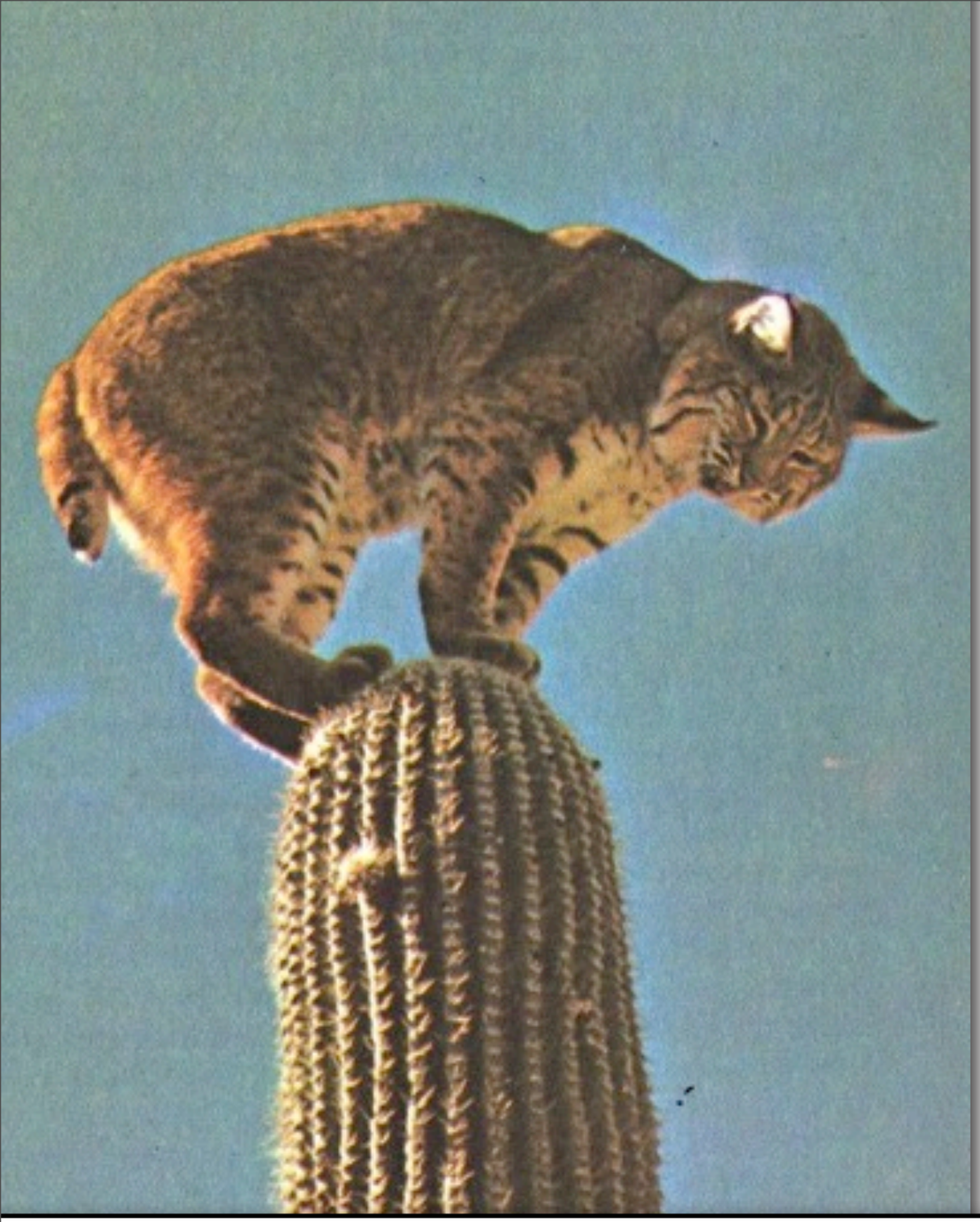


Languages and concurrency

a thorny relationship

Francesco Zappa Nardelli

Inria, France

Based on work *done by* or *with*

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

U. Cambridge, U. Kent, MPI-SWS, Inria

Shared memory

(according to Wikipedia)

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

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

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

Shared memory

(according to Wikipedia)

...relatively easy to program...

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

...bottleneck to performance...

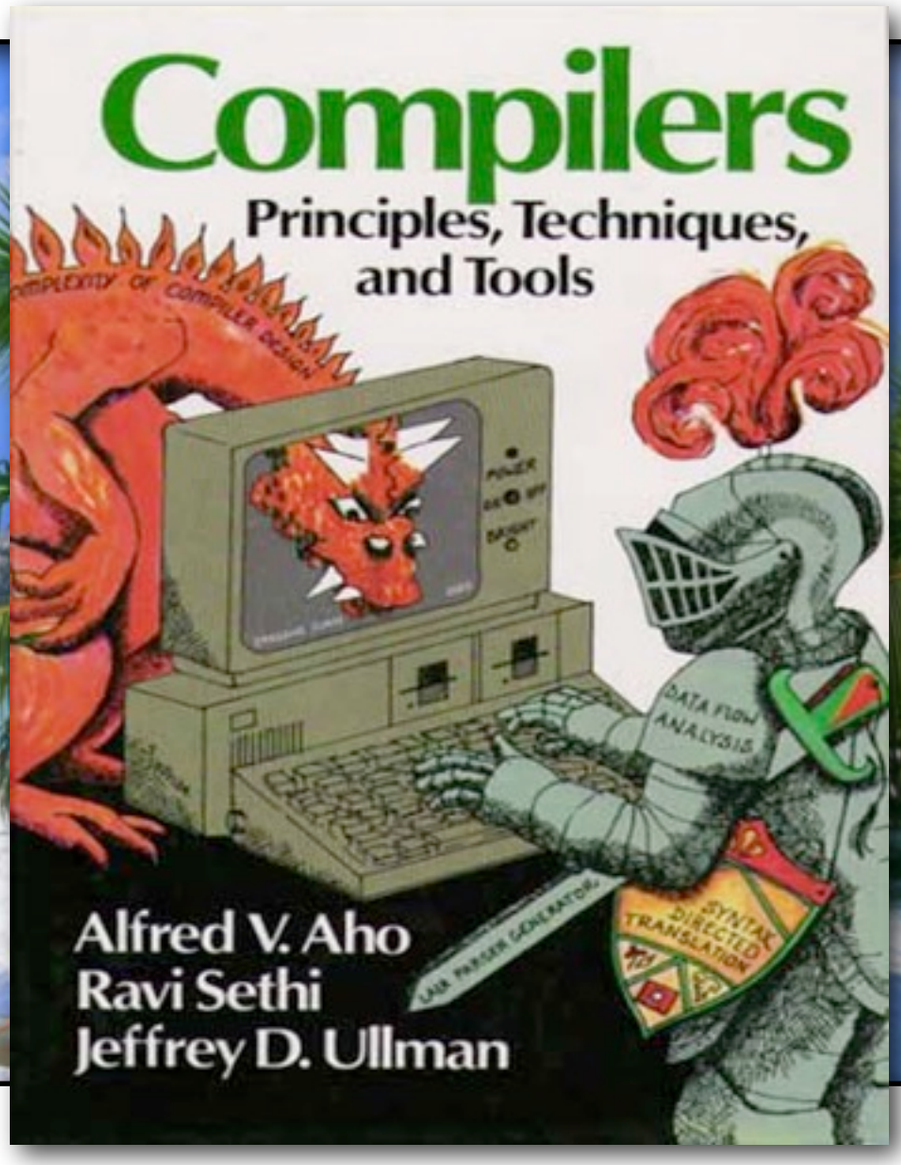
In computer h s memory (RAM) that ca iple-processor computer sys

A shared memory system is relatively easy to program since all processors share a single view of data and th a same location. The issue y and will likely cache memo

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

Imagine an ideal world

Imagine an ideal world



Programmers and compilers cooperate
to make great software

Constant propagation

A simple, and *innocuous*, optimisation:

Source code

```
x = 14
y = 7 - x / 2
```

Optimised code

```
x = 14
y = 7 - 14 / 2
```

```
x = 14
y = 0
```

Shared memory concurrency

Shared memory

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
```

```
if (x == 1) {
  x = 0
  y = 1 }
```

Thread 2

Shared memory concurrency

Shared memory

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
```

```
if (x == 1) {
  x = 0
  y = 1 }
```

Thread 2

Intuitively this program always prints 0

Shared memory concurrency

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

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
```

Shared memory concurrency

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

```
x = y = 0
```

Thread 1

```
x = 1
if (y == 1)
  print x
   print 1
```

(In Thread 1, `print x` is struck through and replaced by `print 1`.)

Thread 2

```
if (x == 1) {
  x = 0
  y = 1 }
```

...the program always writes 1 rather than 0.

Shared memory concurrency

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

