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
Another old idea: Multiprocessors

Parallel hardware/concurrent programs
Another old idea: Multiprocessors

Parallel hardware/concurrent programs

BURROUGHS D825, 1962

Outstanding features include truly modular hardware with parallel processing throughout.

FUTURE PLANS
The complement of compiling languages is to be expanded.
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?
...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
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:

<table>
<thead>
<tr>
<th>Initially: t0wants = FALSE; t1wants = FALSE;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>t0wants = TRUE;</td>
</tr>
<tr>
<td>if (NOT t1wants) {</td>
</tr>
<tr>
<td>{ ...CRITICAL1 };</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>t1wants = TRUE;</td>
</tr>
<tr>
<td>if (NOT t0wants) {</td>
</tr>
<tr>
<td>{ ...CRITICAL2 };</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Does it work?
Example: Mutual Exclusion Litmus Test

Distilling that example

<table>
<thead>
<tr>
<th>Initially:</th>
<th>x = 0; y = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>x := 1;</td>
<td>y := 1;</td>
</tr>
<tr>
<td>r₀ := y;</td>
<td>r₁ := x;</td>
</tr>
<tr>
<td>Finally: r₀ = 0 ∧ r₁ = 0 ??</td>
<td></td>
</tr>
</tbody>
</table>

- Forbidden on SC (no interleaving allows that result)
We use the litmus7 tool (diy.inria.fr, Alglave and Maranget)

SB.litmus

X86 SB
"Fre PodWR Fre PodWR"
{ x=0; y=0; }

PO | P1 ;
MOV [x],$1 | MOV [y],$1 ;
MOV EAX,[y] | MOV EAX,[x] ;
locations [x;y;]
exists (0:EAX=0 \ 1:EAX=0)
$ 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
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

**Upshot:**

No longer a *sequential consistent* memory model

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


Topics: [Discussion]

Erich Boleyn, Ingo Molnar, Mads Haahr, Matt Ralston, Jeff V. Merkey

Kernel Traffic: [Discussion]

Motivated by a script I found on how spin-unlocks does from about 22 ticks for the "lock, spin, unlock" case, to 9 ticks for a simple "spin, spin-lock instructions, a huge gain. Until he rejected that bugs Mular noticed a 4x speed-up on a benchmark test, making the optimization very valuable. Some also added that the optimization worked on the Pentium III, Pentium II and some previous. But Linux has been on the whole thing, I

It doesn't work!

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

The window may be small, but it will be fine if they do, although some spin unlocks aren't reliable anymore.

The issue is not worth being issued in-order (although I don't think it is worth both, until you still have to assume that in-order write behavior - let it be the case in the long run)

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

For example, with a single 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 of '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 lock(x) does not crash now is quite possibly because of either:

• we have a lot of documentation on our spinlocks these days. That might hide the problem, because the compiler, will see the result of the cache references still means that the spinlock itself works fine - it's just that it no longer works when there are true re-orderings as an exclusion

• 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, "lock" may be the worst possible thing to use for this, and you might test whether a simpler "spin" might work better. It's still not a problem, but you have normal 12 cycles that Intel always seems to want on serializing instructions rather than 22 cycles.

Elsewhere, he goes on to explain:

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 CPUs:

```
int test_locking(void)
{  
int a = 0;
   /* cache miss satisfied, the "a" line
   is bouncing back and forth */
   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 no re-orderings will have happened, so the value of "a" will never change.

Erich Boleyn, as a member of the Pentium II development group at Intel also replied to Linux, pointing out a possible exception to the proposed explanation. Regarding the code line pointed, Erich replied:

