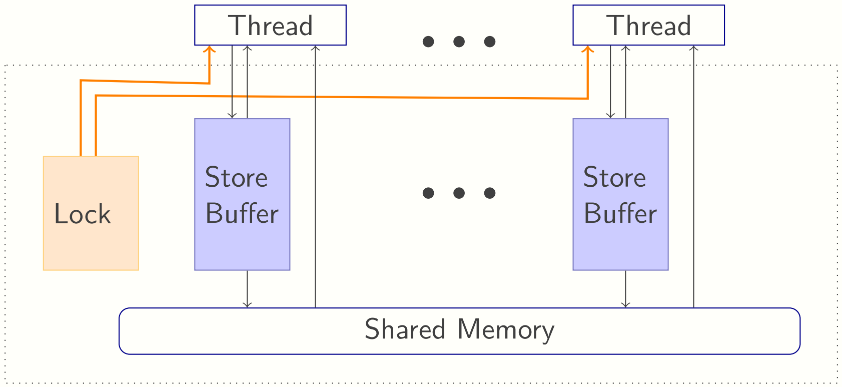
- x86 is very much not RISC

- Instructions like `INC`rement
- Can be made atomic by using a `LOCK` prefix
- ...in the early days, literally locked the bus

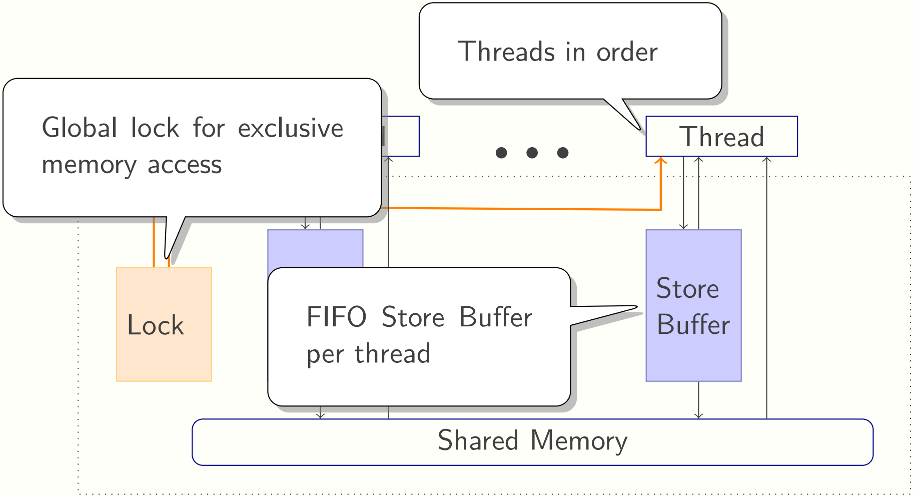
- Instructions like `LOCK CMPXCHG`
- Atomic (either exchanges if memory as expected, or not)

- Store Buffers must be empty
- Note: fence effect

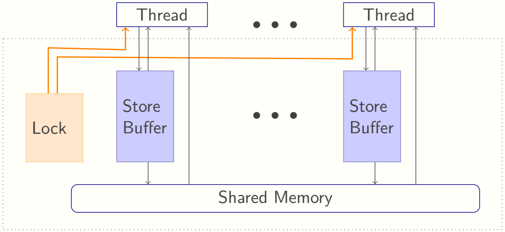
x86-TSO: An abstract machine



x86-TSO: An abstract machine

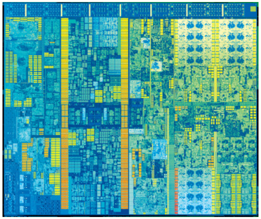

Threads in order
Global lock for exclusive memory access
Thread
• • •
Lock
FIFO Store Buffer per thread
Store Buffer
Shared Memory

x86-TSO: An *abstract* machine



$\supseteq_{\text{beh}}$

$\neq_{\text{hw}}$



Force: Of the internal optimizations of processors, *only* per-thread FIFO write buffers are visible to programmers.