```
x = y = 0
```

A compiler can break your code

...the program always writes 1 rather than 0.

That pesky hardware (1)

Consider misaligned 4-byte accesses:

`int32_t a = 0`	
`a = 0x44332211`	`if (a == 0x00002211)` `  print "error"`

(*Disclaimer*: compiler will normally ensure alignment)

Intel SDM x86 atomic accesses:

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

Question: what about *multi-word high-level language values*?

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`

Can you guess the final register values: **`EAX`** = ? **`EBX`** = ?

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**

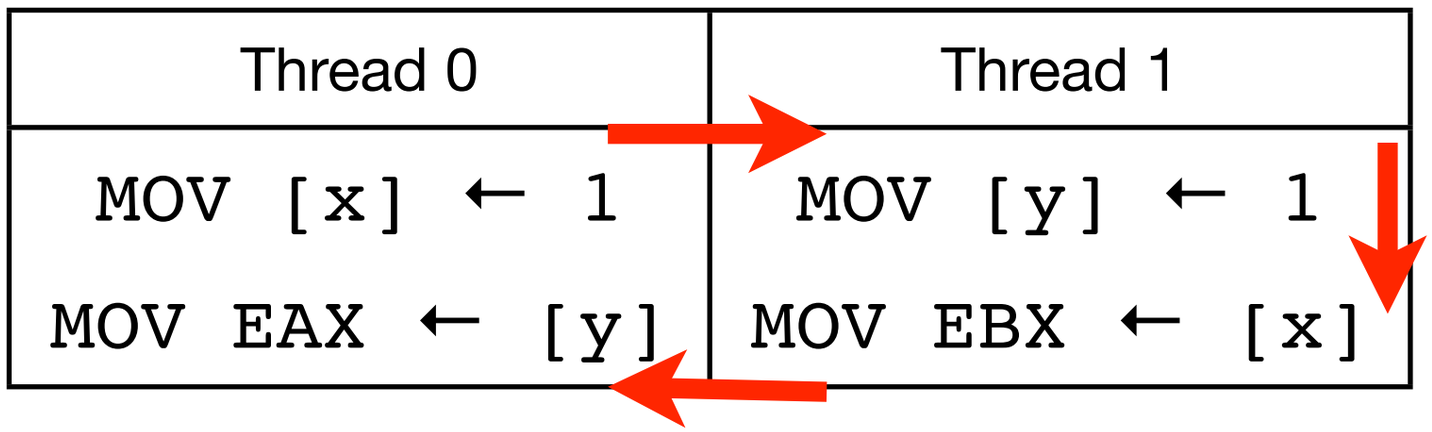
Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`

Can you guess the final register values: **`EAX`** = 1 **`EBX`** = 1

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**

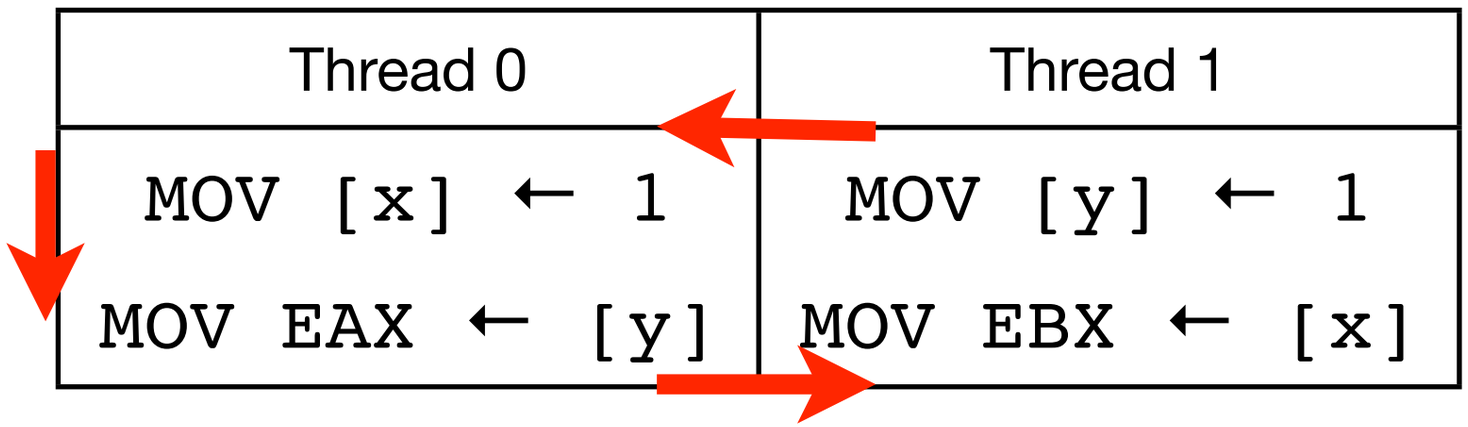


Can you guess the final register values: **`EAX`** = 1 **`EBX`** = 1

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**

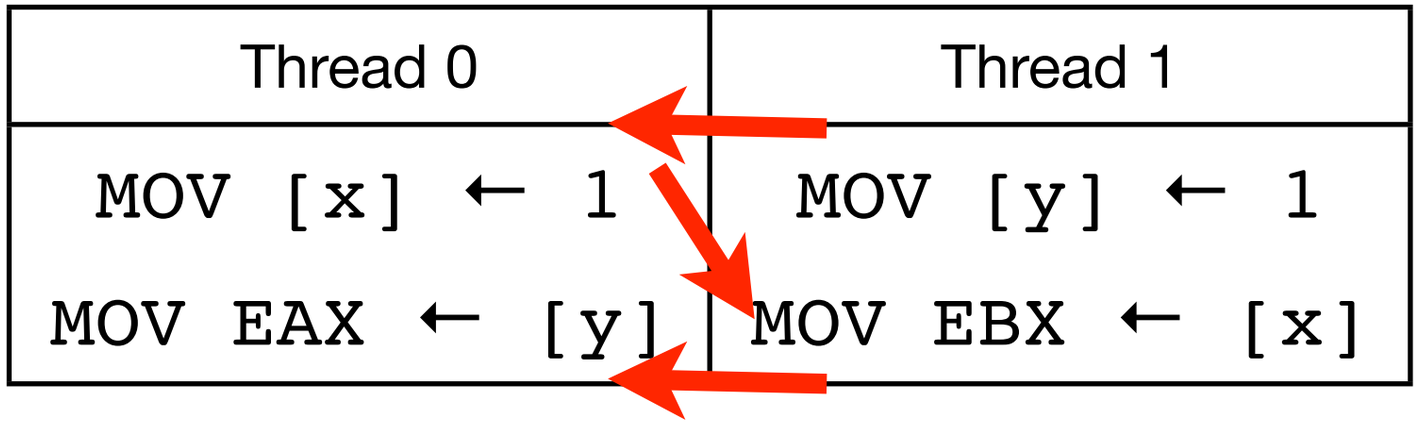


Can you guess the final register values: **`EAX`** = 1 **`EBX`** = 1

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**



Can you guess the final register values: **`EAX`** = 1 **`EBX`** = 1

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`

Can you guess the final register values: **`EAX`** = 1 **`EBX`** = 0

That pesky hardware (2)

Initial shared memory values: `[x]=0` `[y]=0`

Per-processor registers: **`EAX`** **`EBX`**

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`

Can you guess the final register values: **`EAX`** = 0 **`EBX`** = 1

That pesky hardware (2)

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`

The possible outcomes should be:

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

That pesky hardware (2)

Thread 0	Thread 1
MOV	
MOV	

Let's see...

The po

- `EAX`
- `EAX`
- `EAX`

That pesky hardware (2)

Thread 0	Thread 1
MOV	
MOV	

We can observe

$$EAX = EBX = 0$$

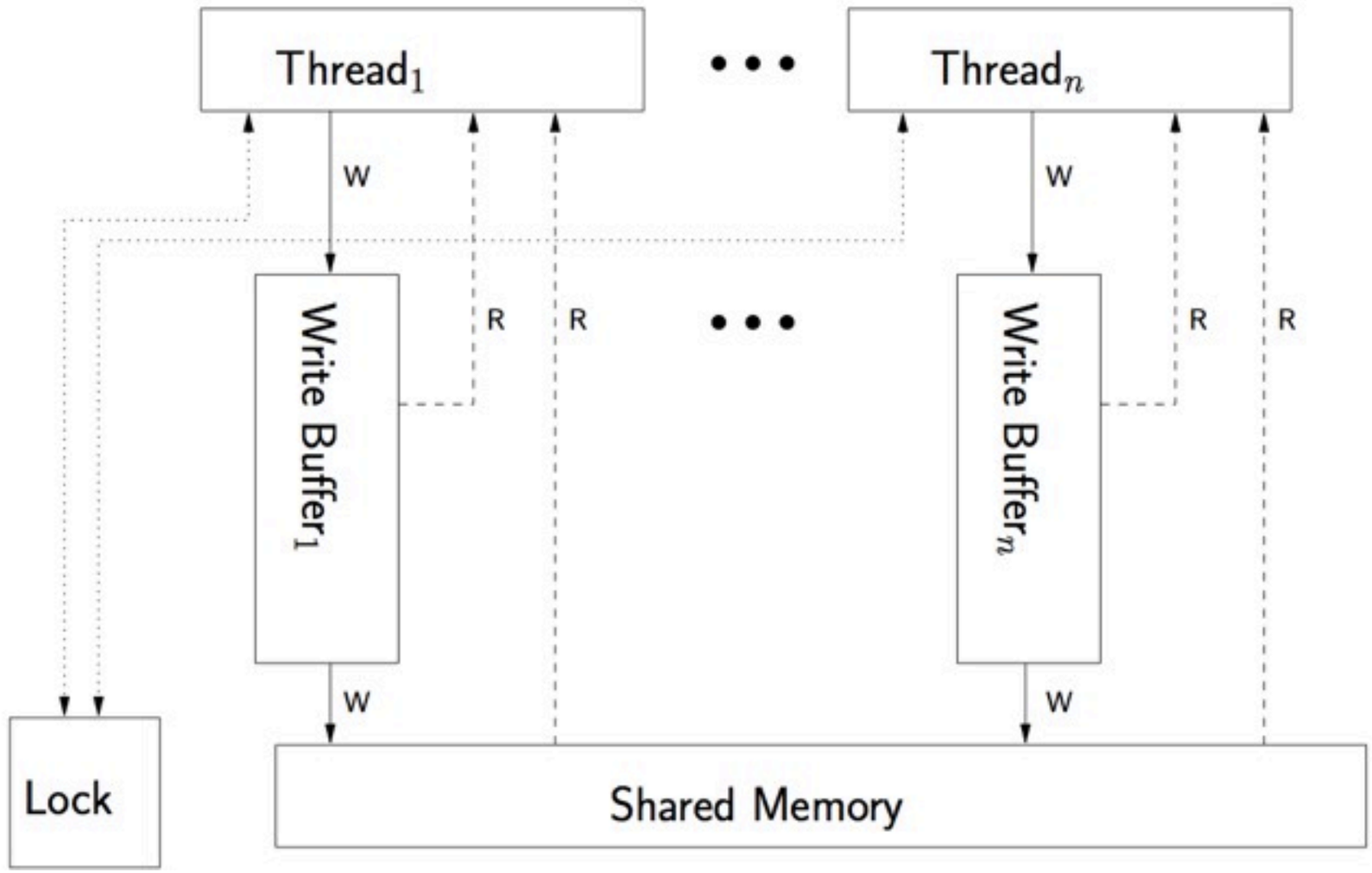
as well

The po

- EAX
- EAX
- EAX

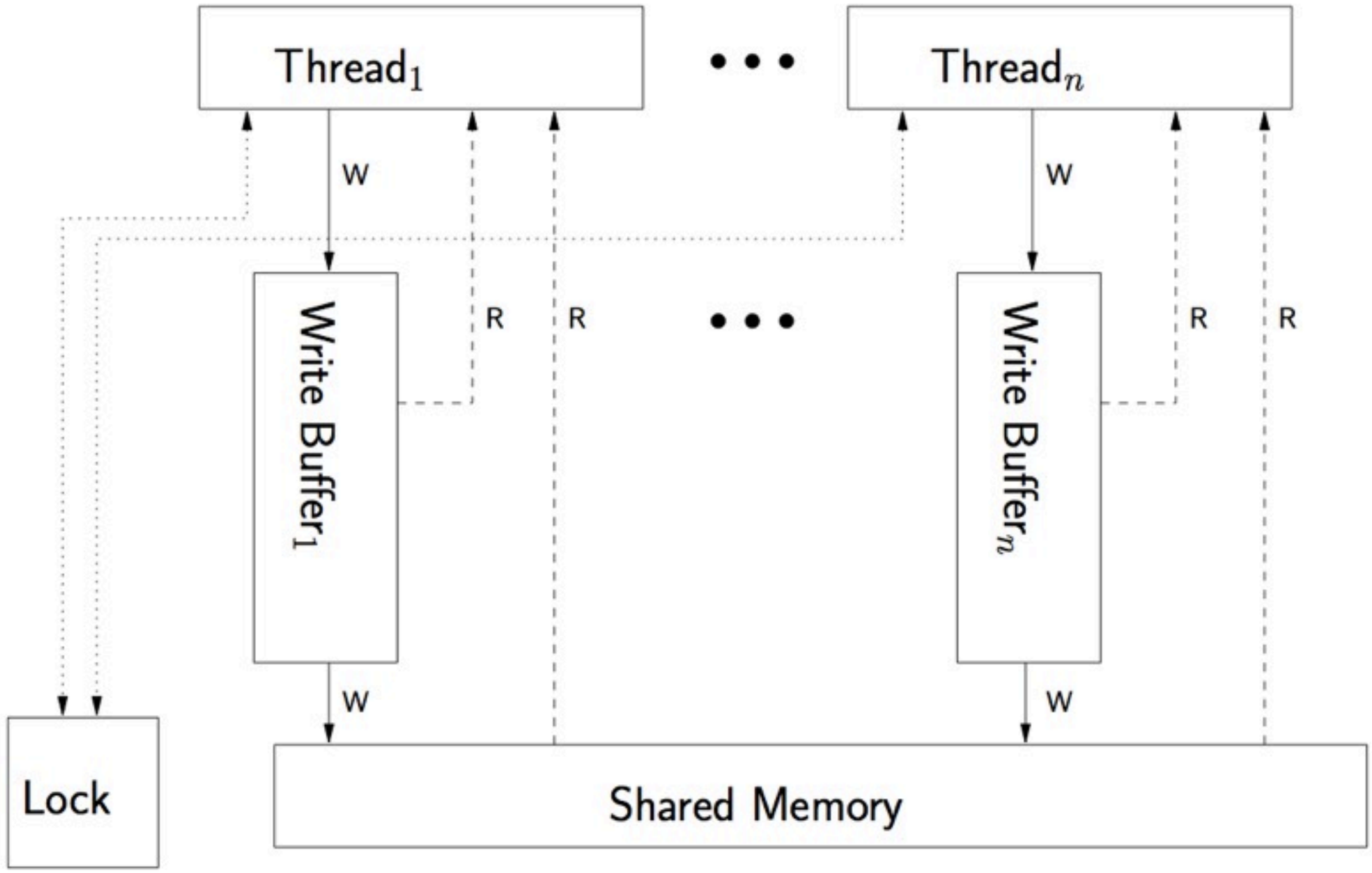
Hardware store buffering

Store buffers hide the latency of memory writes



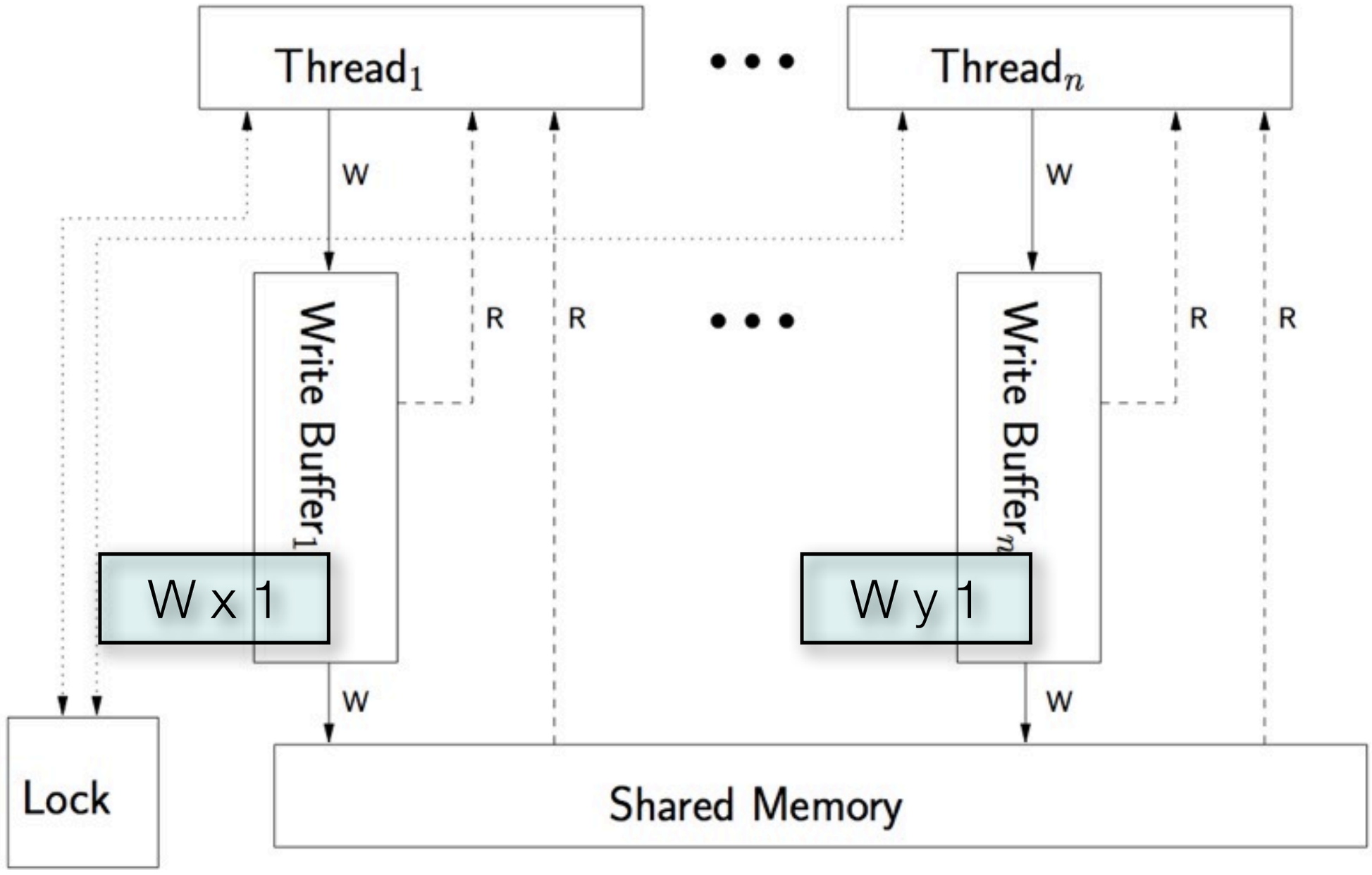
Hardware store buffering

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`



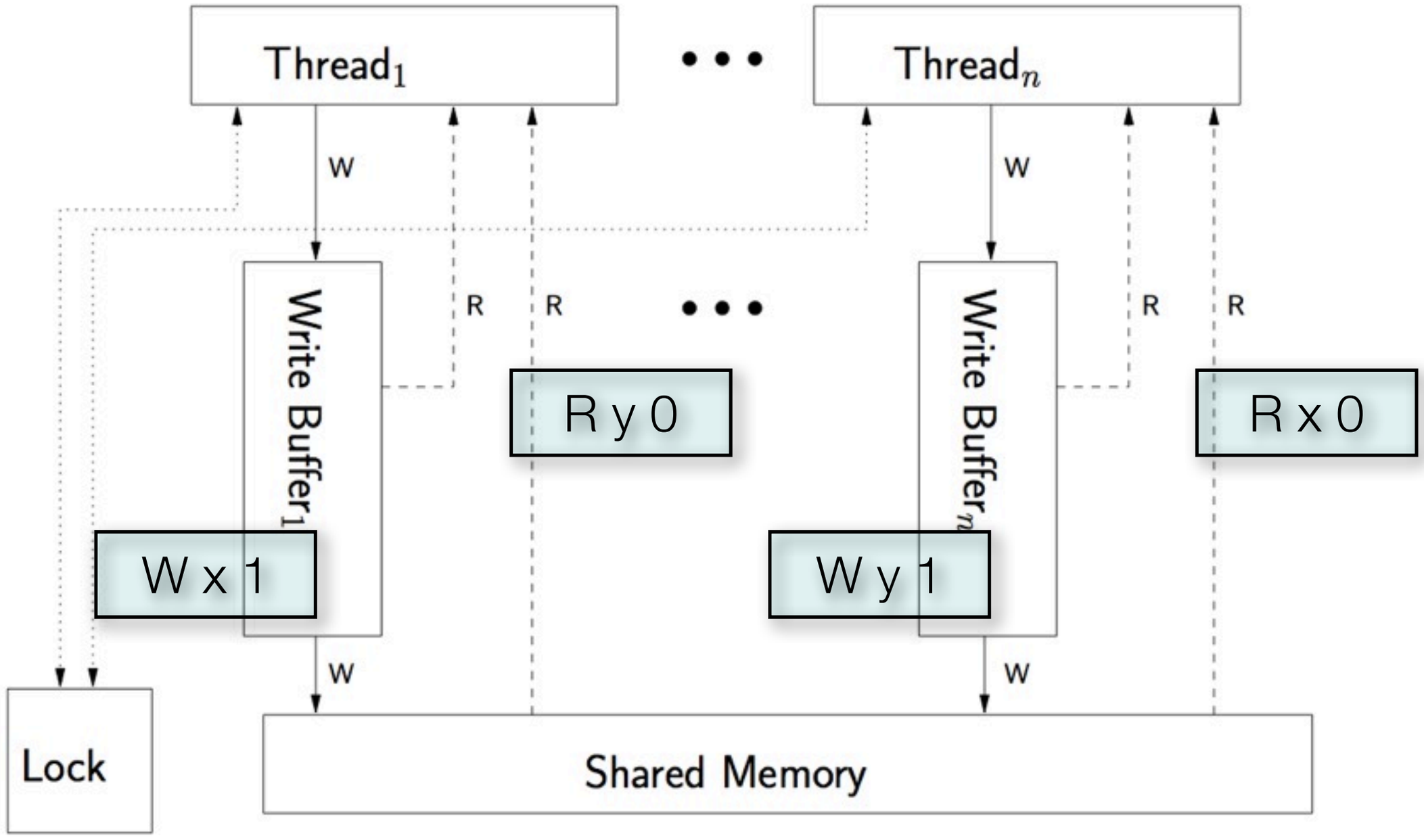
Hardware store buffering

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`



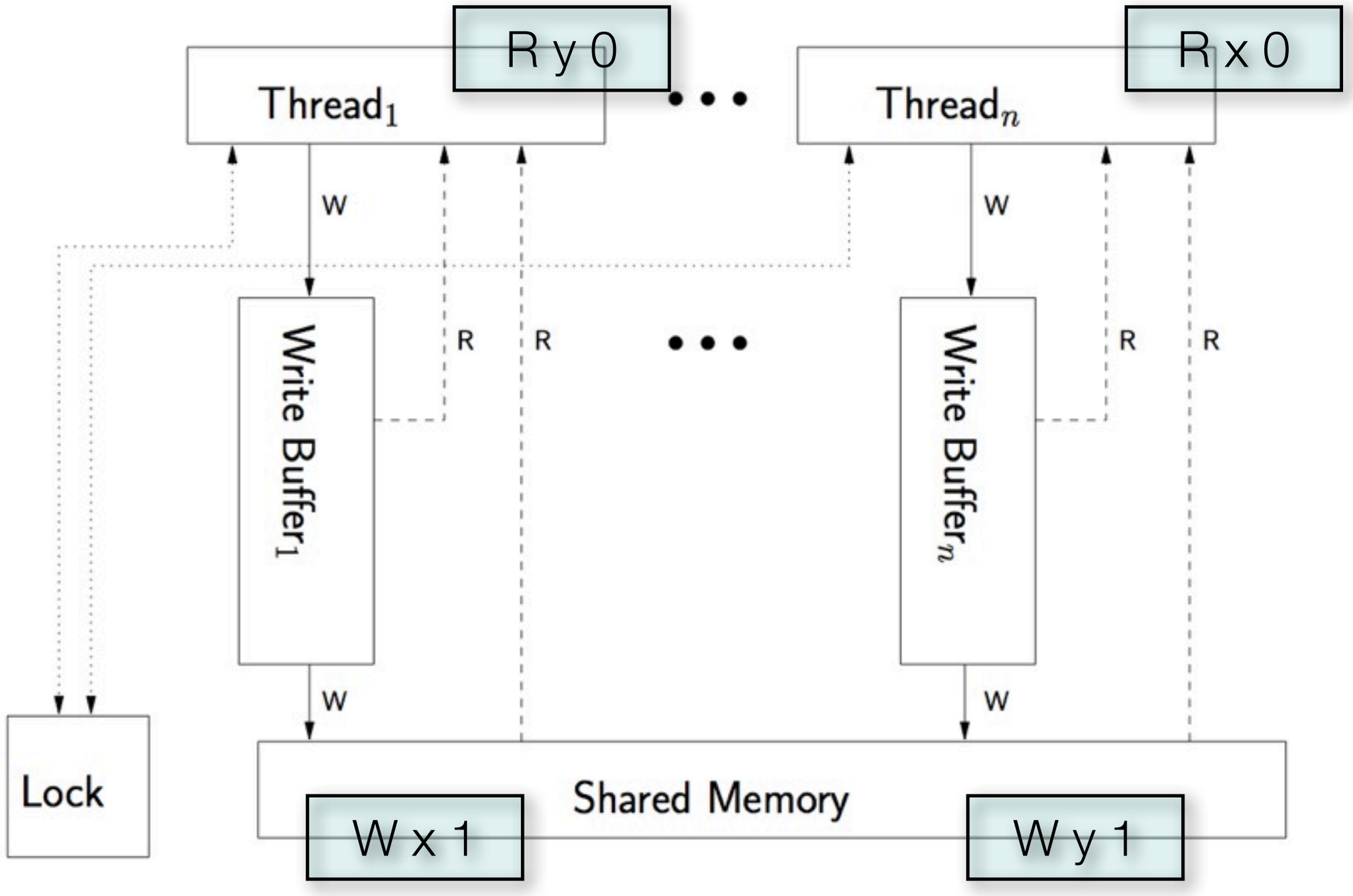
Hardware store buffering

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`



Hardware store buffering

Thread 0	Thread 1
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV EAX ← [y]`	`MOV EBX ← [x]`



That pesky hardware (3)

...and differ between architectures...

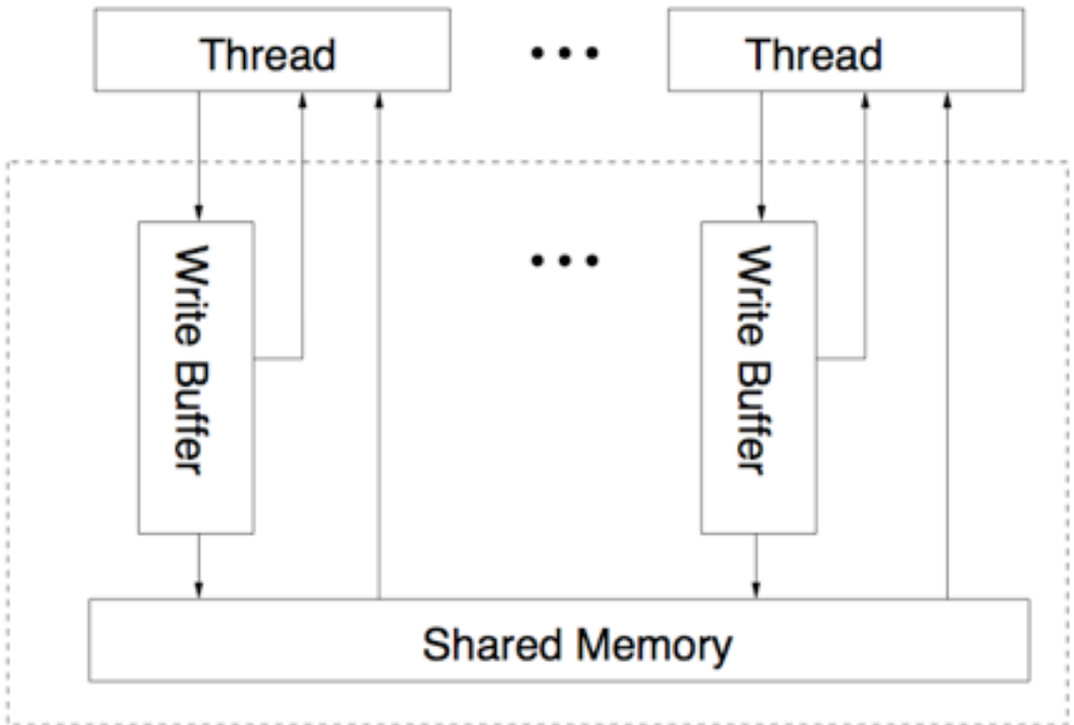
Thread 0	Thread 1
`x = 1`	`print y`
`y = 1`	`print x`

On x86, we only get

0 0

1 1

is printed on the screen.



That pesky hardware (3)

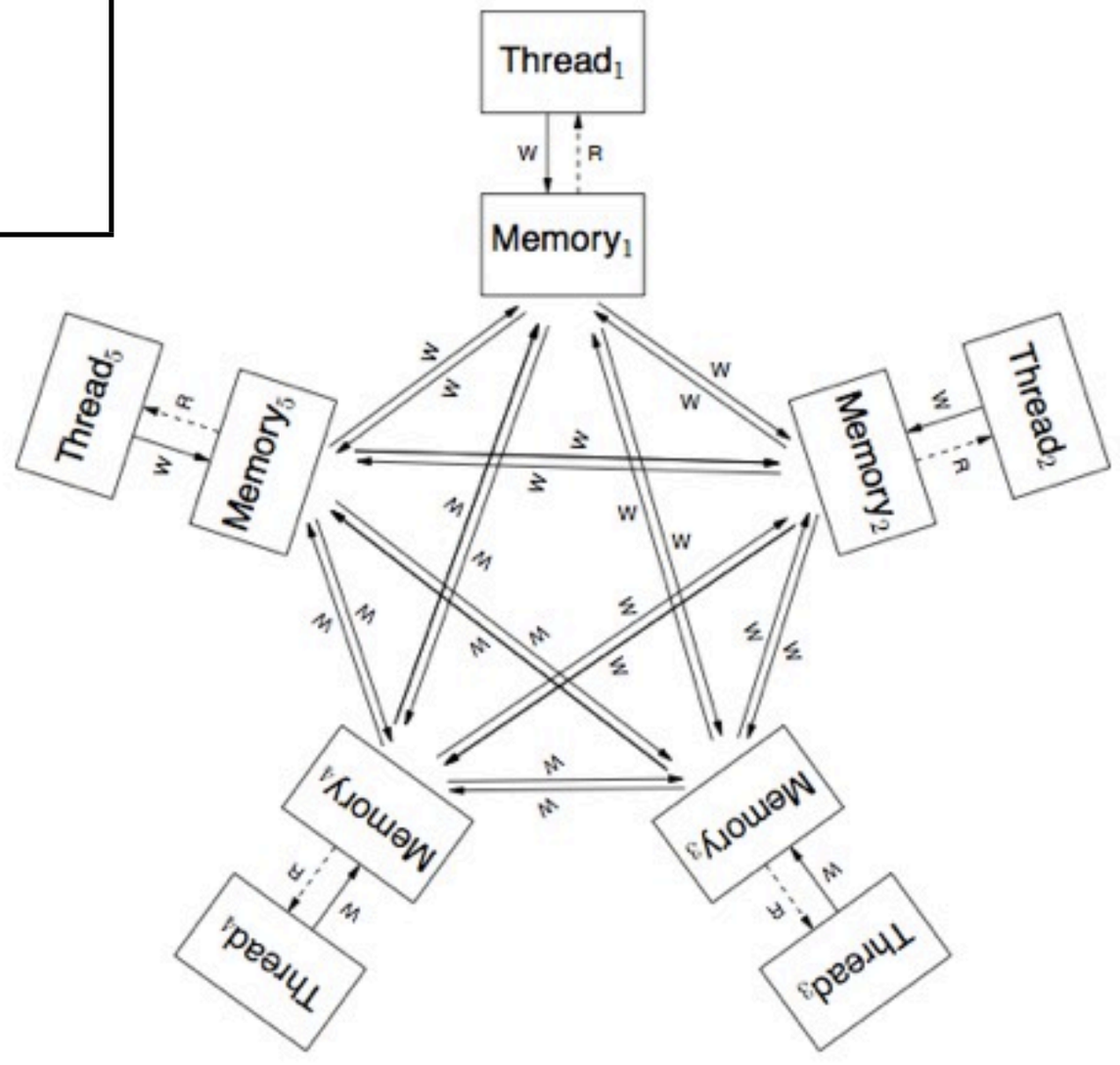
...and differ between architectures...