```
14
a = 0;
mb();
```

"cache misses satisfied, the "a" is becoming back and forth?"

It gets the value 1

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

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

Motivated by that according to the Pentium Processor Family Developers Manual, Intel [Chapter 6.3 Memory Access Ordering] "When a pipeline is unable to maintain a consistent view of buffered writes in most situations, internally, CPU reads from cache lines will not be reordered around buffered writes. Memory reordering does not occur at the time, cache (pipeline) write, and writing in order. "The processor is disabled from intervening in the Pentium processor's access to the data. In any case, if there are no re-ordering instructions around the unlock.

A Pentium is a in-order machine, without any of the preventing speculative set reorder etc. etc. So on a Pentium you'll never see the problem.

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

It's why the PFs have the MTRRs's exactly to lets the cache do speculative (a Pentium doesn't need MTRRs'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 Linux says here is correct for PFs and above. Using some instruction in unlock does work for one (a Pentium), but not for the others, but that was no surprise, though, the window is so

Additionally, one kernel doesn't do it (Netware 4.5 uses this method but it's spinlock understood this and the code is written to handle it. The most obvious profound behavior was that cache inconsistencies would occur randomly. PFs used to be stuck that the pipelines are no longer existing and the buffers shouldn't be blown out.

I have seen the behavior Linux describes on a hardware emulator. This is EMBP, WMCPD and ABOVE. I guess the Intel people must still be on older Pentium hardware that's why they don't see this in the code.

The reason is in this that stores can only possibly be reordered ahead of buffered writes. In most situations, internally, CPU reads from cache lines will not be reordered around buffered writes. Memory reordering does not occur at the time, cache (pipeline) write, and writing in order. "The processor is disabled from intervening in the Pentium processor's access to the data. In any case, if there are no re-ordering instructions around the unlock.

The reason for this is that stores can only possibly be reordered ahead of buffered writes. Still, in most situations, internally, CPU reads from cache lines will not be reordered around buffered writes. Memory reordering does not occur at the time, cache (pipeline) write, and writing in order. "The processor is disabled from intervening in the Pentium processor's access to the data. In any case, if there are no re-ordering instructions around the unlock.

The reason for this is that stores can only possibly be reordered ahead of buffered writes. Still, in most situations, internally, CPU reads from cache lines will not be reordered around buffered writes. Memory reordering does not occur at the time, cache (pipeline) write, and writing in order. "The processor is disabled from intervening in the Pentium processor's access to the data. In any case, if there are no re-ordering instructions around the unlock.

Since the instructions for the store in the spin, unlock have to have been absolutely ordered for spin, lock to be acquired (assuming a correctly functioning spinlock), of course, then the earlier instructions to set "b" to the other variable in the same cache line that is accessed through another spinlock.

And the reason that we can't do it is that the store in the spin, unlock has to have been absolutely ordered for spin, lock to be acquired (assuming a correctly functioning spinlock), of course, then the earlier instructions to set "b" to the other variable in the same cache line that is accessed through another spinlock.

Nobody 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 some explanations for why serialization (as opposed to the simple lock/unlock) is required for the lock() side but not the unlock() side, and that lack of symmetry was what bothered me the most.

Of course, a strong case that the lack of symmetry can be explained by just simple the lack of symmetry set specification of reads and writes. I feel comfortable saying:

Thanks, guys, we'll be a lot 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 Sembolich pointed out to me after KT publication (and even thanks to the above):

"Yes report that Linux was convinced to do the spinlock optimization on Intel, but apparently someone has since changed the Intel code.

I'm not sure if it makes the Pentium II spinlock works without it, but hey, add another access to another variable in the same cache line that is accessed through another spinlock.

In general, the Linux Processor should do 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
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 is the kernel. It crashed, in random ways.

林纳斯·托瓦兹

1. spin_unlock() Optimization On Intel

20 Nov 1999 - 7 Dec 1999 (143 posts)

Kernel Traffic #47 For 20 Dec 1999

---

**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.

```
movl $0, %0
and it returns "1", which is wrong for any working spinlock.
Unlikely? Yes, definitely. See line 40 as a potential bug.
```

```
mb();
a = 0;
and it returns "1", which is wrong for any working spinlock.
```

```
mb();
a = 1;
and it returns "0", which is right.
```

```
spin_unlock
```

```
spin_unlock_string
```

```
lock ; btrl $0,%0"
```

---

**Linus Torvalds:**

```
spin_unlock_string
```

```
lock ; btrl $0,%0"
```

---

```
lock ; btrl $0,%0
```

---

```
lock ; btrl $0,%0
```

---

```
lock ; btrl $0,%0
```

---

```
lock ; btrl $0,%0
```

---

**Jeff V. Merkey:**

```
spin_lock;
a = 1;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
a = 0;
```

---

```
spin_lock;
```
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.