Still quite a loose spec: unbounded buffers, nondeterministic unbuffering, arbitrary interleaving

Formalising Models

Syntactic Domains:

r	Local variables
x	Shared locations
v	Values (integers)

Commands:

$$\begin{array}{lcl} e & ::= & r \mid v \mid e_1 + e_2 \mid \ldots \\ c & ::= & \texttt{skip} \mid c_1; c_2 \mid r := e \mid r := x \mid x := e \mid \\ & & \text{if } r \text{ then } c \mid \ldots \end{array}$$

Programs:

$p ::= c_1 \mid \ldots \mid c_n$

Operational Models

- We sketched an abstract machine
- Relatable to microarchitectural intuitions
- Formalised as a (Labelled) Transition System

Formalisation: Outcomes

- The transition system has states, initialised to initial values
- Behaviour is a function on the final state (representing a final observation)
- A satisfying state is reachable iff behaviour allowed

Model Structure

Typically, (inspired by microarchitectural intuitions):

Separate

Thread subsystem (what the thread knows and can execute by itself)

from

Storage subsystem (what the interconnection knows and can execute)

Synchronising when they communicate (hence the labels)

Model Structure

Typically, (inspired by microarchitectural intuitions):

Separate

Thread subsystem (what the thread knows and can execute by itself)

from

Storage subsystem (what the interconnection knows and can execute)

Synchronising when they communicate (hence the labels)

Not a forced decision (we can divide responsibility *almost* arbitrarily)
...but turns out simpler to abstract actual hardware

Sequential Consistency, Operationally – 1

Thread subsystem: execute thread local operations in order

$$\frac{}{x := v, s \xrightarrow{t:Wxv} \mathsf{skip}, s}\ \text{T-Store}$$

$$\frac{}{r := x, s \xrightarrow{t:Rxv} \mathsf{skip}, s[r \mapsto (v)]}\ \text{T-Load}$$

and rules for composition and local (silent) transitions

Sequential Consistency, Operationally – 2

Storage subsystem: state is just a
Memory $M : \mathit{loc} \to \mathit{val}$

$$\frac{}{M \xrightarrow{t:Wxv} M[x \mapsto v]}\ \text{SCS-Store}$$

$$\frac{M(x) = v}{M \xrightarrow{t:Rxv} M}\ \text{SCS-Load}$$

Thread and storage subsystems synchronise when their labels match
(and can take silent transitions when they are unlabelled transitions)

x86-TSO, Operationally – 1

Same thread subsystem as in SC

Storage subsystem: store is a combination of

- Memory $M : loc \rightarrow val$
- Buffers $B : threadid \rightarrow (loc \times val)^\star$

x86-TSO, Operationally – 2

Important storage subsystem rules:

$$\frac{}{\langle M, B\rangle \xrightarrow{t:Wxv} \langle M, B[t \mapsto \langle x, v\rangle \cdot B(t)]\rangle}\ \text{TSOS-Store}$$

$$\frac{B(t) = b \cdot \langle x, v\rangle}{\langle M, B\rangle \xrightarrow{\epsilon} \langle M[x \mapsto v], B[t \mapsto b]\rangle}\ \text{TSOS-Mem}$$

$$\frac{M(x) = v \quad B(t) \text{ is free of loc } x}{\langle M, B\rangle \xrightarrow{t:Rxv} \langle M, B\rangle}\ \text{TSOS-LoadMem}$$

$$\frac{B(t) = b_1 \cdot \langle x, v\rangle \cdot b_2 \quad b_2 \text{ is free of loc } x}{\langle M, B\rangle \xrightarrow{t:Rxv} \langle M, B\rangle}\ \text{TSOS-LoadBuf}$$

Axiomatic Models

- Can be much more abstract in modeling
- Axiomatic (or declarative) models are very different in style