Thread 0	Thread 1
`x = 1`	`print y`
`y = 1`	`print x`

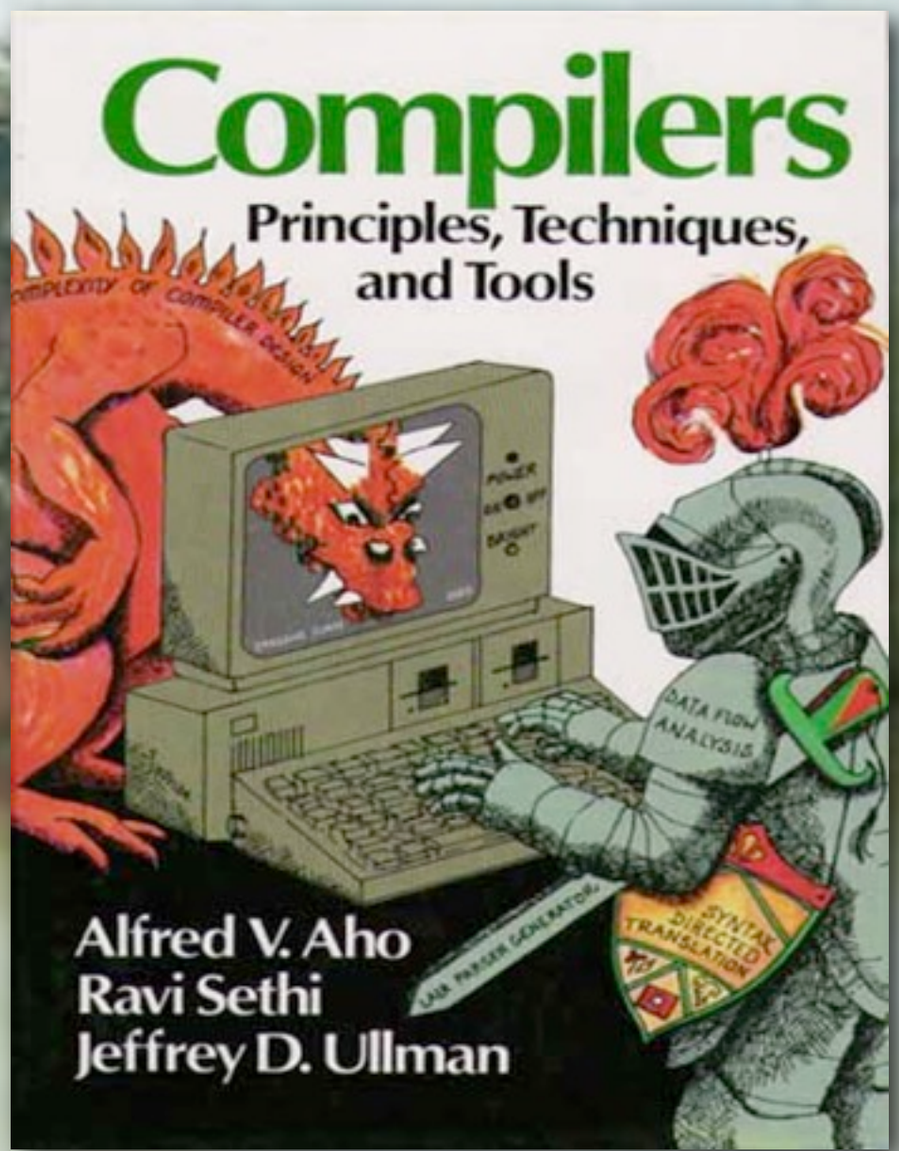
On IBM Power or ARM

1 0

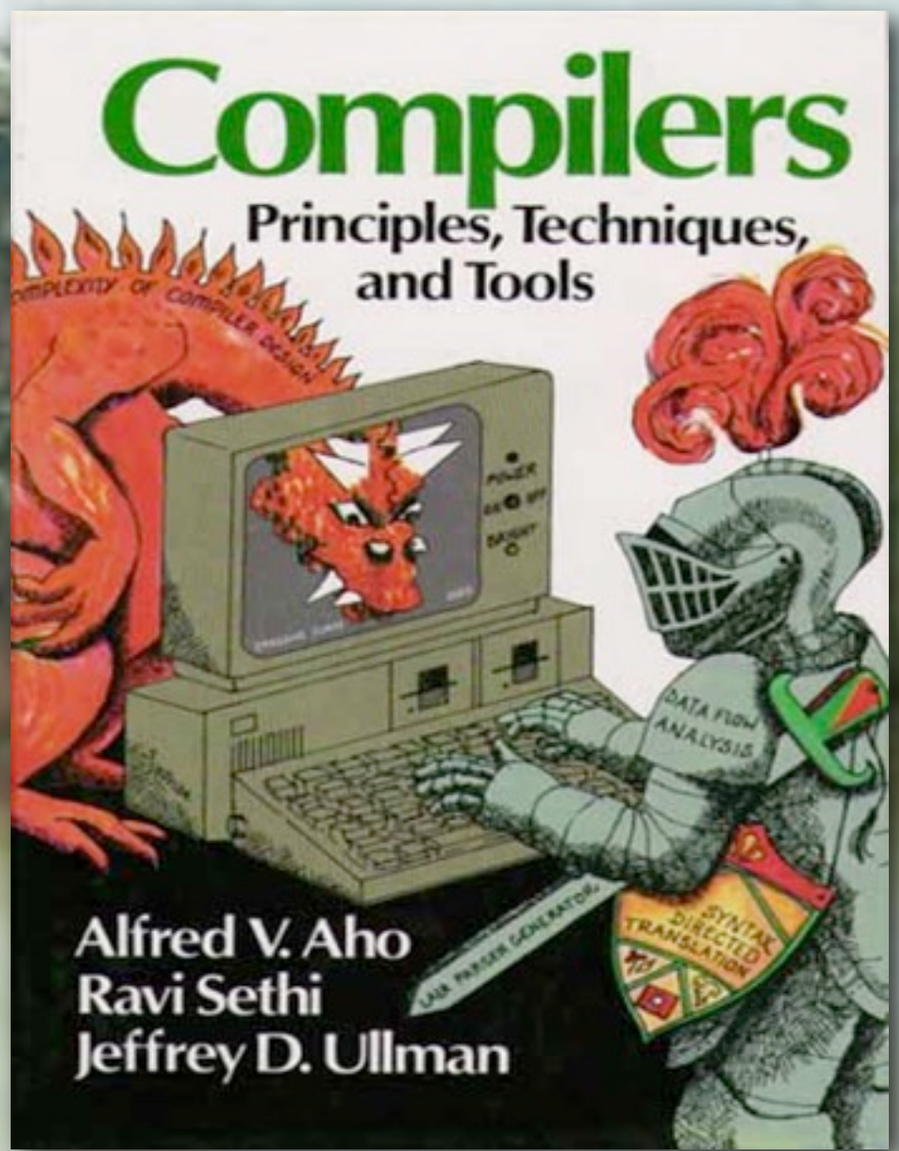
can be printed on the screen.



The fundamental problem

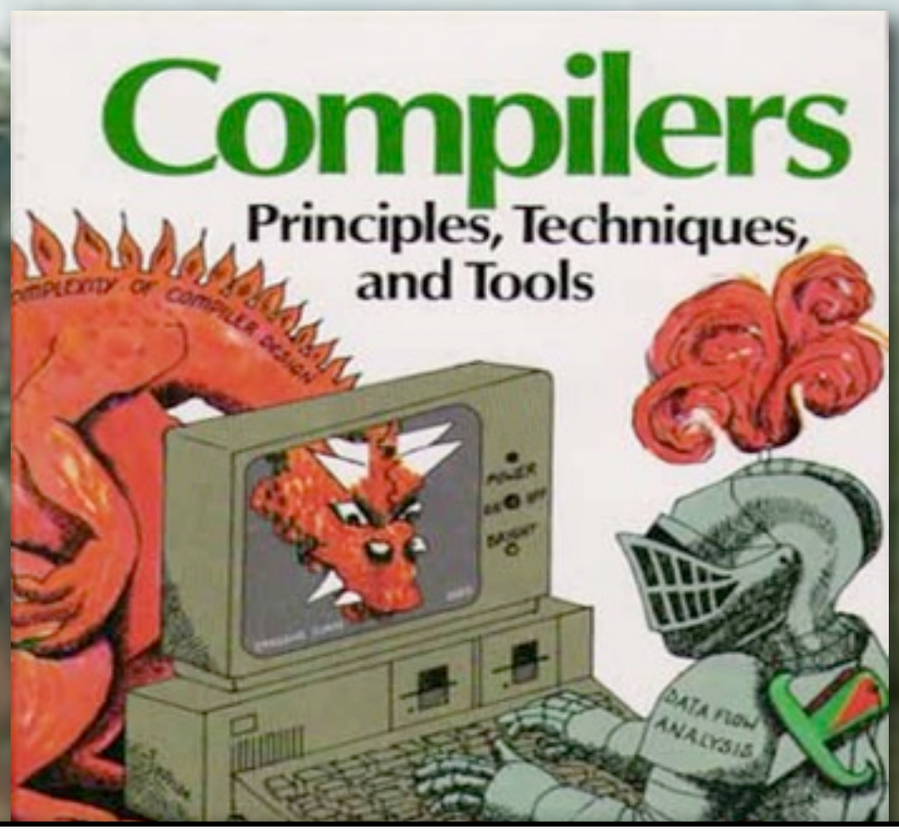


The fundamental problem



The programmer wants to understand the code he writes

The fundamental problem



The programmer wants to understand the code he writes

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

The fundamental problem

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

Which is the semantics of a concurrent program?

The programmer wants to understand the code he writes

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

Not new

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

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

Eclipsed for a long time (except in high-end) by advances in performance:

- transistor counts (continuing)
- clock speed (hit power dissipation limit)
- ILP (hit smartness limit?)

Mass market multiprocessors since 2005

Intel Xeon E7
up to 20 hardware threads

IBM Power 795 server
up to 1024 hardware threads

Best quad core phone: 4 contenders examined

EARLY VIEW HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad

Mass market multiprocessors since 2005

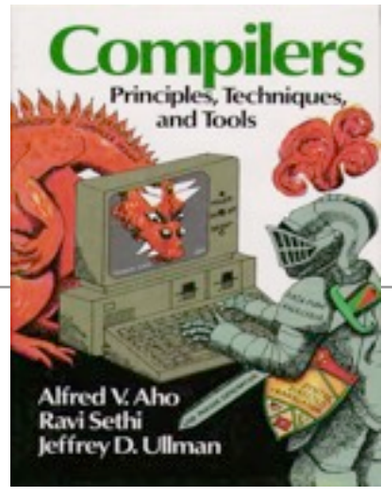
Intel Xeon E7
up to 20 hardware threads

Programming multiprocessors
no longer just for specialists

Best quad core phone: 4 contenders examined

EARLY VIEW HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad

Topics

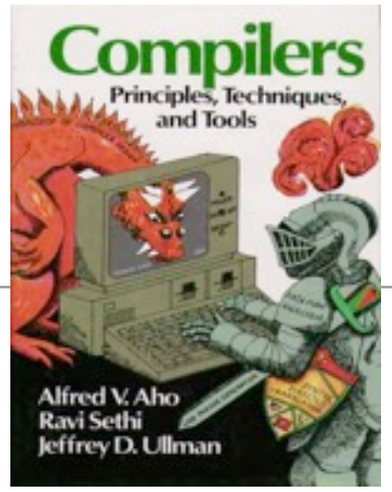


1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation

Topics



1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation

Architectures

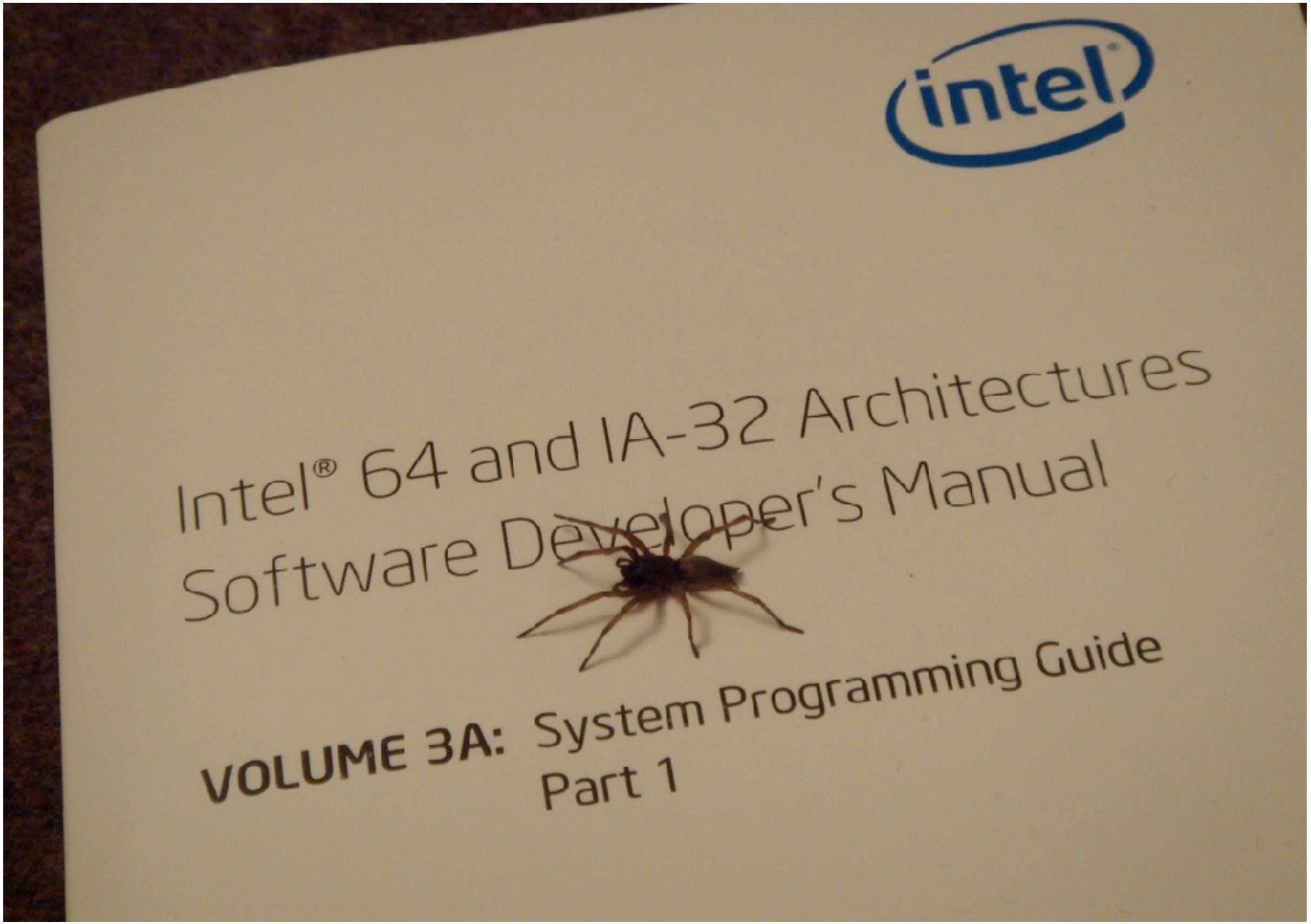
Hardware manufacturers document **architectures**:

- *loose* specifications
- claimed to cover a *wide range* of past and future processor implementations.

Architectures should:

- *reveal enough* for effective programming;
- without *unduly constraining* future processor design.

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


intel
Intel® 64 and IA-32 Architectures
Software Developer's Manual
VOLUME 3A: System Programming Guide
Part 1

In practice

Architectures described by informal prose:

> *In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.*
>
> (Intel SDM, november 2006, vol3a, 10-5)

As we shall see, such descriptions are:

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

Fundamental problem: prose specifications cannot be used to *test programs* or to *test processor implementations*.

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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

The
sure

For
hap
stal
spin

spin

spin

a =
/* ca

b ge

a =
and

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.

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 [illegible]

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.