```
#define spin_unlock_string

int test_locking(void)
{
    static int a; /* protected by spinlock */
    int b;
    spin_lock()
    a = 1;
    /* cache miss satisfied, the "a" line is bouncing back and forth */
    b = a;
    spin_unlock();
    return b;
}
```

It will always return 0. You don't need "spin_unlock()" to make the optimization very valuable. In fact, as Peter Samuelson pointed out to me after KT publication turnaround by Linus:

```
#define spin_unlock_string

int test_locking(void)
{
    static int a; /* protected by spinlock */
    int b;
    spin_lock()
    a = 1;
    /* cache miss satisfied, the "a" line is bouncing back and forth */
    b = a;
    spin_unlock();
    return b;
}
```

Note that another thing is that yes, "btcl" may be the instruction, a huge gain.

The reason for this is that stores can only possibly be reordered 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 be serialized (lock) for why serialization (as opposed to just the simple protection of spin lock) rules _are_ strong enough that all of this really is legal, and it returns "1", which is spinlock.

Unlikely? Yes, definitely. See below with a potential hang.

Manfred objected that according to the Developers Manual, Intel's 'Optimal Interleaving of Take the following code to the Pentium (Netware 4/5 uses this method but it's spinlock understands this and the code is written to handle it). The most obvious fix is to return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, but hey, add another access to another variable in the

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

But a Pentium also has a re-fetching from a VM standpoint three days. It's just too weak with too little per-CPU/cache etc. This is why the PPro's has the MMU's and it returns "1", which is spinlock...it'sspinlock understands this and the code is written to handle it. The most obvious fix is to return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, but hey, add another access to another variable in the

```
#define spin_unlock_string

int test_locking(void)
{
    static int a; /* protected by spinlock */
    int b;
    spin_lock()
    a = 1;
    /* cache miss satisfied, the "a" line is bouncing back and forth */
    b = a;
    spin_unlock();
    return b;
}
```

It will always return 0. You don't need "spin_unlock()" to make the optimization very valuable. In fact, as Peter Samuelson pointed out to me after KT publication turnaround by Linus:

```
#define spin_unlock_string

int test_locking(void)
{
    static int a; /* protected by spinlock */
    int b;
    spin_lock()
    a = 1;
    /* cache miss satisfied, the "a" line is bouncing back and forth */
    b = a;
    spin_unlock();
    return b;
}
```

Note that another thing is that yes, "btcl" may be the instruction, a huge gain.

The reason for this is that stores can only possibly be reordered 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 complete first, cache-misses or not. We want:

Since the instructions for the store in the spin unlock have to have been atomically observed for spin lock to be applied (passing a correctly functioning spinlock, of course), then the earlier instructions in set "S" to the value of "a" have to have completed first. In general, 14.12 in PowerPC order for cacheable accesses. Speculation doesn't affect this. Also, stores are not observed specially one other processors.

It was a long clarification discussion, resulting in a complete
Ingo Molnar:

...4% speedup in a benchmark test, making the optimization very valuable. The same optimization 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, ...

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.”

But a Pentium is also very unintuitive from a WoW standpoint these days. It’s just too weak with too little per-CPU/cache etc.

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

Jeff Y. Merkey added:

What Linus says here is correct for PPPros and above. Using spin instructions to lock does work for a Pentium, but again, it is very obvious that the PPro and later WP Pro processors can (and do) re-order the memory accesses. This was no longer the case, though the wisdom is so

unintentionally small, most kernels don’t do it (Netware 4.3 uses this method but it’s opinion is understood this and the code is written to handle it. The most obvious

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

We want:

Since the instructions for the store in the spin lock have to have been externally observed for spin, lock to be signaled (processing a correctly functioning spinlock), of course, then the earlier instructions to set “a” to the value of “b” have to have completed first.

In general, the PPro and Processor-Oriented 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

optimized version for Intel’s MMUs and caches. The

optimized version will work on Intel’s MMUs and

caches. The

- lock ; btrl $0,%0"

/* Which is noticeably faster.

#define spin_unlock_string

*/

* #define spin_unlock_string

*/

#define spin_unlock_string

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

Note that I actually thought this was a legal

behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the

pipelines are not observed speculatively on other processors.

There was a long clarification discussion, resulting in a complete

optimized version for Intel’s MMUs and caches. The

optimized version will work on Intel’s MMUs and

caches. The

- lock ; btrl $0,%0"

/* Which is noticeably faster.

#define spin_unlock_string

*/

* #define spin_unlock_string

*/

#define spin_unlock_string

-- Ed: [23 Dec 1999 00:00:00 -0800]
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. CPU reads (cache hits) and writes appear in-order. Memory reordering does not occur at the pins, reads (cache miss) and writes appear in-order.”

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 an in-order machine, without any of the interesting speculation wrt reads etc. So on a Pentium you’ll never see the problem.
**Why Care? – A Motivating Tale**

**Manfred Spraul:**

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.

---

**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.

---

**Linus Torvalds:**

Unlikely? Yes, definitely. Something we are willing to do.
Why Care? – A Motivating Tale

Manfred Spraul:

I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ALSO PEOPLE MUST STILL BE ON OLDER PENTIUM HARDWARE AND IN THE LONG RUN.

Jeff V. Markey:

...as long as spin lock is protected, and the code is written to handle it. The most obvious access is not observed speculatively on other processors. There was a long clarification discussion, resulting in a complete symmetry wrt speculation of reads vs writes. I feel comfortable again.

Erich Boleyn (Architect, Intel):

It will always return 0. You don’t need “spin_unlock()” to be serializing.
Linus Torvalds:

I feel comfortable again.

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

People must still be on older chips that’s why they don’t show this cases.

Erich Boleyn (Architect, Intel):

It will always return 0. You don’t need “spin_unlock()” to be serializing.
Introduction

SC and x86-TSO Memory Models

ARM, Power, RISC-V Memory Models

Programming Language Memory Models: C11 and Release Acquire

What next? Reasoning over Weak Memory Models
Recall: Dekker/SB test

Initially: \[ x = 0; \quad y = 0; \]

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x := 1; ]</td>
<td>[ y := 1; ]</td>
</tr>
<tr>
<td>[ r_0 := y; ]</td>
<td>[ r_1 := x; ]</td>
</tr>
</tbody>
</table>

Finally: \[ r_0 = 0 \land r_1 = 0 \]

- 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

Thread 0

T0’s Store Buffer

Thread 1

T1’s Store Buffer

(Shared) Memory

\[ x = y = 0 \text{ initially} \]