Axiomatic Models

- Can be much more abstract in modeling
- Axiomatic (or declarative) models are very different in style
- The idea: consider candidate executions of a program, containing all events (memory events) in one execution
- Have a set of rules (axioms) to say if each such is legal

Axiomatic Models

- Can be much more abstract in modeling
- Axiomatic (or declarative) models are very different in style
- The idea: consider candidate executions of a program, containing all events (memory events) in one execution
- Have a set of rules (axioms) to say if each such is legal
- Turns out most axioms can be phrased as
  rules over binary relations and their composition (algebra of relations)

Candidate Executions - Part 1

A candidate execution is: a set of memory events (stores, loads, fences etc), together with ...

the memory events corresponding to a complete execution path through each thread

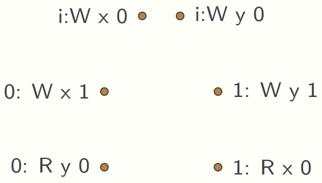
with loads getting *some* values

Illustration

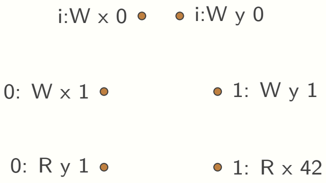
x = y = 0 initially

Thread 0	Thread 1
x := 1	y := 1
r_0 := y	r_1 := x

Execution 1



Execution 2



...

Candidate Executions - Part 2

A candidate execution is: a set of memory events (stores, loads, fences etc), together with relations:

- program-order (po);
- reads-from (rf);
- ...

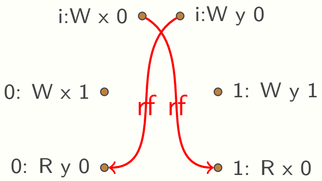
Reads-from relates a Store to a Load that gets its value

Illustration – 2

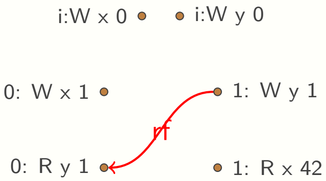
x = y = 0 initially

Thread 0	Thread 1
x := 1	y := 1
r_0 := y	r_1 := x

Execution 1



Execution 2



...

An axiomatic SC model

Suppose E is a candidate execution

If there is a total order S on the events of E;
such that $E.po \subseteq S$; and
$E.rf \subseteq S$; and
if $(W, R) \in E.rf$ then there is no $W' \neq W$ to the same location between W and R in S;

then E is SC-consistent

Basically Lamport's definition: "some sequential order, respecting the order specified by the program."

Execution Candidate (last piece): Coherence Order

Coherence order (co): For every location, a linear order of stores *to that location*

All common hardware platforms ensure there is one such order per location such that all observations are consistent with the order

...Because hardware (cache protocols) are designed for this

Coherence implications

This leads to an important *derived* relation:

$\mathbf{fr} \equiv \mathsf{rf}^{-1}; \mathsf{co}$

Intuition: reads-before in coherence order

An alternative axiomatic SC model

Suppose E is a candidate execution

If $(E.po \mid E.rf \mid E.co \mid E.fr)^\star$ is acyclic,

then E is SC-consistent

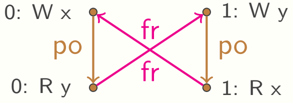
Equivalent to Lamport's definition

Alternative model advantages

Phrased as an acyclicity check

This lets us generate tests which have cycles in certain relations

The `diy7` tool will let us generate litmus tests from such a definition

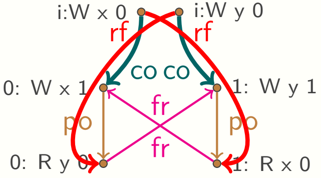


Alternative model advantages

Phrased as an acyclicity check

This lets us generate tests which have cycles in certain relations

The `diy7` tool will let us generate litmus tests from such a definition



Axiomatic vs Operational

Axiomatic Models

- Easier to state
- Checking behaviour is easier
- Entire executions; modularising is not obvious