```
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 [illegible]

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..

Example: Linux kernel mailing list, 20 nov. - 7 déc. 1999 (143 posts).

A one-instruction programming question, a microarchitecural debate!

Keywords: speculation, ordering, causality, retire, cache...

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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.

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.

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.. */

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..

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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.. */

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.

spin_unlock();

spin_lock();

a = 1;
/* cache miss satisfied, the "a" line is boun

b gets the value 1

a = 0;
and it re

Unlikely
bug in a **ial**

Manfred
Manual,
the Penti
writes in
around b
(cache n
CPU wo
a Pentiu
manual,
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.

this was no longer the case, though the window is so infinitesimally small most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is writtne to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the 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..

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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.

spin_unlock();

spin_lock();

a = 1;
/* cache miss satisfied, the "a" line is boun

b gets the value 1

a = 0;
and it re

Unlikely ... ial
bug in a

Manfred ... *Manual*, ... ce, the Penti... writes in... around b... (cache m... nd CPU wo... t on a Pentiu... o manual, ... ort for speculative reads and store-buffer forwarding." He explained:

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

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

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

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache 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

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.

```
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 ... end up doing is equivalent to

, we'll delay this.. */

is that stores can only possibly be observed when all prior ...etired (i.e. the store is not sent outside of the processor ...d state, and the earlier instructions are already committed ...mpleted first, cache-m...

He went on:

Since the instructi... een externally observe... ly functioning spinlo... b" to the value of "a" have t...

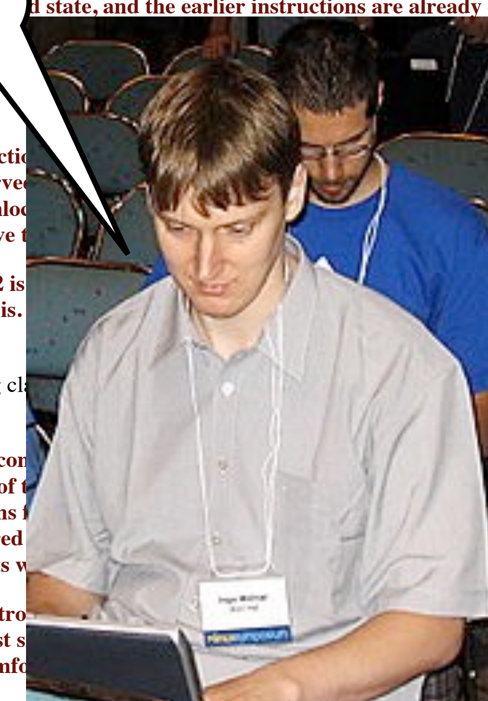
In general, IA32 is ... ation doesn't affect this. ... er processors.

There was a long cl... und by Linus:

Everybody has co... strong enough that all of t... otten sane explanations t... e locked access) is required ... hat lack of symmetry was w...

Oliver made a stro... ely explained by just s... ds vs writes. I feel comf...

Thanks, guys, we'l...



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

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

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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
stale value for any of the re
spinlock.

spin_unlock();

spin_lock();

a = 1;
/* cache miss satisfied, the

b gets the value 1

a = 0;
and it re

Unlikely
bug in a

Manfred
Manual,
the Penti
writes in
around b
(cache m
CPU wo
a Pentiu
manual,
for speculative reads and stor

A Pentium is a in-order ma
wrt reads etc. So on a Pentium you'll never see the problem.

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

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

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache 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.

It does NOT WORK!

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

g the optimization very valuable.
ame optimization cropped up in
the FreeBSD mailing list.
ck for a simple "movl
uction, a huge gain.

n't hit it (Netware 4/5 uses this method but it's spinlocks
d this and the code is writtne to handle it. The most obvious
behavior was that cache inconsistencies would occur randomly.
lock to signal that the piplines are no longer invalid and the buffers
blown out.

n the behavior Linus describes on a hardware analyzer, BUT
N SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD
st still be on older Pentium hardware and that's why they don't
can bite in some cases.

yn, an Architect in an IA32 development group at Intel, also replied to
nting out a possible misconception in his proposed exploit. Regarding
inus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

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

```
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?

turn 1 with the proposed optimization. I doubt you
but hey, add another access to another variable
accessed through another spinlock (to get cache-
ects), and I suspect you can make it happen even
e above.

ite legally is that your new "spin_unlock()" isnot
is very little effective ordering between the two

erent data (ie no data dependencies in sight). So
is equivalent to

y this.. */

is that stores can only possibly be observed when all prior
tired (i.e. the store is not sent outside of the processor
d state, and the earlier instructions are already committed
mpleted

He went on:

Since the instructi
externally observe
functioning spinloc
value of "a" have t

In general, IA32 is
doesn't affect this.
processors.

There was a long cl
Linus:

Everybody has co
enough that all of t
sane explanations
access) is required
of symmetry was w

Oliver made a stro
explained by just s
writes. I feel comfo

Thanks, guys, we'll

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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!

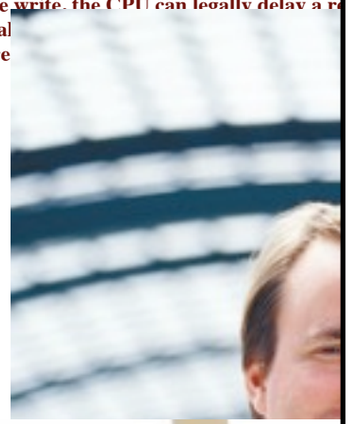
Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

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
sure that the processor doesn't re-order things around the unlo

For example, with a simple write, the CPU can legally delay a re
happened inside the critical
stale value for any of the re
spinlock.



spin_unlock();

A Pentium is a in-order ma
wrt reads etc. So on a Pentium you'll never see the problem.

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

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

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache 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.

It does NOT WORK!