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1 )</td>
<td></td>
<td>( y := 1 )</td>
</tr>
<tr>
<td>( r_0 := y )</td>
<td>( r_1 := x )</td>
<td></td>
</tr>
</tbody>
</table>
Another Test: SB+rfi-pos

Initially: \( x = 0; \ y = 0; \)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; )</td>
<td>( y := 1; )</td>
</tr>
<tr>
<td>( r_2 := x; )</td>
<td>( r_3 := y; )</td>
</tr>
<tr>
<td>( r_0 := y; )</td>
<td>( r_1 := x; )</td>
</tr>
</tbody>
</table>

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:

$x = 0; \ y = 0$

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1$;</td>
<td>$y := 1$;</td>
</tr>
<tr>
<td>MFENCE();</td>
<td>MFENCE();</td>
</tr>
<tr>
<td>$r_0 := y$;</td>
<td>$r_1 := x$;</td>
</tr>
</tbody>
</table>

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

- **Not** observed on x86

- Store buffer must be **emptied** (**flushed**) at fence
• x86 is very much not RISC

• Instructions like INCrement

• 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 INCrement
- 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)
x86 is very much not RISC

Instructions like INCrement

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

- Global lock for exclusive memory access
- Threads in order
- Lock
- FIFO Store Buffer per thread
- Shared Memory
- Store Buffer
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:
- \( e ::= r \mid v \mid e_1 + e_2 \mid \ldots \)
- \( c ::= \text{skip} \mid c_1; c_2 \mid r := e \mid r := x \mid x := e \mid \text{if } r \text{ then } c \mid \ldots \)

Programs:
- \( p ::= c_1 \mid \ldots \mid c_n \)
We sketched an abstract machine

Relatable to microarchitectural intuitions

Formalised as a (Labelled) Transition System
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.
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)
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
Thread subsystem: execute thread local operations in order

\[
\text{T-Store} \\
\begin{align*}
x &:= v, s \xrightarrow{t:W_{xv}} \text{skip}, s
\end{align*}
\]

\[
\text{T-Load} \\
\begin{align*}
r &:= x, s \xrightarrow{t:R_{xv}} \text{skip}, s[r \mapsto (v)]
\end{align*}
\]

and rules for composition and local (silent) transitions
Storage subsystem: state is just a
Memory $M : loc \rightarrow val$

\[
\begin{align*}
M & \xrightarrow{t : W_{xv}} M[x \mapsto v] \\
SCS-Store \\
M(x) & = v \\
SCS-Load \\
M & \xrightarrow{t : R_{xv}} M
\end{align*}
\]

Thread and storage subsystems synchronise when their labels match
(and can take silent transitions when they are unlabelled transitions)
Same thread subsystem as in SC

Storage subsystem: store is a combination of

- Memory $M : loc \rightarrow val$
- Buffers $B : threadid \rightarrow (loc \times val)^*$
Important storage subsystem rules:

**TSOS-Store**

\[
\langle M, B \rangle \xrightarrow{t:Wxv} \langle M, B[t \mapsto \langle x, v \rangle \cdot B(t)] \rangle
\]

**TSOS-Mem**

\[
B(t) = b \cdot \langle x, v \rangle
\]

\[
\langle M, B \rangle \xrightarrow{c} \langle M[x \mapsto v], B[t \mapsto b] \rangle
\]

**TSOS-LoadMem**

\[
M(x) = v \quad B(t) \text{ is free of loc } x
\]

\[
\langle M, B \rangle \xrightarrow{t:Rxy} \langle M, B \rangle
\]

**TSOS-LoadBuf**

\[
B(t) = b_1 \cdot \langle x, v \rangle \cdot b_2 \quad b_2 \text{ is free of loc } x
\]

\[
\langle M, B \rangle \xrightarrow{t:Rxy} \langle M, B \rangle
\]
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)
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
x = y = 0 initially

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1</td>
<td>y := 1</td>
</tr>
<tr>
<td>r_0 := y</td>
<td>r_1 := x</td>
</tr>
</tbody>
</table>

Execution 1

- i:W x 0
- i:W y 0
- 0: W x 1
- 0: R y 0

Execution 2

- i:W x 0
- i:W y 0
- 0: W x 1
- 1: R x 0
- 0: R y 1
- 1: R x 42
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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1$</td>
<td>$y := 1$</td>
</tr>
<tr>
<td>$r_0 := y$</td>
<td>$r_1 := x$</td>
</tr>
</tbody>
</table>

Execution 1

1. $i:W x 0$
2. $i:W y 0$
3. $0: W x 1$
4. $1: W y 1$
5. $0: R y 0$
6. $1: R y 1$

Execution 2

1. $i:W x 0$
2. $i:W y 0$
3. $0: W x 1$
4. $1: W y 1$
5. $0: R y 1$
6. $1: R x 42$
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.”
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
This leads to an important derived relation:

\[ \text{fr} \equiv \text{rf}^{-1}; \text{co} \]

Intuition: reads-before in coherence order
An alternative axiomatic SC model

Suppose $E$ is a candidate execution
If $(E.po \mid E.rf \mid E.co \mid E.fr)^*$ is acyclic,
then $E$ is SC-consistent

Equivalent to Lamport’s definition
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 \emph{all} be axiomatically allowed (look at traces)

Axiomatically allowed behaviours should \emph{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

1. Introduction
2. SC and x86-TSO Memory Models
3. ARM, Power, RISC-V Memory Models
4. Programming Language Memory Models: C11 and Release Acquire
5. What next? Reasoning over Weak Memory Models
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)

<table>
<thead>
<tr>
<th>Initially:</th>
<th>d = 0; f = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>d := 1;</td>
<td>while (f == 0)</td>
</tr>
<tr>
<td>f := 1;</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>r := d;</td>