Operational Models

- State machine intuition
- Abstracts machine behaviour
- Can perform incremental calculation

Relating the Models

Naturally, want both styles

...and want them to be equivalent

Operationally allowed behaviours should *all* be axiomatically allowed (look at traces)

Axiomatically allowed behaviours should *all* be operationally allowed (use nondeterminism in operational models)

An axiomatic x86-TSO model

x86-TSO has a more complex (than SC) model, *but not much more*

Essentially have to account for:

- Store-to-Load on same thread is not part of globally enforced order
- Loads get ordered only if they see a value from other threads
- Same-thread coherence violations are not allowed
- Fences order everything before to everything after

See `x86tso.cat` distributed with `herd7` for more details

Outline

Weaker memory models

- x86-TSO is a relatively strong memory model
- SPARC is similar

- IBM Power much weaker (more interesting)
- ARM similar (since ARMv8 stronger in a particular way)
- RISC-V “RVWMO” similar to ARMv8

Litmus test: Message Passing (MP)

Initially: $d = 0;\ f = 0;$	
Thread 0	Thread 1

`d := 1;`
`f := 1;`
	`while (f == 0)`
	`    {};`
	`r := d;`

Finally: $r = 0$??	

- Forbidden on SC
- Forbidden on x86-TSO
- Observed on Power7 (1.7G/167G)
- ...and on ARM Tegra3 (138k/16M)

Explaining MP via Microarchitecture

Three possible explanations (at least):

- Thread core execution does stores out of order
- Stores propagate between threads out of order
- Thread core execution does loads out of order

Power and ARM and RISC-V can do *all three*

LB test

Initially: $x = 0;\ y = 0;$	
Thread 0	Thread 1

`r`$_0$ `:= y;`
`x := 1;`
	`r`$_1$ `:= x;`
	`y := 1;`

Finally: $r_0 = 1 \land r_1 = 1$??	

- SC and TSO forbid this
- ARM, Power, RISC-V allow this
- ...though current hardware do not show this

Fences

- All these architectures have a *full* fence like MFENCE
- But also weaker fences which are (usually) cheaper to only stop some reorderings
- Power: `lwsync` only orders stores-to-stores, loads-to-loads and stores; `eieio` only orders stores-to-stores
- ARM: `dmb.ld` only orders loads-to-loads and stores; `dmb.st` only orders stores-to-stores
- RISCV: `fence p,s` for all combinations $p, s \subseteq r, w$

Dependencies

In MP, if the stores are somehow stopped from being reordered (store-store fence), load reordering can still show us the questionable result

But usually, algorithms are written with some dependencies (when you do a load and use the resulting value)

Some (but not all) dependencies are guaranteed to be respected by hardware

...and some code (e.g. Linux RCU) does really depend on this

Dependencies

Address Dependency
value read by one load is used to calculate *address* of subsequent load/store

Data Dependency
value read by one load is used to calculate *value* of subsequent store

Dependencies

Address Dependency
value read by one load is used to calculate *address* of subsequent load/store

Data Dependency
value read by one load is used to calculate *value* of subsequent store

Control Dependency
value read by one load is used to calculate *whether to perform* subsequent events

Dependencies

Address Dependency
value read by one load is used to calculate *address* of subsequent load/store

Data Dependency
value read by one load is used to calculate *value* of subsequent store

Control Dependency
value read by one load is used to calculate *whether to perform* subsequent events

Control-Isync Dependency
value read by one load is used to calculate *whether to perform* subsequent events, with intervening `isync`

Dependencies

Address Dependency
value read by one load is used to calculate *address* of subsequent load/store