Let the FreBSD people use it and

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."

n't hit it (Netware 4/5 uses this method but it's spinlocks
d this and the code is writtne to handle it. The most obvious
behavior was that cache inconsistencies would occur randomly.
lock to signal that the piplines are no longer invalid and the buffers
blown out.

n the behavior Linus describes on a hardware analyzer, BUT
N SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD
st still be on older Pentium hardware and that's why they don't
can bite in some cases.

yn, an Architect in an IA32 development group at Intel, also replied to
nting out a possible misconception in his proposed exploit. Regarding
inus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

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

```
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?

turn 1 with the proposed optimization. I doubt you
but hey, add another access to another variable
accessed through another spinlock (to get cache-
ects), and I suspect you can make it happen even
e above.

ite legally is that your new "spin_unlock()" isnot
is very little effective ordering between the two

So

ior
ted
d

functioning spinloc **b" to the**
value of "a" have t

In general, IA32 is **ation**
doesn't affect this. **er**
processors.

There was a long cl **und by**
Linus:

Everybody has co **strong**
enough that all of t **otten**
sane explanations f **e locked**
access) is required **hat lack**
of symmetry was w

Oliver made a stro **ely**
explained by just s **ds vs**
writes. I feel comfo

Thanks, guys, we'l

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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
warn you NOT to assume
case in the long run).

The issue is that you _have
sure that the processor do

For example, with a simpl
happened inside the critic
stale value for any of the r
spinlock.

spin_unlock();

s

a
/*

b

a
a

U
b

M
M
th
w
a
(
C
a
manual,
for speculative reads and stor

A Pentium is a in-order ma
wrt reads etc. So on a Pentium you'll never see the problem.

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

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

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache 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.

n't hit it (Netware 4/5 uses this method but it's spinlocks
d this and the code is writtne to handle it. The most obvious
behavior was that cache inconsistencies would occur randomly.
lock to signal that the piplines are no longer invalid and the buffers
blown out.

n the behavior Linus describes on a hardware analyzer, BUT
N SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD
st still be on older Pentium hardware and that's why they don't
can bite in some cases.

yn, an Architect in an IA32 development group at Intel, also replied to
nting out a possible misconception in his proposed exploit. Regarding
inus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

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

```
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?

turn 1 with the proposed optimization. I doubt you
but hey, add another access to another variable
accessed through another spinlock (to get cache-
ects), and I suspect you can make it happen even
e above.

ite legally is that your new "spin_unlock()" isnot
is very little effective ordering between the two

So

ior
ted
d

b" to the

ation
er

und by

strong
otten
e locked
hat lack

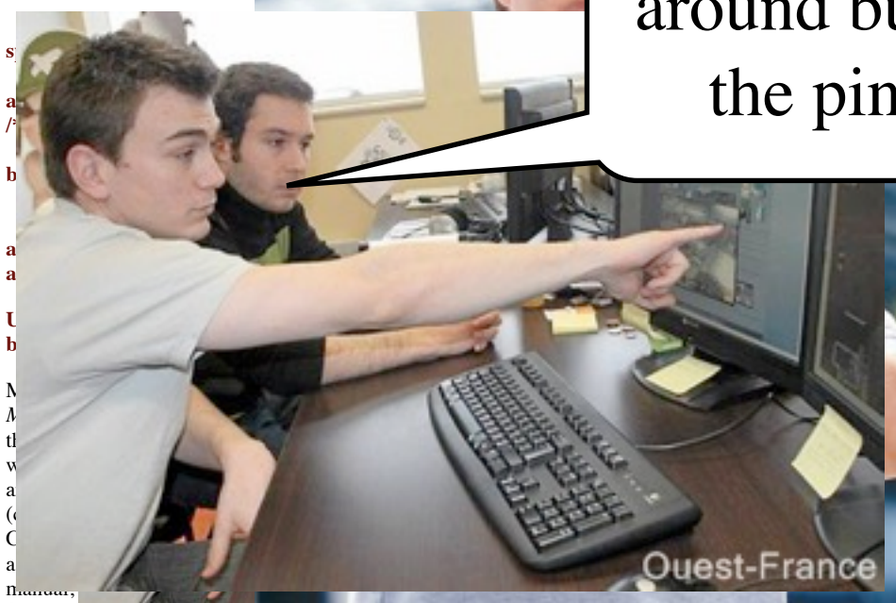
ely
ds vs

It does NOT WORK!

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

around buffered writes. Memory re
the pins, reads (cache miss) and

ssor Family Developers
ory Access Ordering, "to



1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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
warn you NOT to assume
case in the long run).

The issue is that you _have
sure that the processor do

For example, with a simpl
happened inside the critic
stale value for any of the r
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)

```
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
lse, the spinlock would

imization. I doubt you
s to another variable
pinlock (to get cache-
make it happen even

"spin_unlock()" isnot
ring between the two

So

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.

From the Pe
only enhanc
processor i
speculative

writes. Memory re
s (cache miss) and

are 4/5 uses this method but it's spinlocks
code is writtne to handle it. The most obvious
hat cache inconsistencies would occur randomly.
hat the piplines are no longer invalid and the buffers

Linus describes on a hardware analyzer, BUT
HAT WERE PPRO AND ABOVE. I guess the BSD
der Pentium hardware and that's why they don't
e cases.

t in an IA32 development group at Intel, also replied to
ble misconception in his proposed exploit. Regarding
inus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

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

A Pentium is a in-order ma
wrt reads etc. So on a Pentium you'll never see the problem.

ior
ted
d

b" to the
ation
er
und by
strong
otten
e locked
hat lack
ely
ds vs

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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
eventually.

The window may be sm
reliable any more.

The issue is not writes
warn you NOT to assu
case in the long run).

The issue is that you _h
sure that the processor

For example, with a sin
happened inside the cri
stale value for any of 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)

```
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
lse, the spinlock would

imization. I doubt you
s to another variable
spinlock (to get cache-
make it happen even

"spin_unlock()" isnot
ring between the two

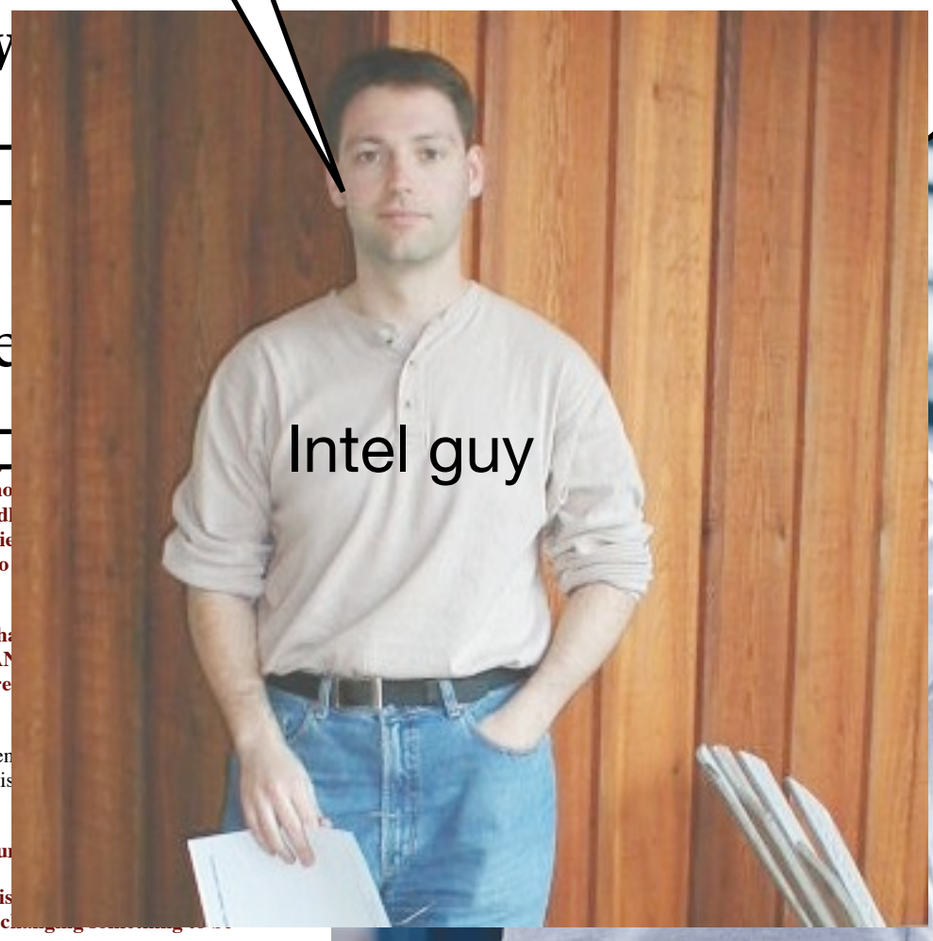
So

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

processor
speculative

on older Pentium hard
they don't know

Linus describes on a
BUT ONLY ON
ERE PPRO AND
D people must still be
ware and that's why



writes.
s (cache

are 4/5 uses this metho
code is writtne to handl
hat cache inconsistencie
hat the piplines are no

Linus describes on a ha
HAT WERE PPRO AN
der Pentium hardware
e cases.

t in an IA32 developmen
ble misconception in his
inus posted, Erich replied:

ays return 0. You don't need "spin_u

hing you need is to make sure there is
s kind of true by the fact that you're c
observable on other processors.

for speculative reads and stor

A Pentium is a in-order ma
wrt reads etc. So on a Pentium you'll never see the problem.

ior
ted
d

b" to the
ation
er

und by

strong
otten
e locked
hat lack

ely
ds vs

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (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
eventually.

The window may be sm
reliable any more.

The issue is not writes
warn you NOT to assu
case in the long run).

The issue is that you _h
sure that the processor

For example, with a sin
happened inside the cri
stale value f
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)

```
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
lse, the spinlock would

imization. I doubt you
s to another variable
pinlock (to get cache-
make it happen even

"spin_unlock()" isnot
ring between the two

So

ior
ted
d

b" to the

ation
er

und by

strong
otten
locked
hat lack

ely
ds vs

Linus describes on a ha
HAT WERE PPRO AN
der Pentium hardware
e cases.

ays return 0. You don't need "spin_u

hing you need is to make sure there is
s kind of true by the fact that you're c
observable on other processors.

Power ISA 2.06 and ARM v7

Key concept: actions being performed.

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

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

Power ISA 2.06 and ARM v7

Key concept: actions being performed.

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

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

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

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

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

Power ISA

Key concept: act

> *A load by a* *ct to any*
> *processor (P* *e load can*
> *no longer be*

Used to compute *ntics of barriers.*

> *The definition o* *ore by P2.*

A memory mod *n is allowed.*

It is is awkward *ntifies over all*
hypothetical var

See Alglave et al., PLDI, 2011.

A way out?

Way out? Create *rigorous* memory models

- Unambiguous

- Sound w.r.t. experience

- Consistent with what we know of vendor intentions

Way out? Create *rigorous* memory models

- Unambiguous

  mathematical language

- Sound w.r.t. experience

- Consistent with what we know of vendor intentions

Way out? Create *rigorous* memory models

- Unambiguous

  mathematical language

- Sound w.r.t. experience

  rigourous testing of the model against the hardware

- Consistent with what we know of vendor intentions

Way out? Create *rigorous* memory models

- Unambiguous

  mathematical language

- Sound w.r.t. experience

  rigourous testing of the model against the hardware

- Consistent with what we know of vendor intentions

  interaction with hardware developers

Mathematical language

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

Want reusable specifications!

LEM: a DSL for discrete-math definitions

You write:

- definitions of types, functions, inductive relations
- with quantifiers, *set comprehensions*, and top-level type polymorphism

(roughly intersection of HOL4, Isabelle/HOL, and Coq)

LEM gives you:

- type-checking of the definitions
- decent typesetting
- *whitespace-preserved* prover definitions in HOL4, Isabelle/HOL (&Coq?)
- OCaml code (ind.rel.?) (Haskell?)

LEM: a DSL for discrete-math definitions

Example taken form the IBM POWER memory model

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

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

- OCaml code (ind.ref.?) (Haskell?)

The ARM / IBM POWER memory model formalisation

Executing the specifications

Make the model accessible to programmers

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

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

DEMO [ppcmem]

Testing the specifications

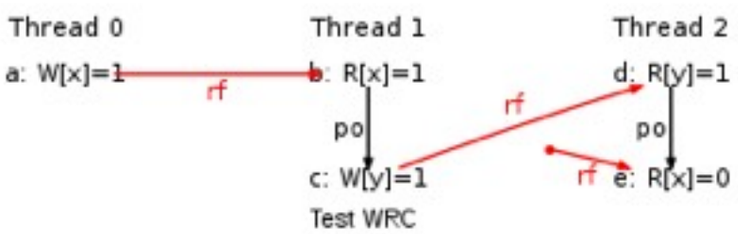
1. Systematically generate litmus tests out of the spec

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

WRC: Write to Read Causality

Test WRC

Run WRC in model, using ppcmem