</tr>
<tr>
<td>Finally: r = 0 ??</td>
<td></td>
</tr>
</tbody>
</table>

- Forbidden on SC
- Forbidden on x86-TSO
- Observed on Power7 (1.7G/167G)
- ...and on ARM Tegra3 (138k/16M)
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*
Initially: \[ x = 0; \quad y = 0; \]

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_0 := y; )</td>
<td>( r_1 := x; )</td>
</tr>
<tr>
<td>( x := 1; )</td>
<td>( y := 1; )</td>
</tr>
</tbody>
</table>

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 $p, s \subseteq r, w$
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**

A value read by one load is used to calculate the address of subsequent load/store.

**Data Dependency**

A value read by one load is used to calculate the 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 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

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

<table>
<thead>
<tr>
<th>Initially:</th>
<th>d = 0; f = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread 0</td>
</tr>
<tr>
<td>st d 1;</td>
<td>loop: ld f rtmp;</td>
</tr>
<tr>
<td>lwsync;</td>
<td>cmp rtmp 0;</td>
</tr>
<tr>
<td>st f 1;</td>
<td>beq loop;</td>
</tr>
<tr>
<td></td>
<td>isync;</td>
</tr>
<tr>
<td></td>
<td>ld d r;</td>
</tr>
<tr>
<td>Finally: r = 0 ??</td>
<td></td>
</tr>
</tbody>
</table>

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

(Unlike x86-PSO)
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)

<table>
<thead>
<tr>
<th>Initially:</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d = 0; f = 0;</code></td>
<td><code>d := 1;</code></td>
<td><code>while (d == 0) {};</code></td>
<td><code>while (f == 0) {};</code></td>
</tr>
<tr>
<td></td>
<td><code>f := 1;</code></td>
<td></td>
<td><code>isync();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>r := d;</code></td>
</tr>
<tr>
<td>Finally: <code>r = 0</code> ??</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The dependencies forbid in-thread reordering
- Observed on Power
Listmus test: Iterated Message Passing (WRC)

<table>
<thead>
<tr>
<th>Initially:</th>
<th>( d = 0; \ f = 0; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td></td>
</tr>
<tr>
<td>( d := 1; )</td>
<td></td>
</tr>
<tr>
<td>while ((d == 0))</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>( f := 1; )</td>
<td></td>
</tr>
<tr>
<td>Thread 1</td>
<td></td>
</tr>
<tr>
<td>while ((d == 0))</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>while ((f == 0))</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>isync()</td>
<td></td>
</tr>
<tr>
<td>( r := d; )</td>
<td></td>
</tr>
</tbody>
</table>

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”
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)
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

1. Introduction
2. SC and x86-TSO Memory Models
3. ARM, Power, RISC-V Memory Models
4. Programming Language Memory Models: C11 and Release Acquire
5. What next? Reasoning over Weak Memory Models
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

<table>
<thead>
<tr>
<th>Initially:</th>
<th>data = 0; flag = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>data = 1;</td>
<td>r0 = data;</td>
</tr>
<tr>
<td>flag = 1;</td>
<td>while (flag == 0)</td>
</tr>
<tr>
<td></td>
<td>{};</td>
</tr>
<tr>
<td></td>
<td>r = data;</td>
</tr>
<tr>
<td>Finally:</td>
<td>r = 0</td>
</tr>
</tbody>
</table>
Message Passing, Again

<table>
<thead>
<tr>
<th>Initially:</th>
<th>data = 0; flag = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>data = 1; flag = 1;</td>
</tr>
<tr>
<td>Thread 1</td>
<td>r₀ = data;</td>
</tr>
<tr>
<td></td>
<td>while (flag == 0)</td>
</tr>
<tr>
<td></td>
<td>{};</td>
</tr>
<tr>
<td></td>
<td>r = r₀;</td>
</tr>
<tr>
<td>Finally:</td>
<td>r = 0</td>
</tr>
</tbody>
</table>

- 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
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
The C11 model is phrased as an axiomatic model

...with the key relation called \texttt{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
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)
Mark atomic variables (accesses have memory order parameter)

<table>
<thead>
<tr>
<th>Initially:</th>
<th>d = 0; f = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>d.store(1,rlx); f.store(1,rlx);</td>
</tr>
</tbody>
</table>
| Thread 1   | 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
Mark release stores and acquire loads

<table>
<thead>
<tr>
<th>Initially:</th>
<th>d = 0; f = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
</tbody>
</table>
| 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

1 Introduction

2 SC and x86-TSO Memory Models

3 ARM, Power, RISC-V Memory Models

4 Programming Language Memory Models: C11 and Release Acquire

5 What next? Reasoning over Weak 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 $\leftrightarrow$ lwsync and ctrlisync
  - Transitive part of happens-before $\leftrightarrow$ 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?
- Considering interleavings not sufficient

- Operational models can be explored (nondeterministic interleavings of transition systems)

- ...but state space has lots of (unnecessary?) machinery
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
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!