Data Dependency
value rea

Control
value rea ent
events

- Sometimes naturally in algorithm
- “Fake” dependencies respected as well

Control-Isync Dependency
value read by one load is used to calculate *whether to perform* subsequent events, with intervening `isync`

Microarchitectural explanation

- **Address dependency** means the subsequent memory event cannot be issued by the thread
- **Data dependency** means the subsequent store cannot be issued by the thread
- **Control dependency** is respected for loads-to-store, as stores are not speculated;
  ...but loads are (observably!) speculated by branch prediction;
  load-to-load not respected
- **Control-isync dependency** is respected (all speculation is stopped)

Doing MP on POWER: MP+lwsync+ctrlisync

Initially: d = 0; f = 0;	
Thread 0	Thread 1

`st d 1;`
`lwsync;`
`st f 1;`
	`loop: ld f rtmp;`
	`      cmp rtmp 0;`
	`      beq loop;`
	`isync;`
	`ld d r;`

Finally: r = 0 ??	

- Forbidden (and not observed) on POWER7, and ARM
- `lwsync` prevents store-store reordering
- `control-isync dependency` prevents load speculation

Thread subsystem

(Unlike x86-TSO)

Power/ARM/RISC-V have *very* interesting thread subsystems

- Allow reordering of memory accesses to different locations
- Allow speculation past branches not (yet) known to be taken
- Pay attention to dependencies
- Forbid reordering across fences of appropriate kinds

Inter-thread communication

In x86-TSO, once a store becomes visible to one other thread, it becomes visible to all other threads

On Power, that is *not necessarily* the case

Technically called lack of **Multi-copy Atomicity**

Litmus test: Iterated Message Passing (WRC)

Initially: $d = 0;\ f = 0;$		
Thread 0	Thread 1	Thread 2

`d := 1;`	`while (d == 0) {};`
	`f := 1;`
		`while (f == 0) {};`
		`isync();`
		`r := d;`

Finally: $r = 0$??		

- The dependencies forbid in-thread reordering
- Observed on Power

Litmus test: Iterated Message Passing (WRC)

Initially: d = 0; f = 0;		
Thread 0	Thread 1	Thread 2

`d := 1;`	`while (d == 0) {};`
	`f := 1;`
		`while (f == 0) {};`
		`isync();`
		`r := d;`

Finally: r = 0 ??		

- The dependencies forbid in-thread reordering
- Observed on Power
- Store on `d` making its way to Thread 1 does not mean it has *also* made its way to Thread 2

Cumulativity in non-Multi-Copy-Atomicity

To restore SC, fences have to do more than just stop reordering

“It’s not just reordering”

Have to make sure current thread-local view is transferred with fence

Hardware folk like to call this “cumulative barriers”

Storage subsystem

For Power, the storage subsystem can just be combination of each thread's local view of memory

Together with point-to-point communication
(no single point of truth aka memory)

Multi-Copy Atomicity

ARM used to be (ARMv7) non-multi-copy-atomic in architecture

Since ARMv8, now multi-copy atomic

Turns out their implementations were not utilising the freedom of the specification

...and they think it is unnecessary

Outline

Programming Language Models

Program in higher level languages... are you safe?

1. Has to be compiled to run on hardware

2. *Furthermore*, compiler can optimise as well

Message Passing, Again

Initially:	$\mathsf{data} = 0;\ \ \mathsf{flag} = 0;$
Thread 0	Thread 1

`data = 1;`
`flag = 1;`
	`r`$_0$ `= data;`
	`while (flag == 0)`
	`    {};`
	`r = data;`

Finally: $\mathsf{r} = 0$	

Message Passing, Again

Initially:	data = 0; flag = 0;
Thread 0	Thread 1

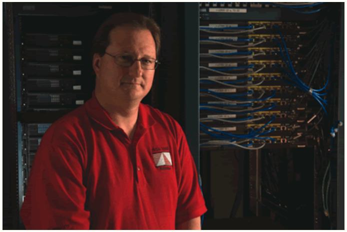
`data = 1;`
`flag = 1;`
	`r`$_0$` = data;`
	`while (flag == 0)`
	`    {};`
	`r = r`$_0$`;`