```
PPC WRC
"Rfe PodRW Rfe PodRR Fre"
Cycle=Rfe PodRW Rfe PodRR Fre
{
0:r2=x;
1:r2=x; 1:r4=y;
2:r2=y; 2:r4=x;
}
 P0            | P1              | P2
 li r1,1       | lwz r1,0(r2)    | lwz r1,0(r2)
 stw r1,0(r2)  | li r3,1         | lwz r3,0(r4)
               | stw r3,0(r4)    |
exists
(1:r1=1 /\ 2:r1=1 /\ 2:r3=0)
```

		Model	PowerG5	Power6	Power7
WRC	Allow	=	Ok, 44k/2.5G	Ok, 1.2M/13G	Ok, 25M/104G
WRC+data+addr	Allow	=	No, 0/3.3G	Ok, 705k/13G	Ok, 166k/105G
			Allow unseen		
WRC+syncs	Forbid	=	Ok, 0/3.3G	Ok, 0/17G	Ok, 0/157G
WRC+sync+addr	Forbid	=	Ok, 0/3.3G	Ok, 0/17G	Ok, 0/157G
WRC+lwsync+addr	Forbid	=	Ok, 0/3.3G	Ok, 0/17G	Ok, 0/137G
WRC+data+sync	Allow	=	No, 0/3.3G	Ok, 176k/13G	Ok, 75k/105G
			Allow unseen		
WRC+addr+ctrl	Allow	=	Ok, 43k/1.3G	Ok, 313k/4.3G	Ok, 4.5M/24G
WRC+addr+ctrlisync	Allow	=	No, 0/2.1G	Ok, 402k/4.3G	Ok, 69k/25G
			Allow unseen		
WRC+addr+isync	Allow	=	No, 0/2.1G	Ok, 403k/4.3G	Ok, 49k/25G
			Allow unseen		

Testing the specifications

1. Systematically generate litmus tests out of the spec

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

Rigourous *testing* and *interaction with hardware architects* to validate the formalisation of the memory models

WRC: Write to Read Causality

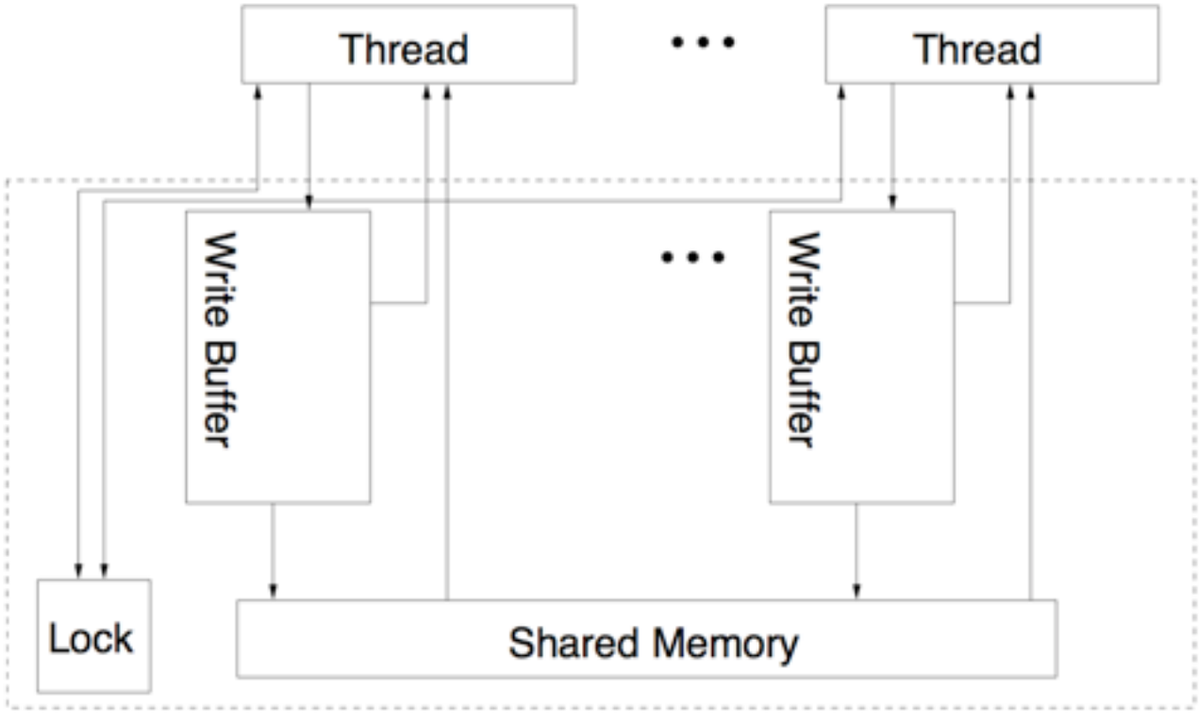
P0	P1	P2
li r1,1	lwz r1,0(r2)	lwz r1,0(r2)
stw r1,0(r2)	li r3,1	lwz r3,0(r4)
	stw r3,0(r4)	

exists
(1:r1=1 /\ 2:r1=1 /\ 2:r3=0)

WRC+addr+ctrlisync	Allow	=	No, 0/2.1G	Ok, 402k/4.3G	Ok, 69k/25G
			Allow unseen		
WRC+addr+isync	Allow	=	No, 0/2.1G	Ok, 403k/4.3G	Ok, 49k/25G
			Allow unseen		

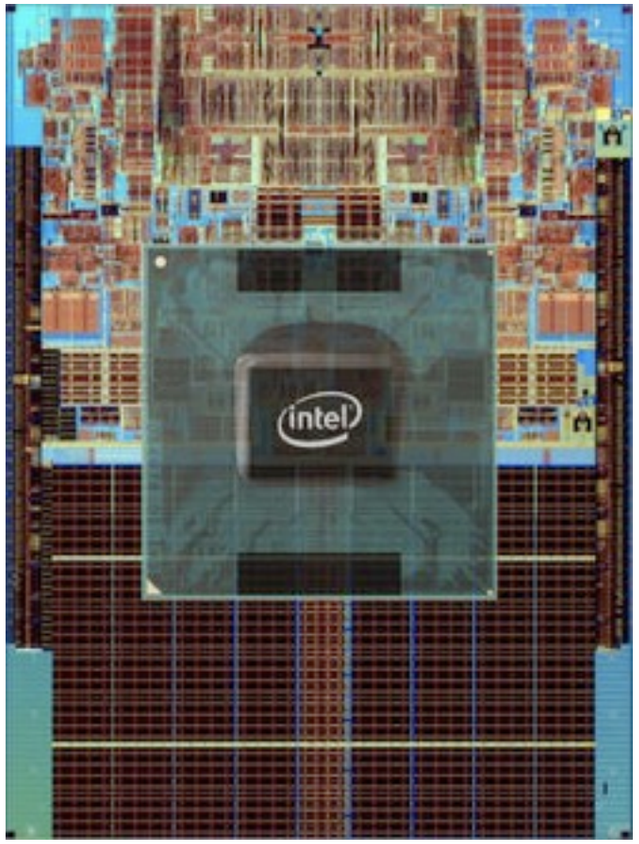
These are abstract machines

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

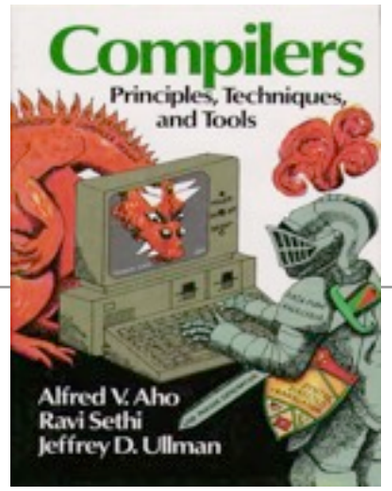


$\supseteq_{beh}$

$\neq_{hw}$



Topics

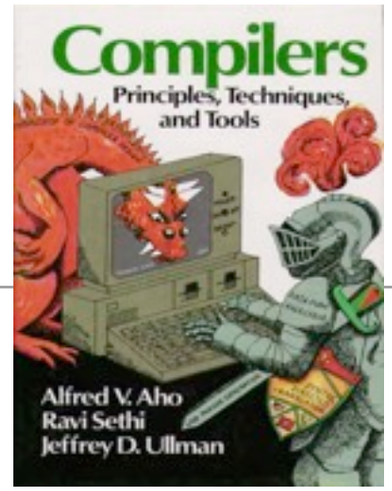


1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation

Topics



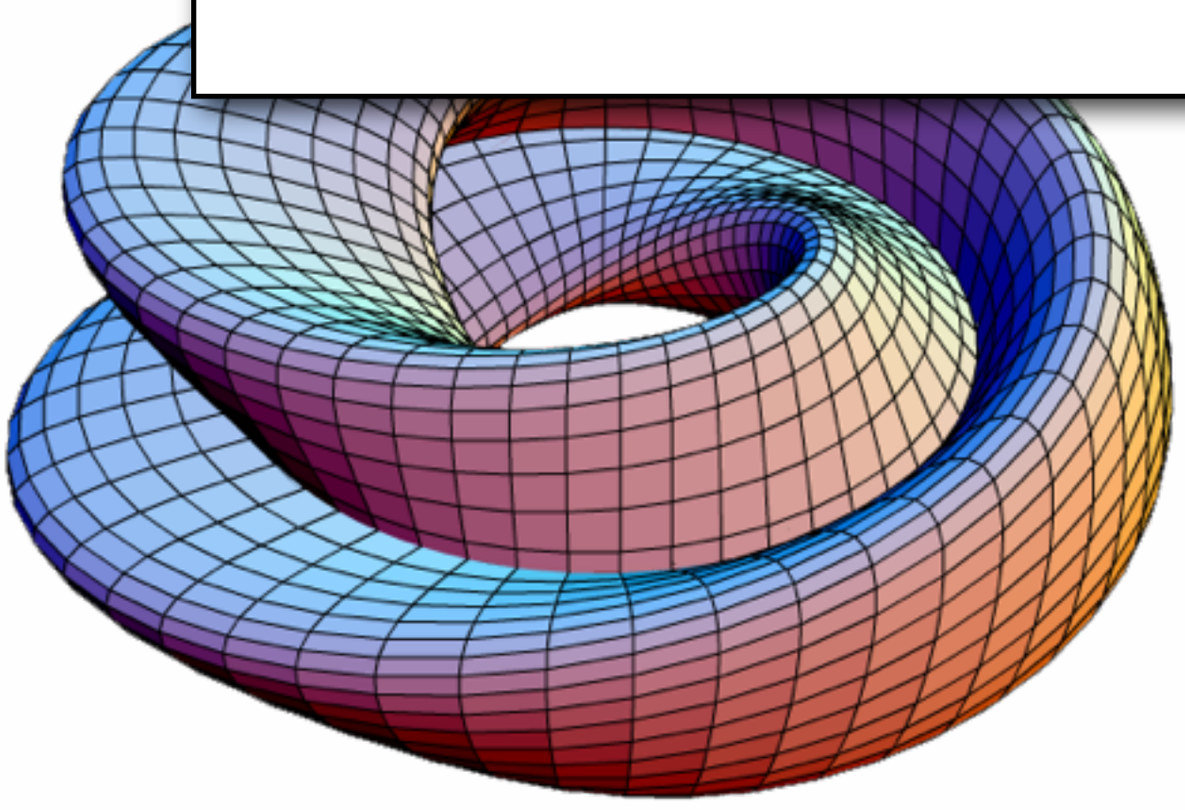
1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compiler and optimisations: proof and/or validation

The simplest memory model

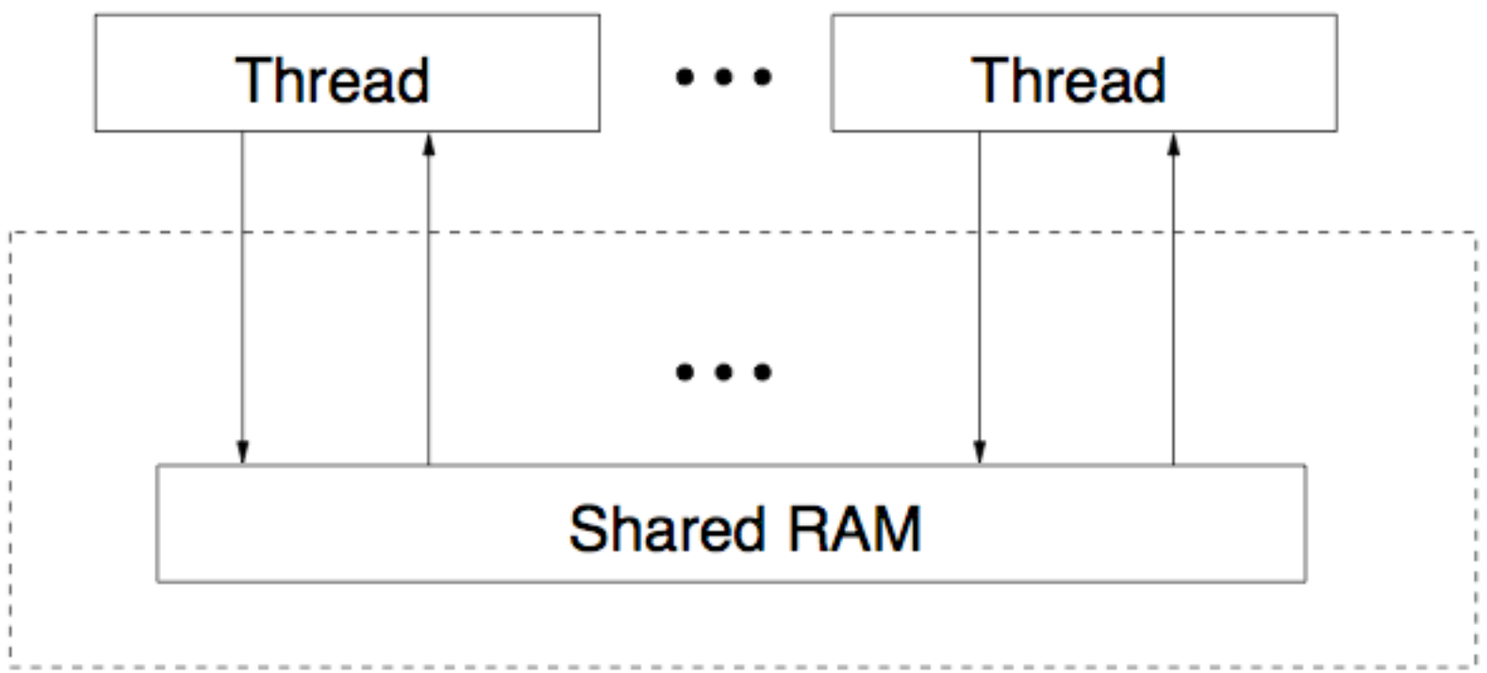
sequential consistency

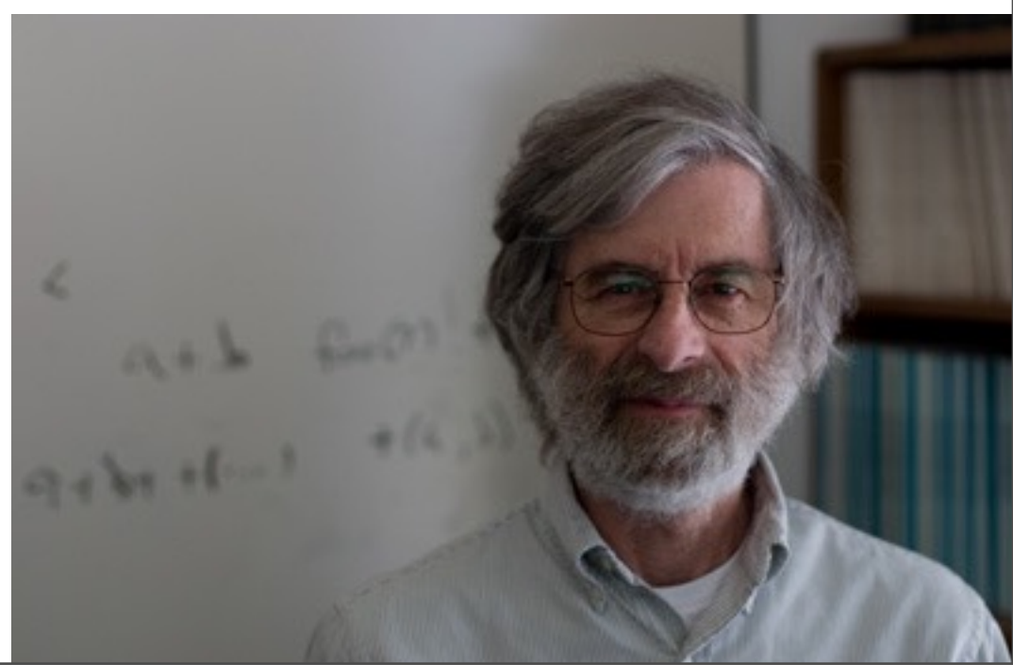


Sequential consistency

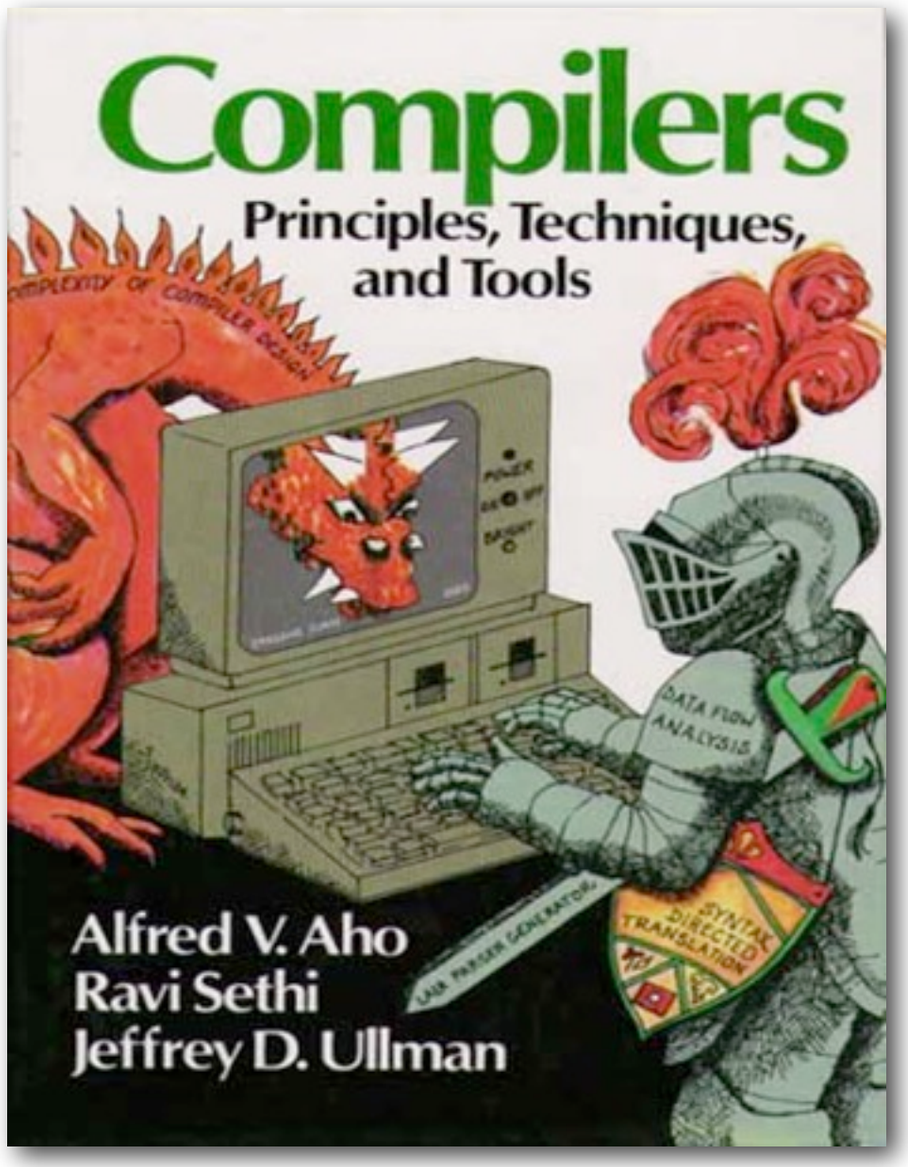
...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...

Lamport, 1979.

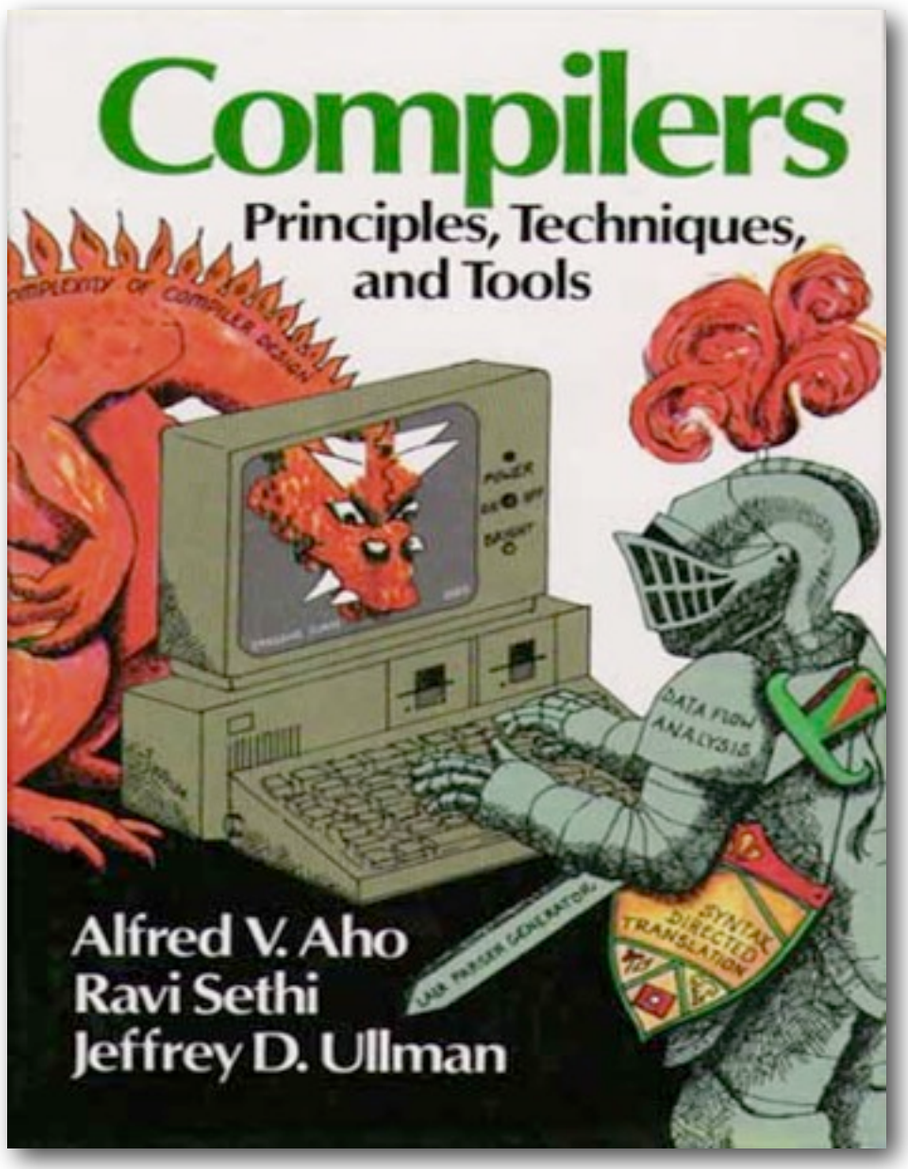




Compilers, programmers & sequential

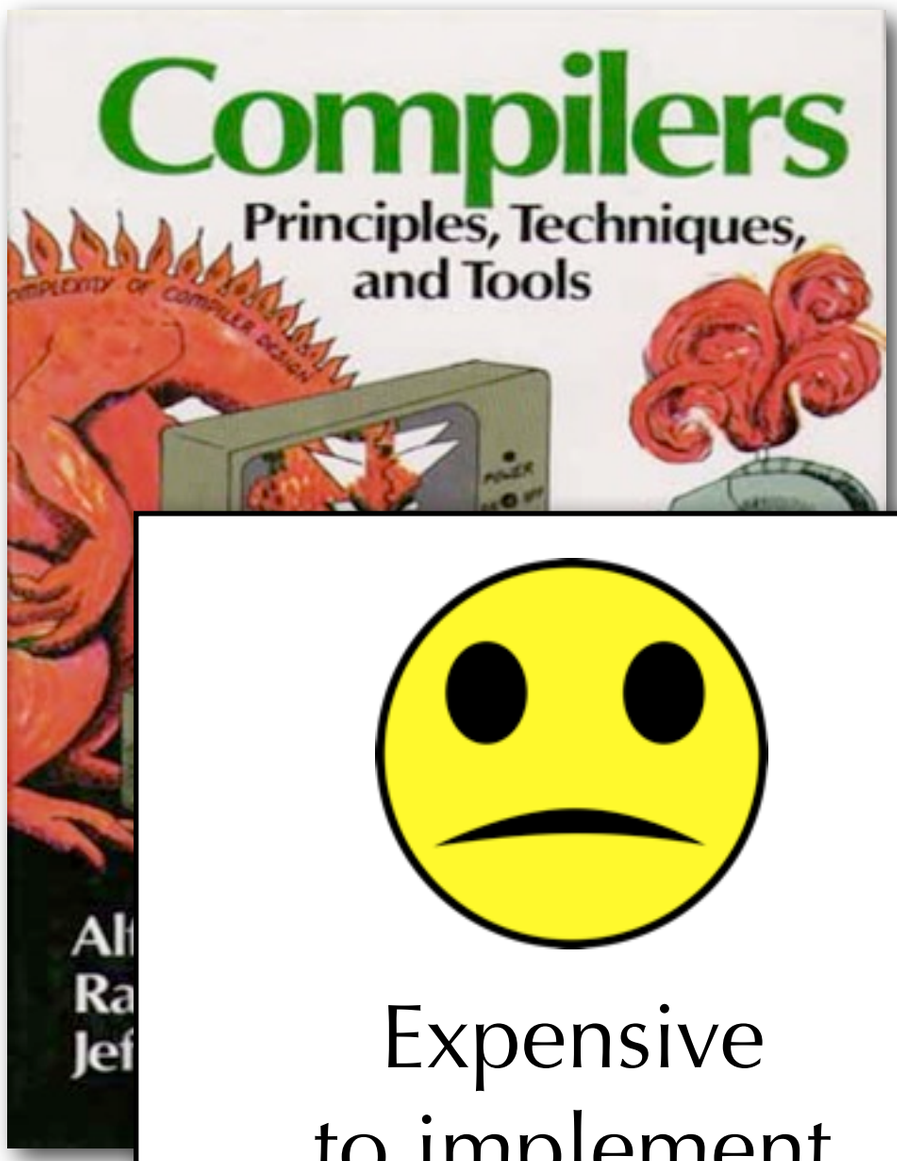


Compilers, programmers & sequential



Simple and intuitive programming model

Compilers, programmers & sequential



Expensive
to implement

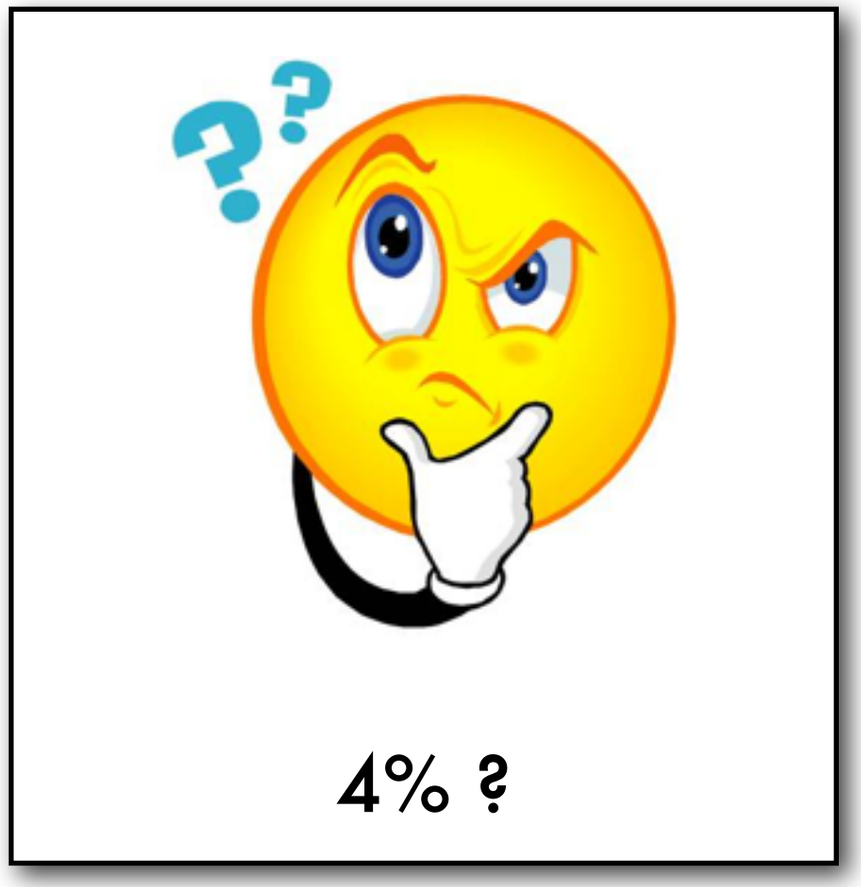
Simple and intuitive
programming model

A Case for an SC-Preserving Compiler

Daniel Marino† Abhayendra Singh* Todd Millstein† Madanlal Musuvathi‡ Satish Narayanasamy*

†University of California, Los Angeles *University of Michigan, Ann Arbor ‡Microsoft Research, Redmond

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



A Case for an SC-Preserving Compiler

Daniel Marino† Abhayendra Singh* Todd Millstein† Madanlal Musuvathi‡ Satish Narayanasamy*

†University of California, Los Angeles *University of Michigan, Ann Arbor ‡Microsoft Research, Redmond

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

EXCESSIVE OVERHEAD

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

The layman solution

forbid data-races

Data-race freedom

Our examples again:

Thread 0	Thread 1
`*y = 1`	`if *x == 1`
`*x = 1`	`then print *y`

Observable behaviour: `0`

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

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

Data-race freedom

Thread 0	Thread 1
*x = 1	if *x == 1

...intuition...

Programming languages provide
synchronisation mechanisms

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

conflicting ways (e.g. one thread writes the datas read by the other).

The basic solution

Thread 0	Thread 1
`*y = 1`	`if *x == 1`
`*x = 1`	`then print *y`

Observable behaviour: `0`

Prohibit *data races*

Defined as follows:

- two memory operations **conflict** if they access the same memory location and at least one is a store operation;
- a **SC execution** (interleaving) **contains a data race** if two conflicting operations corresponding to different threads are adjacent (maybe executed concurrently).

Example: a data race in the example above:

$$\mathsf{W}_{t_1}\, y{=}1, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, y{=}1, \mathsf{P}_{t_2}\, 1$$

The basic solution

Thread 0	Thread 1
`*y = 1`	`if *x == 1`
`*x = 1`	`then print *y`

Observable behaviour: `0`

Prohibit *data races*

Defined as follows:

The definition of data race quantifies *only* over the sequential consistent executions

executed concurrently).

Example: a data race in the example above:

$$\mathsf{W}_{t_1}\, y{=}1, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, y{=}1, \mathsf{P}_{t_2}\, 1$$

How do we avoid data races? (high-level languages)

- **Locks**

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

- **Atomic variables**

  Allow concurrent access "exempt" from data races (called volatile in Java).

Example:

Thread 0	Thread 1

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

How do we avoid data races? (high-level languages)

Thread 0	Thread 1

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

This program is data-race free:

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

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

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

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

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

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

How do we avoid data races? (high-level languages)

- `lock()`, `unlock()` are opaque for the compiler: viewed as potentially modifying any location, memory operations cannot be moved past them
- `lock()`, `unlock()` contain *"sufficient fences"* to prevent hardware reordering across them and global orderering

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

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

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

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

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

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

guages)

Compiler/hardware can continue to reorder accesses

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

be

moved past them

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

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

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

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

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

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

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

Validity of compiler optimisations,

Transformation	SC	DRF
Memory trace preserving transformations	✓	✓
Redundant read after read elimination	✓*	✓
Redundant read after write elimination	✓*	✓
Irrelevant read elimination	✓	✓
Redundant write before write elimination	✓*	✓
Redundant write after read elimination	✓*	✓
Irrelevant read introduction	✓	×
Normal memory accesses reordering	×	✓
Roach-motel reordering	×(✓for locks)	✓
External action reordering	×	✓

* Optimisations legal only on adjacent statements.

Validity of compiler optimisations,

Transformation	SC
Memory trace preserving transformations	✓

Jaroslav Sevcik

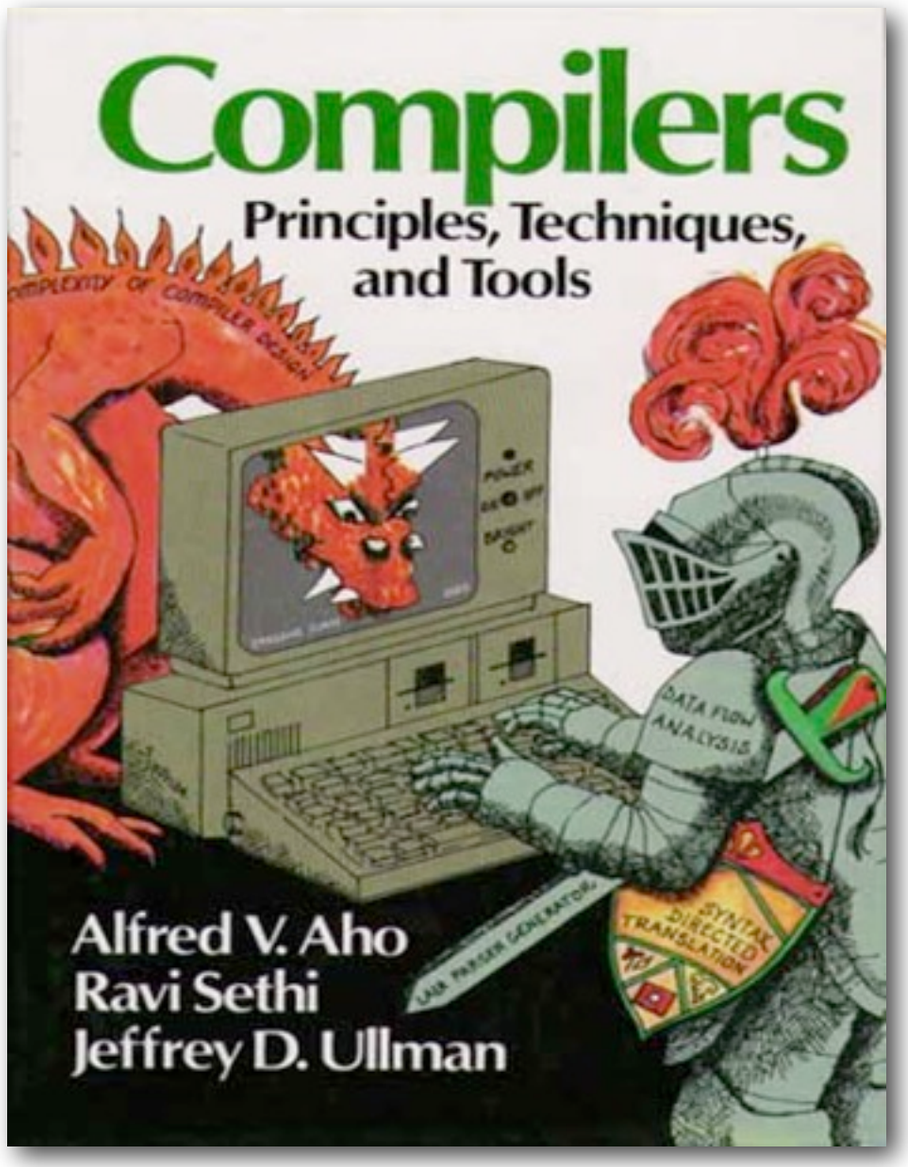
Safe Optimisations for Shared-Memory Concurrent Programs

PLDI 2011

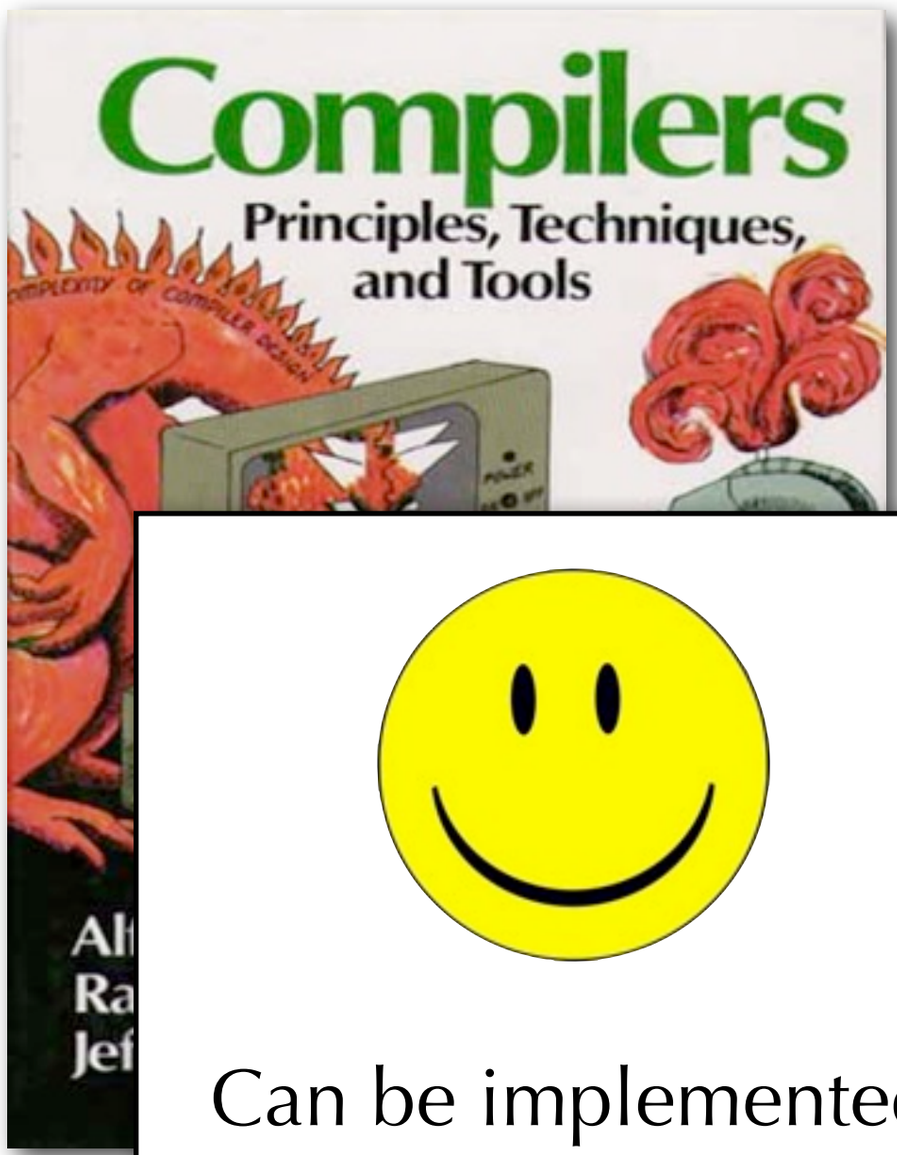
Roach-motel reordering	×(✓for locks)	✓
External action reordering	×	✓

* Optimisations legal only on adjacent statements.

Compilers, programmers & data-race

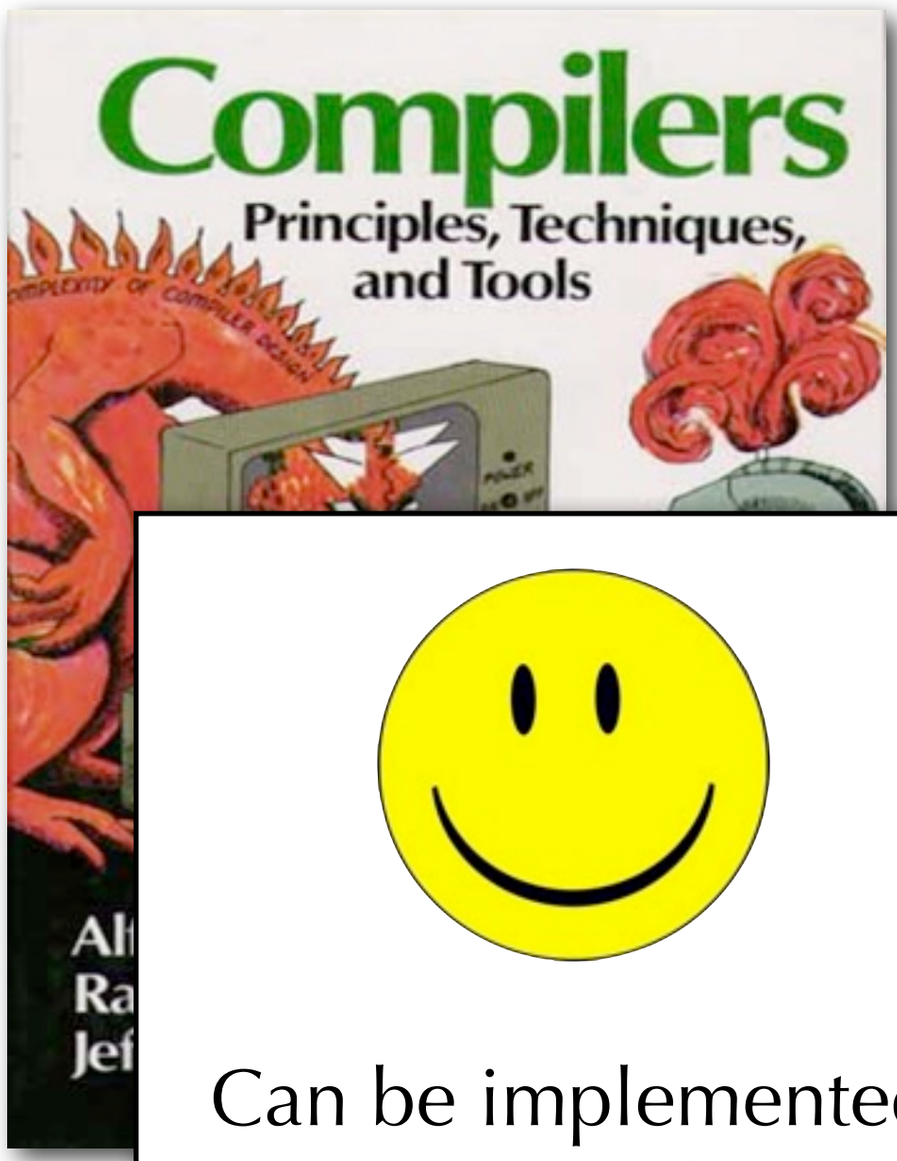


Compilers, programmers & data-race



Can be implemented efficiently

Compilers, programmers & data-race



Can be implemented efficiently

Intuitive programming model (but detecting races is tricky!)

Another example of DRF program

Exercise: is this program DRF?

Thread 0	Thread 1
`if *x == 1`	`if *y == 1`
`then *y = 1`	`then *x = 1`

Another example of DRF program

Exercise: is this program DRF?

Thread 0	Thread 1
`if *x == 1`	`if *y == 1`
`then *y = 1`	`then *x = 1`

Answer: yes!

The writes cannot be executed in any SC execution, so they cannot participate in a data race.

Another example of DRF program

Exercise: is this program DRF?

Data-race freedom is not the ultimate panacea

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

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

  my program ended with a *correct* result:
  my program is correct OR it has a data-race

Defining programming language memory models

Option 1

Don't.

No concurrency.

Implemented by highly-successful programming languages (**OCaml**)

Poor match for current trends

Option 2

Don't.

No shared memory

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

Option 3

Don't.

But language ensures data-race freedom

Possible:

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

Not suitable for general purpose programming.

Option 4

Don't.

Leave it (sort of) up to the hardware

Example:

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

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

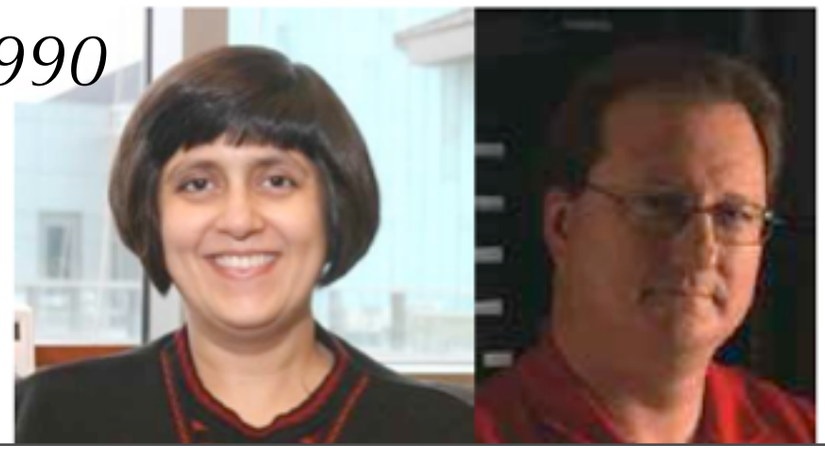
Option 5

Do.

Use data race freedom as a definition

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

Sarita Adve & Mark Hill, 1990



Option 5

Do.

Use data race freedom as a definition

Pro:

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

Cons:

- undecidable premise
- can't write racy programs (escape mechanisms?)

Ada 83

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

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

The execution of the program is erroneous if any of these assumptions is violated.

Data-races are errors

Posix Threads Specification

[IEEE 1003.1-2008, Base Definitions 4.11] Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it.

Data-races are errors

C++ 2011 / C1x

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

Data-races are errors

C++ 2011 / C1x

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

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

Data-races are errors

Escape lanes for expert programmers

Low-level atomics in C11/C++11

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

void lock(unsigned index) {
    if (0 == index) {
        flag0.store(1, std::memory_order_relaxed);
        turn.exchange(1, std::memory_order_acq_rel);

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

        while (flag0.load(std::memory_order_acquire)
            && 0 == turn.load(std::memory_order_relaxed))
            std::this_thread::yield();
    }
}

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

Atomic variable declaration

New syntax
for memory accesses

Qualifier

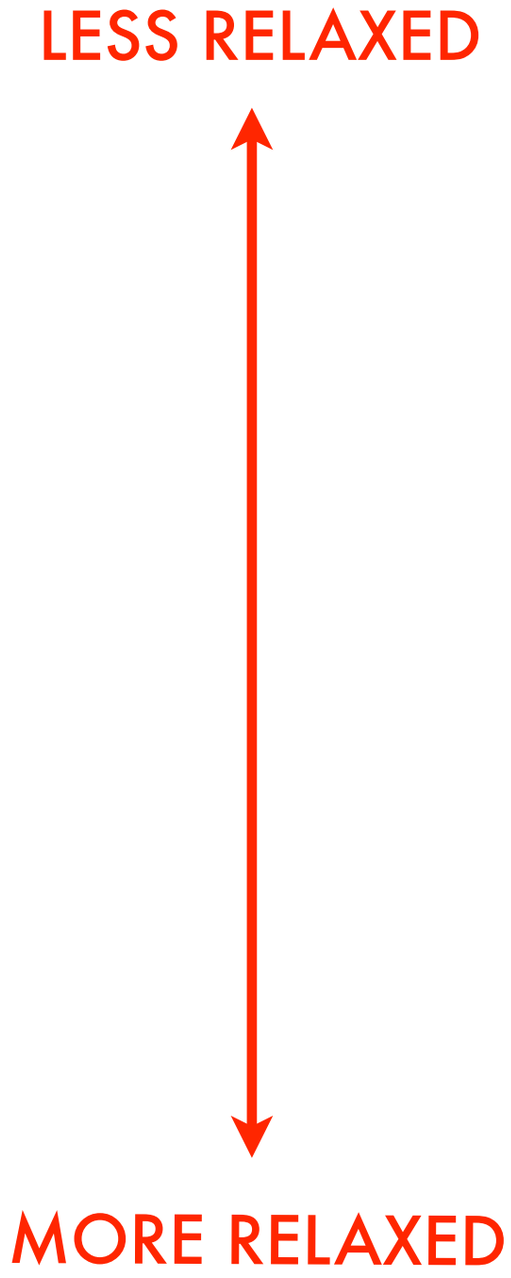
The qualifiers