Finally: r = 0	

- Compiler doing Common Subexpression Elimination
- Can produce unexpected results *even on SC hardware*

Data Race Free Models

Should you even be writing programs like that?
Idea: No! Programmer mistake to write Data Races



Basis of C11 Concurrency

DRF as a programming model

Programs that do not have races (race-free) have only SC behaviour

Programs that have a race in some execution are **Bad**

In C/C++ terms, undefined behaviour

Well, how do you avoid races?

Option 1: Only do "normal" shared-memory accesses guarded by mutexes/locks

- But how do we program locks?
- How about optimisations?

Option 2: Syntactically mark synchronisation accesses

- Since C11/C++11 called `atomic` accesses
- … `_Atomic (Node *) t` in C, or `std::atomic<Node *> t` in C++;
- These are treated specially by compilers

What about low-level algorithms?

C11 introduces marked atomic operations

Races on these are ignored

Can be given parameters (strengths)

- Sequentially consistent (the default)
- Release; Acquire; AcqRel
- Consume
- Relaxed

Happens-before

The C11 model is phrased as an axiomatic model

...with the key relation called happens-before

Nonatomic accesses unrelated by happens-before are data races (UB)

Relaxed accesses do not create happens-before, but do not create data races

Release and Acquire

Happens-before between unlocks and subsequent locks

This is a very common pattern

Essentially, transfer view of one thread at the lock release to the thread acquiring the lock

Release-acquire synchronisation abstracts this phenomenon (when an acquire atomic load reads-from a release atomic store)

MP in C11: mark atomics

Mark atomic variables (accesses have memory order parameter)

Initially: d = 0; f = 0;	
Thread 0	Thread 1

`d.store(1,rlx);`
`f.store(1,rlx);`
	`while (f.load(rlx) == 0)`
	`    {};`
	`r = d.load(rlx);`

Finally: r = 0 ??	

- (Forbidden on SC) (also forbidden on TSO)
- Defined, and possible, in C/C++11
- Allows for hardware (and compiler) optimisations

MP in C11 (contd.): release-acquire synchronisation

Mark release stores and acquire loads

Initially: $d = 0$; $f = 0$;	
Thread 0	Thread 1

`d.store(1,rlx);`
`f.store(1,rel);`
	`while (f.load(acq) == 0)`
	`    {};`
	`r = d.load(rlx);`

Finally: $r = 0$??	

- Forbidden in C/C++11 due to release-acquire synchronisation
- Implementation must ensure result not observed

Outline

Working with Formal Memory Models

Can we implement the C11 model on hardware?

We can now prove correctness of implementation schemes

Compilers require this (they earlier sometimes got it wrong)

Shows correspondence of notions

- Release-acquire synchronisation $\mapsto$ lwsync and ctrlisync
- Transitive part of happens-before $\mapsto$ cumulativity
- ...

Model Checking

- Model checking of code is mature technology
- In the concurrent setting, can detect race bugs (but state-space explosion problems)
- What happens with weak memory?

Model Checking over Weak Memory

- Considering interleavings not sufficient
- Operational models can be explored (nondeterministic interleavings of transition systems
- ...but state space has lots of (unnecessary?) machinery

Program Logics

- Hoare-style assertion reasoning is mainstay of program verification
- Concurrent Separation Logic is a huge success story in concurrent verification
- The standard “heap model” assumes SC
- Not sound for weak memory

Program Logics over Weak Memory

- Can adapt heap model to weak memory (most successful for TSO-like models)
- Can transfer ideas in the opposite way: Release-Acquire is transferring a “view”
- Basis of several RA style logics (work ongoing to scale up to C11)

Much Exciting Research to be Done!

Thank you!