```
MO_SEQ_CST

MO_RELEASE / MO_ACQUIRE

MO_RELEASE / MO_CONSUME

MO_RELAXED
```



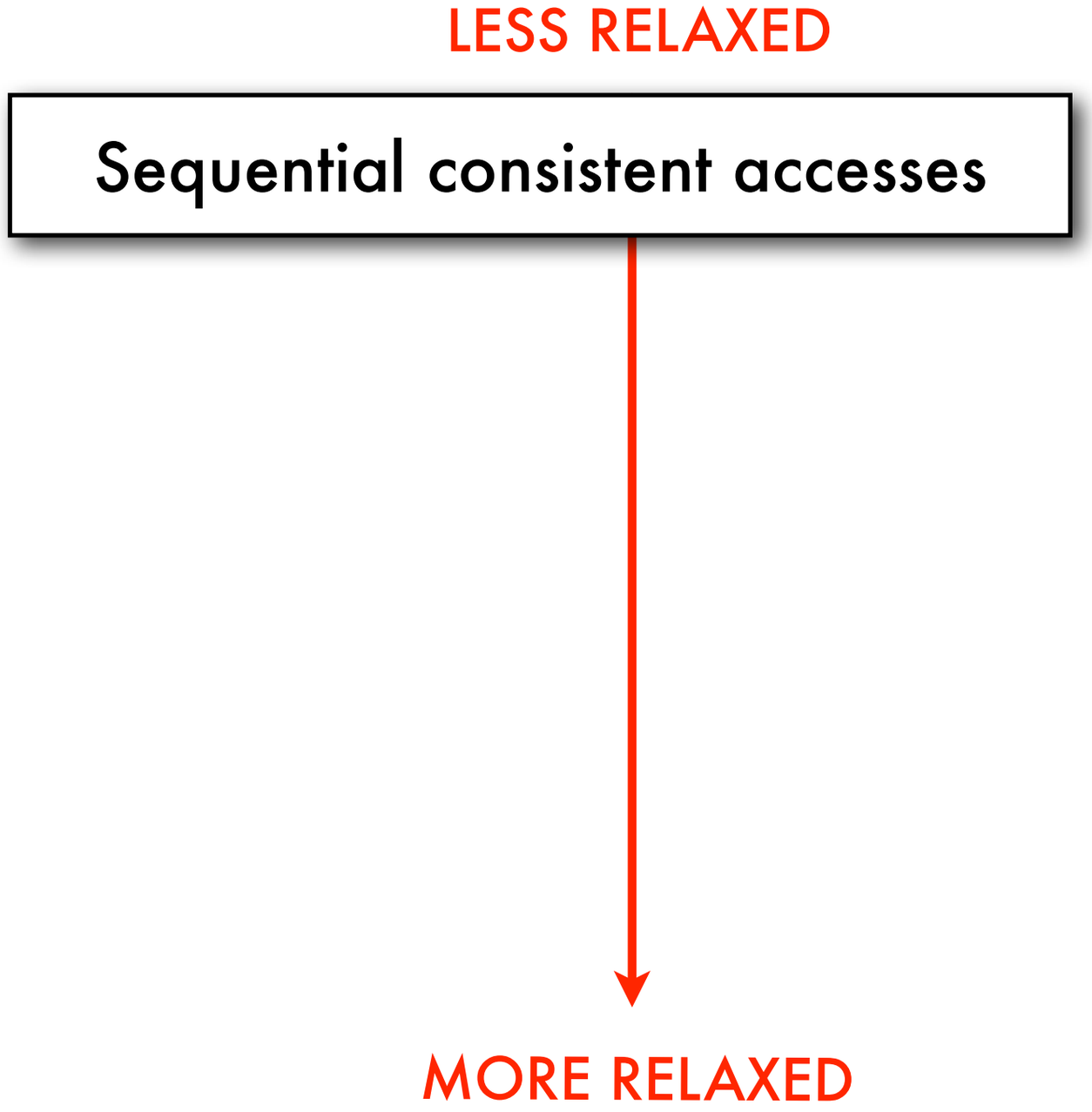
The qualifiers

`MO_SEQ_CST`

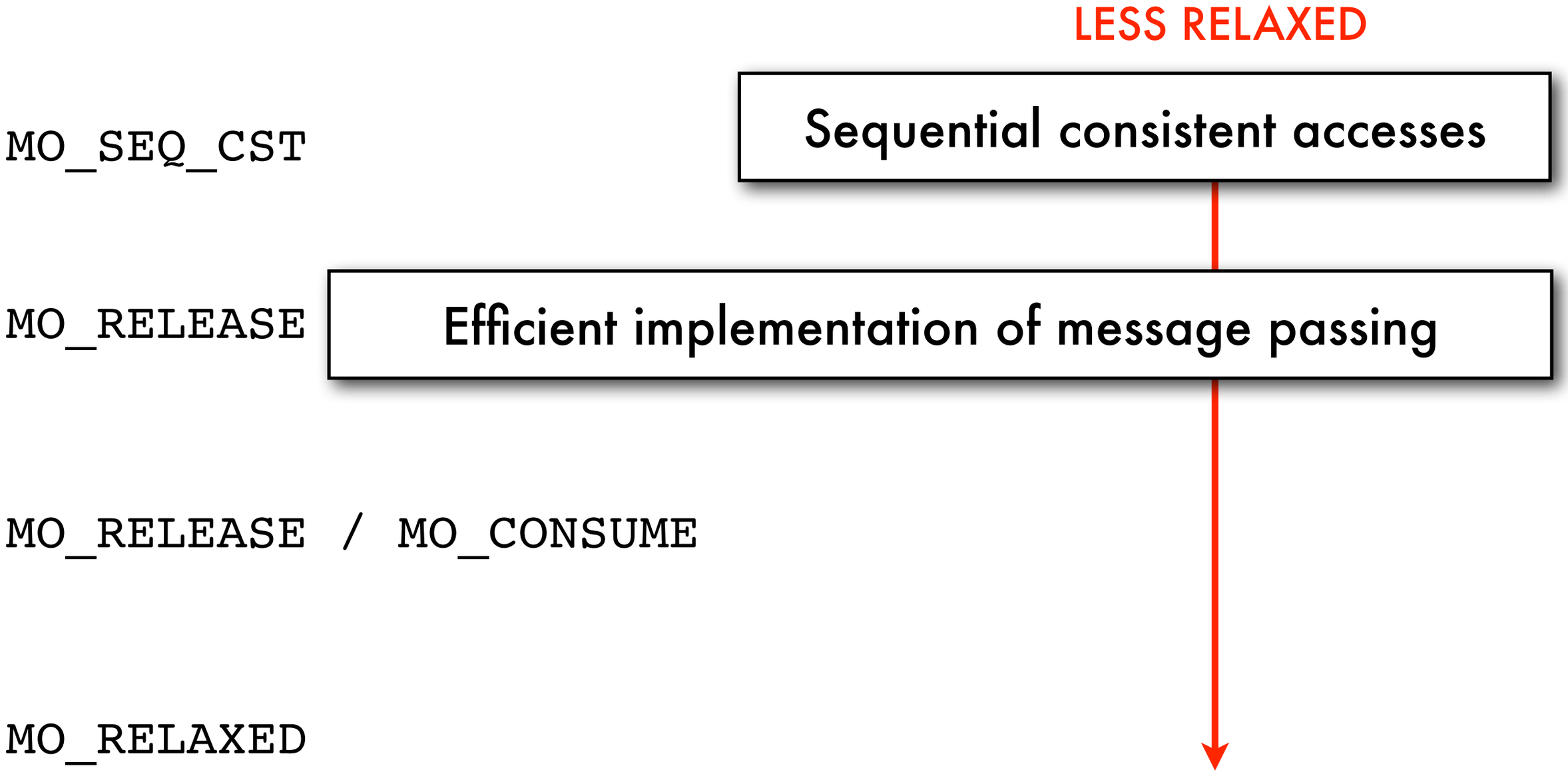
`MO_RELEASE / MO_ACQUIRE`

`MO_RELEASE / MO_CONSUME`

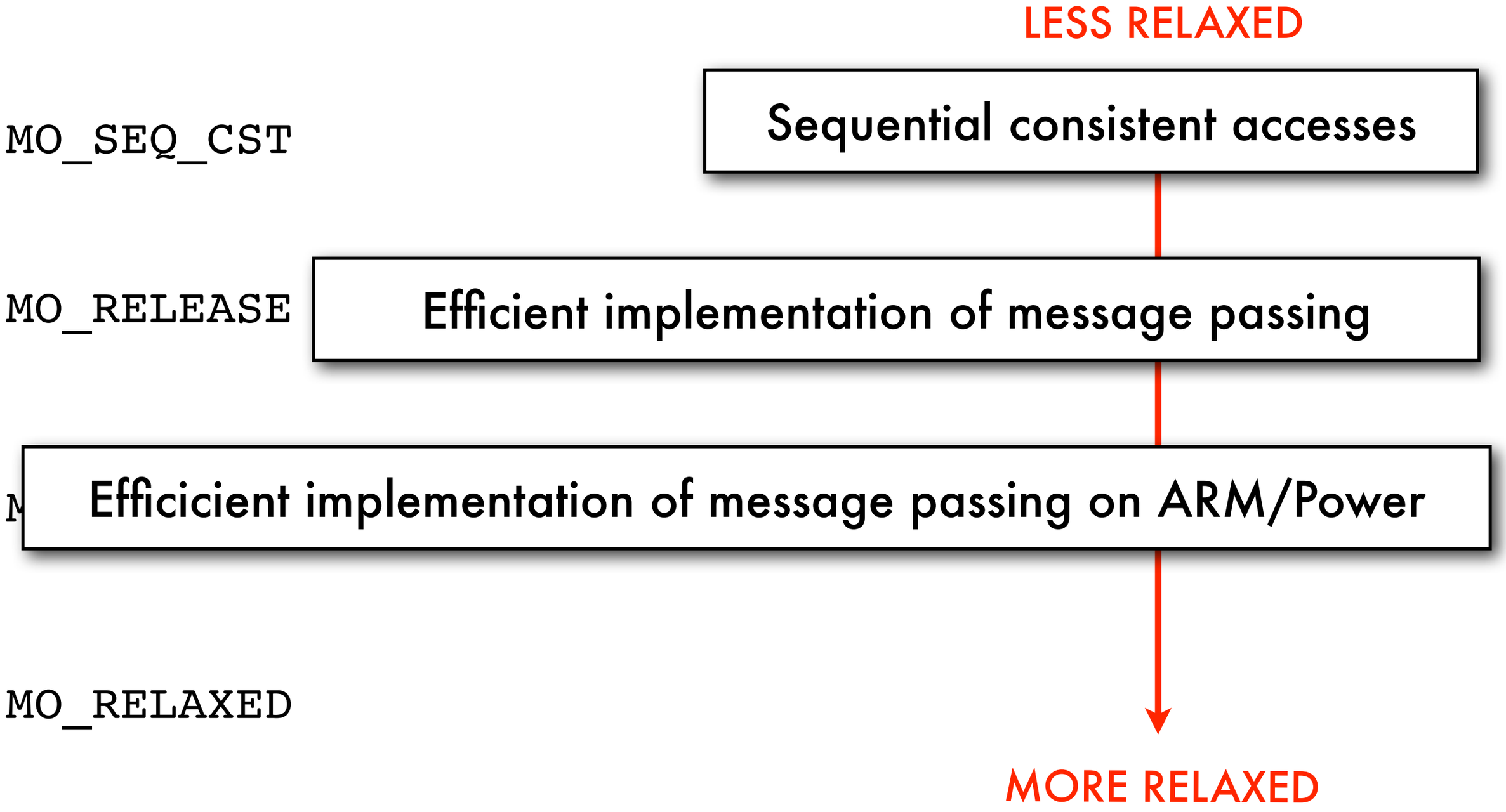
`MO_RELAXED`



The qualifiers



The qualifiers



The qualifiers

LESS RELAXED

MO_SEQ_CST — Sequential consistent accesses

MO_RELEASE — Efficient implementation of message passing

M — Efficicient implementation of message passing on ARM/Power

MO_RELAX — No synchronisation; direct access to hardware

MORE RELAXED

MO_SEQ_CST

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

Thread 0	Thread 1
`x.store(1,MO_SEQ_CST)`	`y.store(1,MO_SEQ_CST)`
`r1 = y.load(MO_SEQ_CST)`	`r2 = x.load(MO_SEQ_CST)`

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

Sample compilation on x86:

store: `MOV; MFENCE`
load: `MOV`

Sample compilation on Power:

store: `HWSYNC; ST`
load: `HWSYNC; LD; CMP; BC; ISYNC`

MO_RELAXED

MO_RELAXED accesses can be reordered by compiler/hardware

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`y.store(1,MO_RELAXED)`
`r1 = y.load(MO_RELAXED)`	`r2 = x.load(MO_RELAXED)`

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

Sample compilation on x86:

store: **`MOV`**
load: **`MOV`**

Sample compilation on Power:

store: **`ST`**
load: **`LD`**

MO_RELEASE / MO_ACQUIRE

Supports a fast implementation of the message passing idiom:

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`r1 = y.load(MO_ACQUIRE)`
`y.store(1,MO_RELEASE)`	`r2 = x.load(MO_RELAXED)`

The program above cannot end with `r1 = 1` and `r2 = 0`.

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

Sample compilation on x86:

store: `MOV`
load: `MOV`

Sample compilation on Power:

store: `LWSYNC; ST`
load: `LD; CMP; BC; ISYNC`

MO_RELEASE / MO_CONSUME

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

Thread 0	Thread 1
`x.store(1,MO_RELAXED)`	`r1 = y.load(x,MO_CONSUME)`
`y.store(&x,MO_RELEASE)`	`r2 = (*r1).load(MO_RELAXED)`

The program above cannot end with `r1 = 1` and `r2 = 0`.

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

Sample compilation on x86:

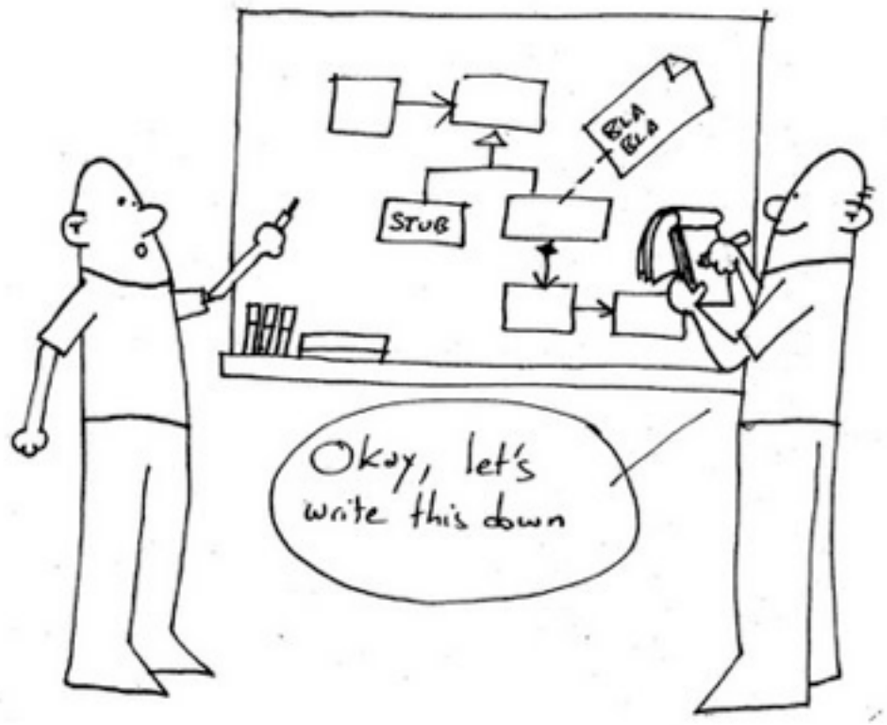
store: **`MOV`**
load: **`MOV`**

Sample compilation on Power:

store: **`LWSYNC; ST`**
load: **`LD`**

The C11/C++11 memory model formalisation

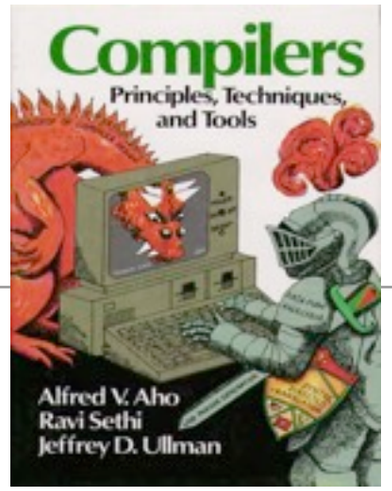
[demo]



Enough about formalising...

...what about reasoning?

Topics

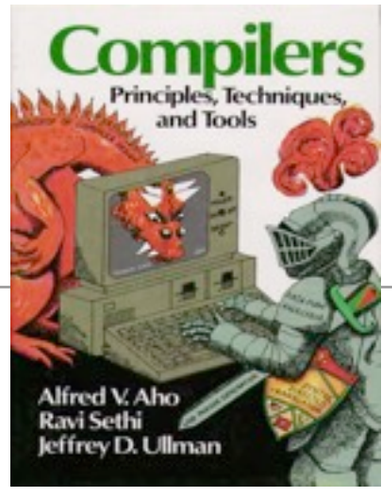


1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compilers and optimisations: proof and/or validation

Topics



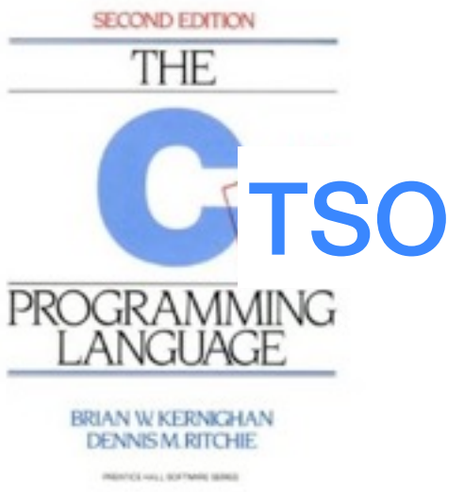
1. Formalisation of hardware memory models

2. Design and formalisation of programming languages

3. Compilers and optimisations: proof and/or validation

CompCertTSO

Idea: the programming language *faithfully* mimics the processor model.



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

A semantic preserving compiler
CompCertTSO
building on CompCert 1.5

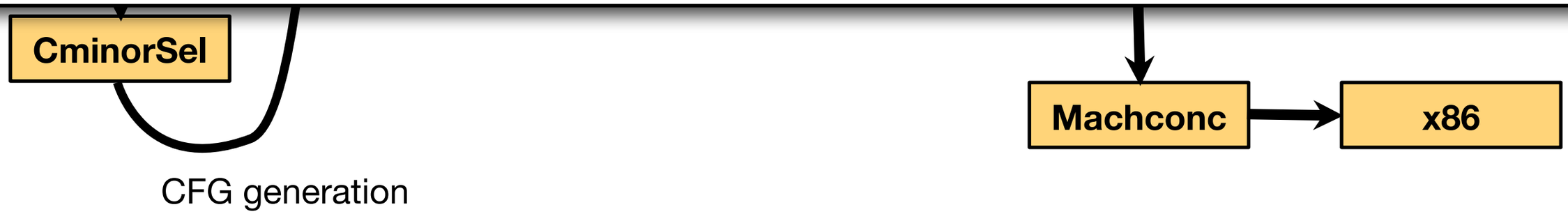
Intel processors implement the x86-TSO MM

CompCertTSO

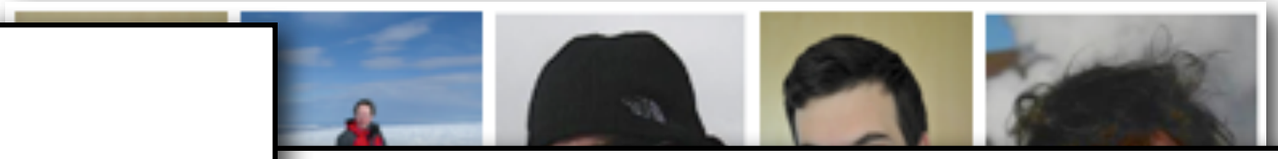
CompCert 1.5 proves that all behaviours of the source program are behaviours of the compiled program (building simulation relations).

The converse follows from determinacy of the semantics.

Problem: in CompCertTSO the semantics is not deterministic...



Semantic engineering



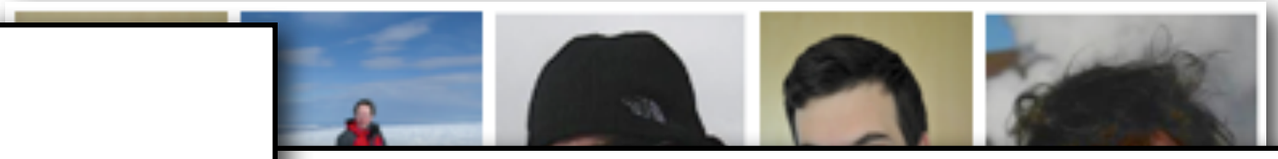
Proof sketch

Want: whole-system *upward* simulation

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

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

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



Semantic engineering

Want: w

Have: L

1. replac

lo 2. port L

- surpr
- tedic

roof)

3. Turn p

lations

4. Turn per-thread upward simulations to whole-system upward simulations

5. Compose the whole system u

ClightTSO small-step semantics
has about 90 reduction rules

How to formalise programming language definitions?

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

The Ott tool

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

```
  {{ type_to_chunk [[ty1]] = Some [[c]] }}
  {{ cast_value_to_chunk [[c]] [[v1]] = [[v2]] }}
------------------------------------------------------------ :: Assign
  v1.[(p1:ty1)= __].k |env   --mem(write p1 c v2)--> skip.k |env
```

Latex

$$\frac{\texttt{type\_to\_chunk}\ ty_1 = \texttt{Some}\ c \qquad \texttt{cast\_value\_to\_chunk}\ c\ v_1 = v_2}{v_1 \cdot [p_1\,{}^{ty_1}\!= \_] \cdot \kappa_s\,|_\rho \xrightarrow{\texttt{mem (write}\ p_1\ c\ v_2\texttt{)}} \texttt{skip} \cdot \kappa_s\,|_\rho}\ \textsc{StepAssign}$$

Proof assistant

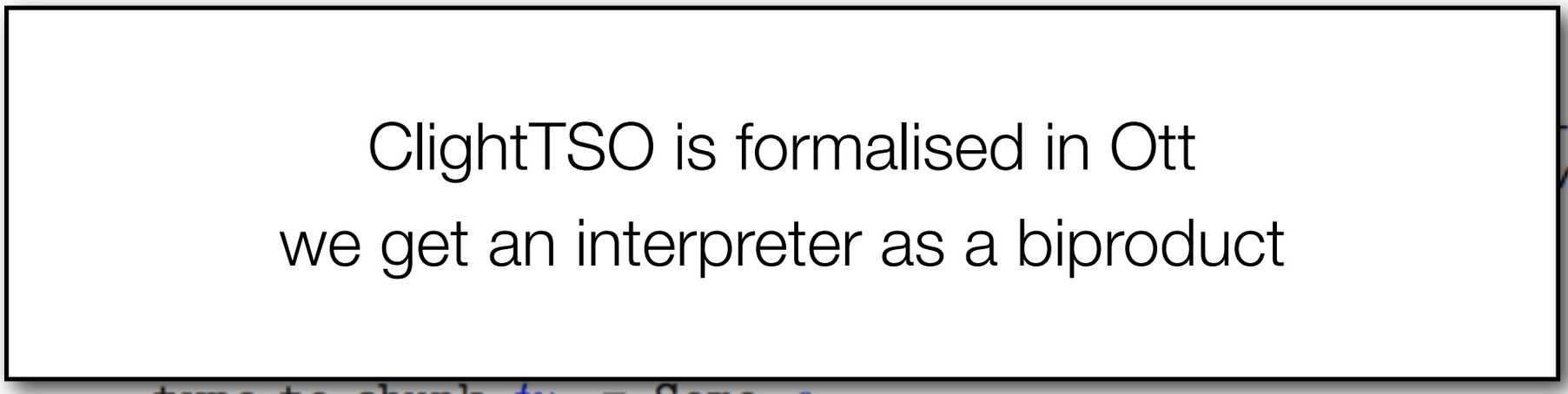
```
 | StepAssign : forall (v1:val) (p1:pointer) (ty1:type) (k:cont) (env:env)
(c:memory_chunk) (v2:val),
     type_to_chunk ty1 = Some c ->
     cast_value_to_chunk c v1 = v2 ->
     cl_step (SKval v1 env (EKassign2 (Vptr p1) ty1 k ) ) (TEmem (MEwrite p1 c
(SKstmt Sskip env k )
```

The Ott tool

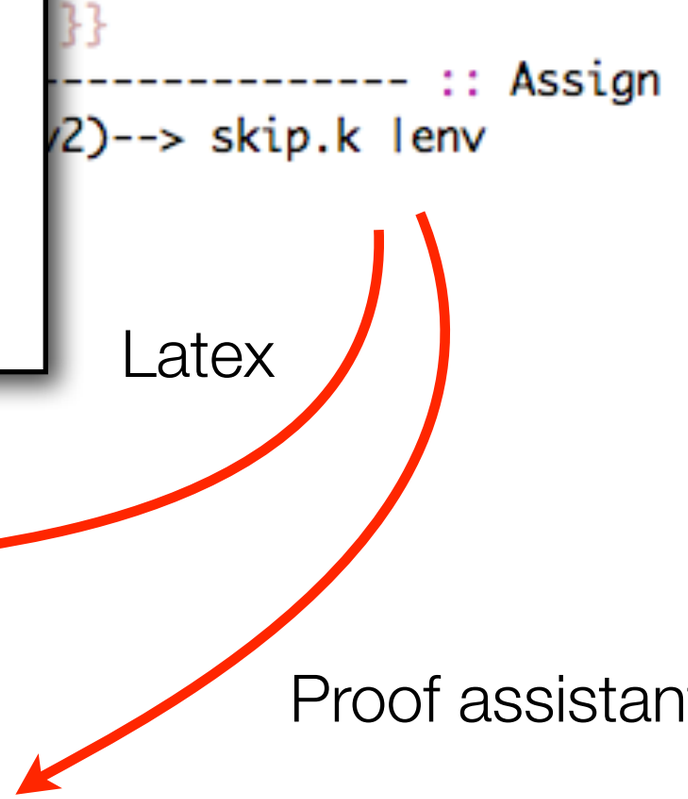
Complement to LEM, specialised for formalising programming language definitions and semantics.

ClightTSO is formalised in Ott

we get an interpreter as a biproduct

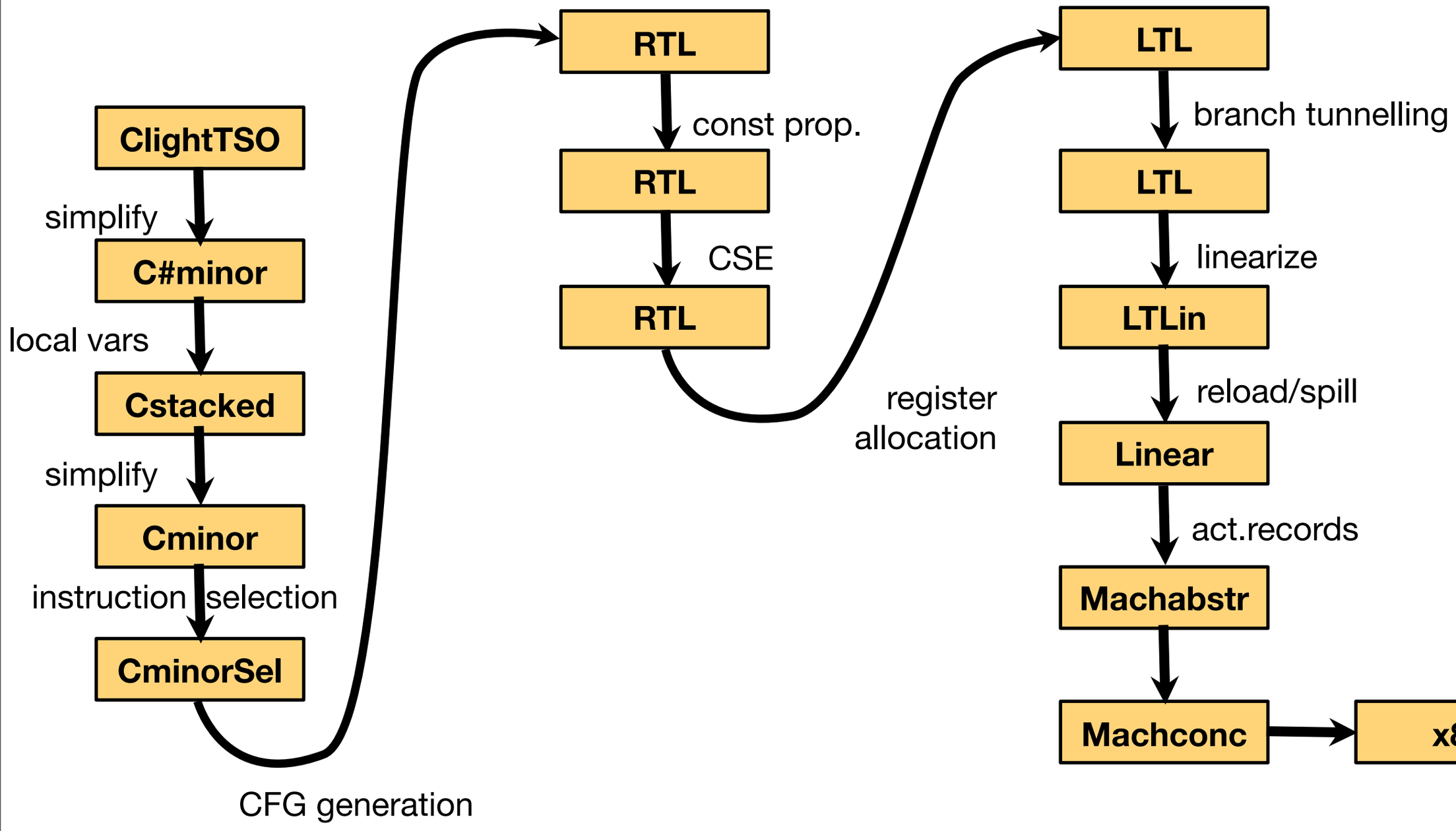


$$\frac{\texttt{type\_to\_chunk}\ ty_1 = \texttt{Some}\ c \qquad \texttt{cast\_value\_to\_chunk}\ c\ v_1 = v_2}{v_1 \cdot [p_1\,{}^{ty_1}\!= \_] \cdot \kappa_s\,|_\rho \xrightarrow{\texttt{mem (write}\ p_1\ c\ v_2\texttt{)}} \texttt{skip} \cdot \kappa_s\,|_\rho}\ \textsc{StepAssign}$$



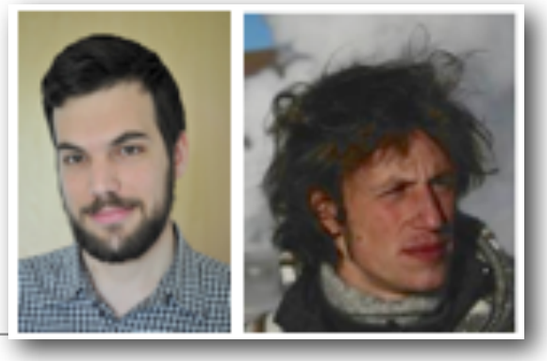

```
 | StepAssign : forall (v1:val) (p1:pointer) (ty1:type) (k:cont) (env:env)
(c:memory_chunk) (v2:val),
     type_to_chunk ty1 = Some c ->
     cast_value_to_chunk c v1 = v2 ->
     cl_step (SKval v1 env (EKassign2 (Vptr p1) ty1 k ) ) (TEmem (MEwrite p1 c
(SKstmt Sskip env k )
```

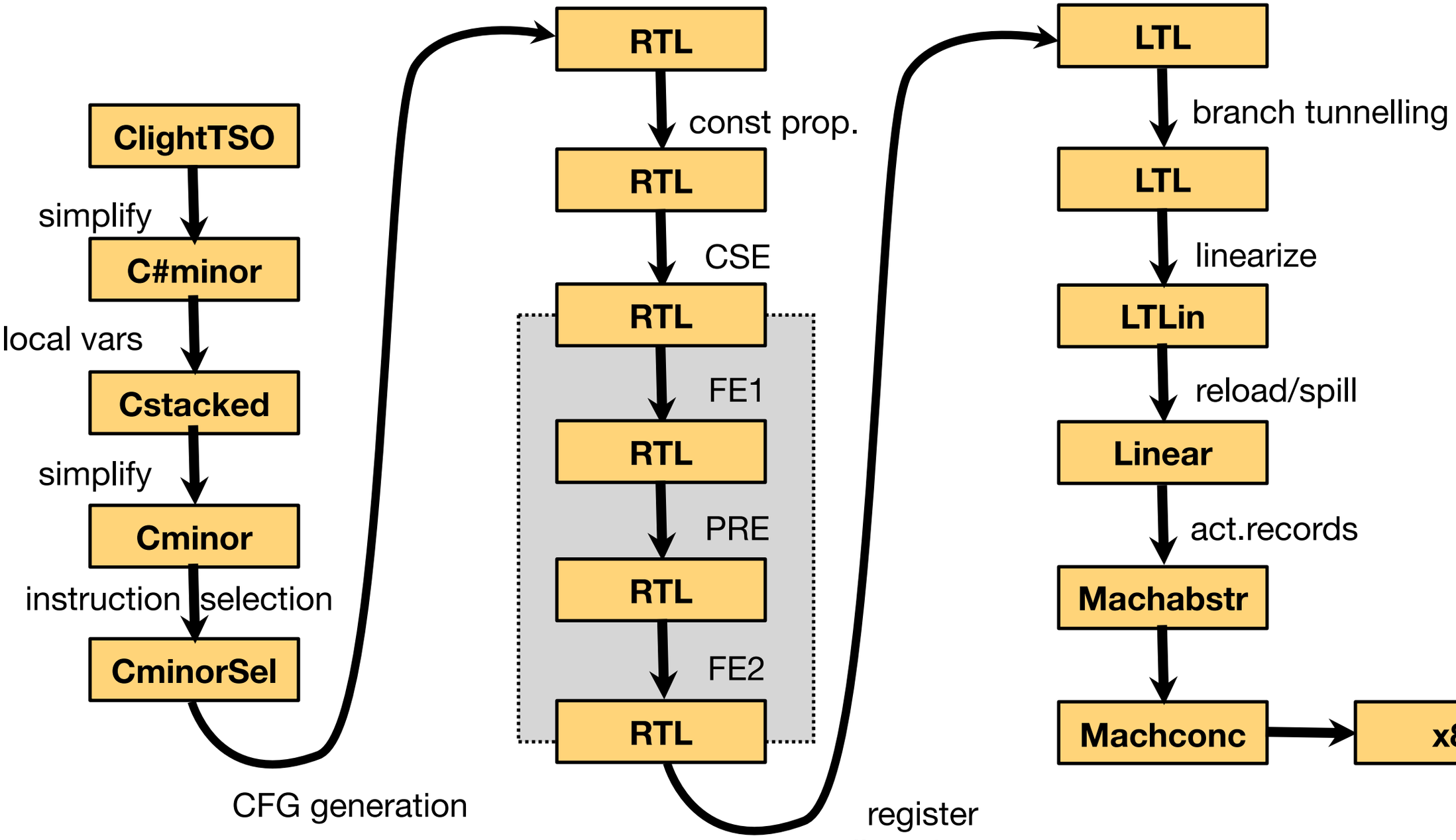
CompCertTSO



[POPL 2011]

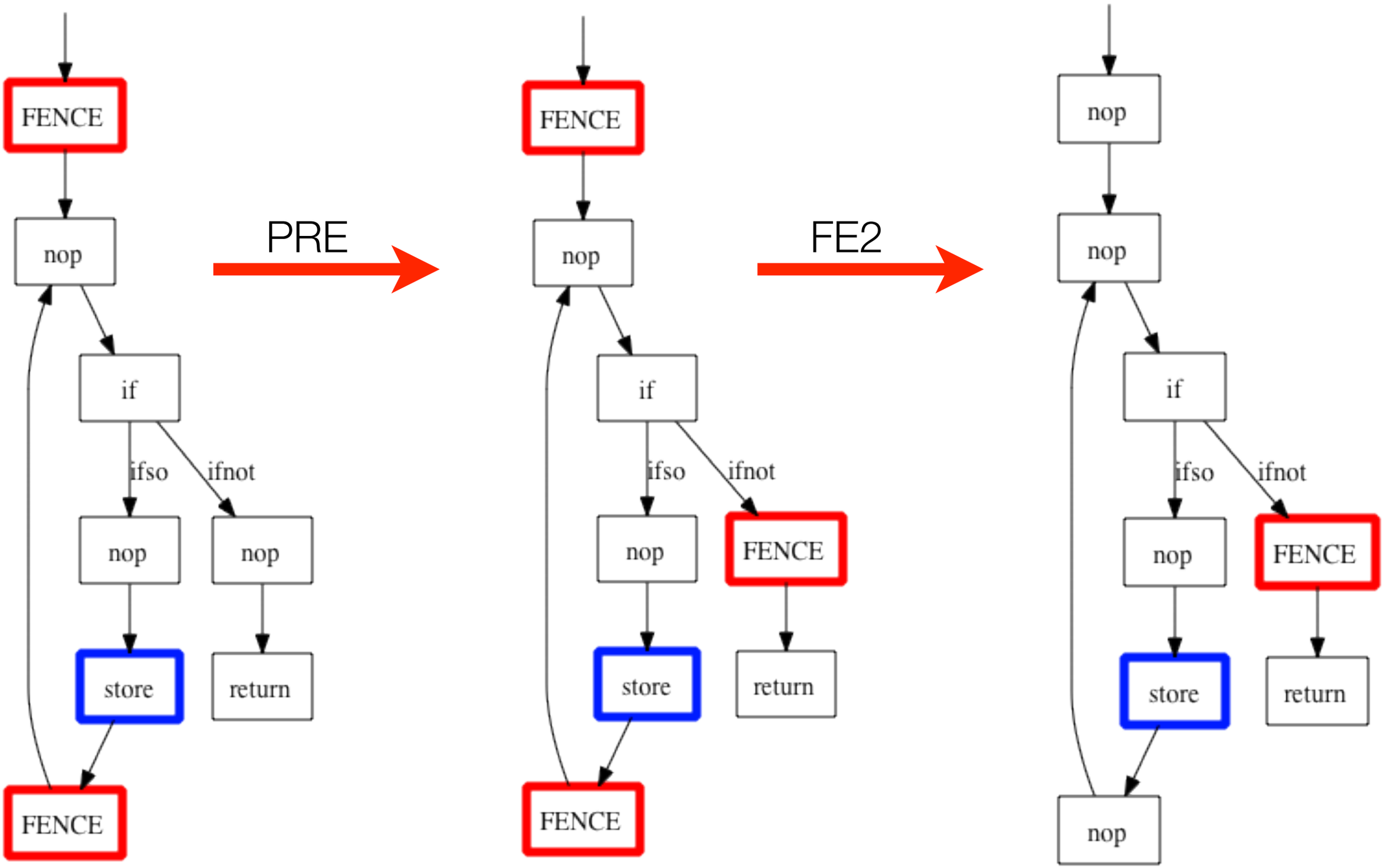
CompCertTSO + fence optimisations



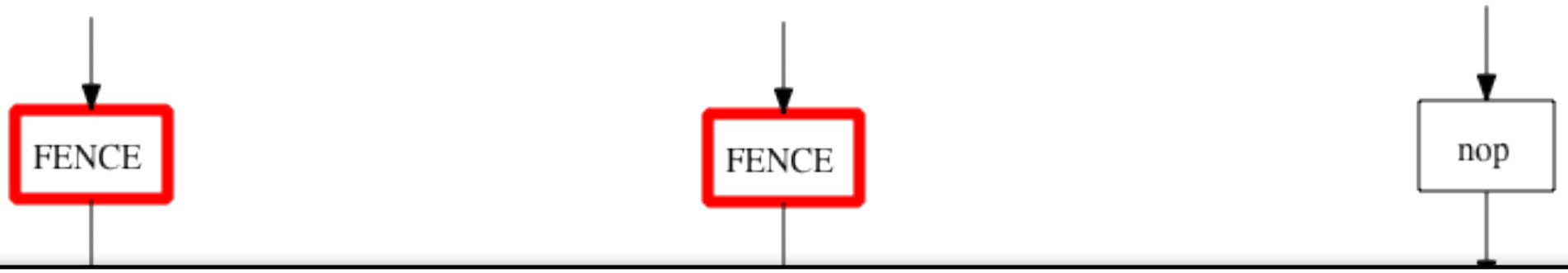


SAS 2012

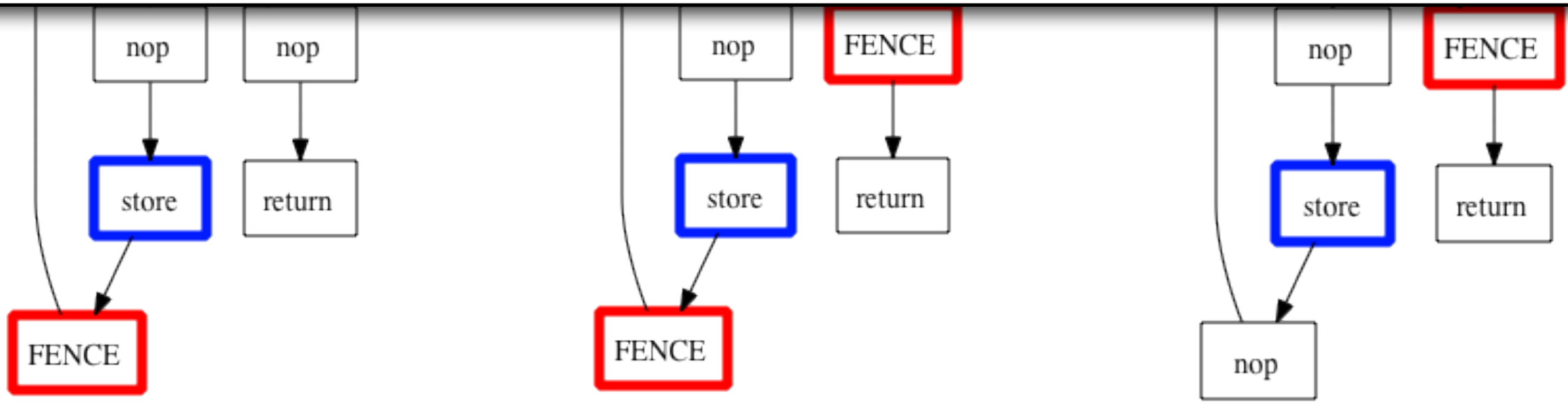
Example of fence elimination in action



Example of fence elimination in action



Proof of correctness requires a novel bisimulation-based proof technique (need to guess if "in the future" a fence instruction will be executed).



What about C11?

Shared memory

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

Thread 1

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

Thread 2

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

Can you guess the output?

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`

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

Shared memory

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

Thread 1

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

Thread 2

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

Thread 1 returns without modifying `b`

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

C11 states that this program *must* print `42`

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

gcc 4.7 -O2

...sometimes we get `0` on the screen

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

The outer loop can be (and is) optimised away

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

gcc 4.7 -O2

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

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

The compiled code saves and restores `b`

Correct result in a sequential setting

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

Shared memory

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

Thread 1

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

Thread 2

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

Shared memory

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

Thread 1

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

Thread 2

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

- Read `a` (1) into `eax`

Shared memory

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

Thread 1

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

Thread 2

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

- Read `a` (`1`) into `eax`
- Read `b` (`0`) into `ebx`

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

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

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

Shared memory

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

Thread 1

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

Thread 2

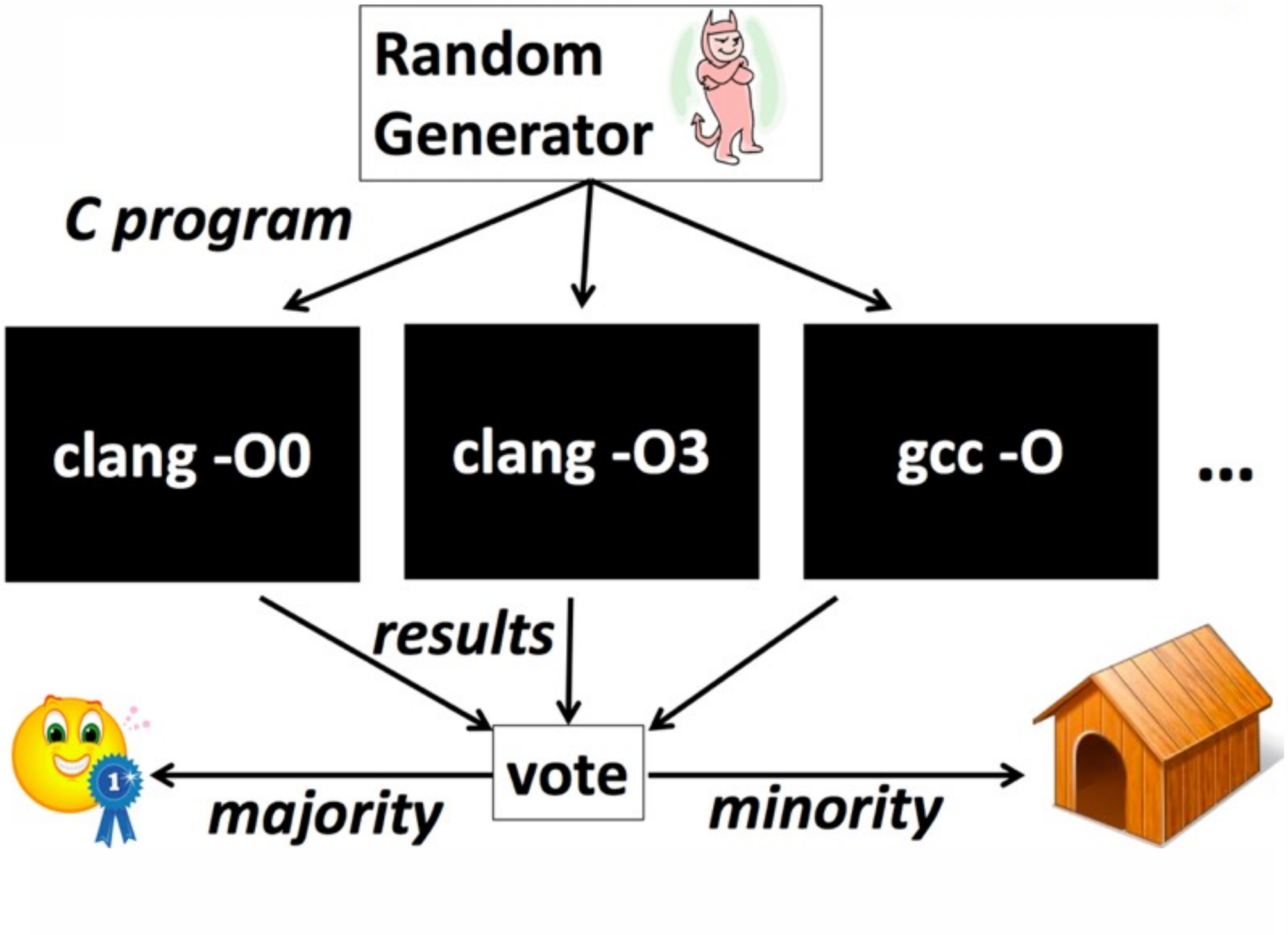
```
b = 42;
printf("%d\n", b);
```

- Read `a` (`1`) into `eax`
- Read `b` (`0`) into `ebx`
- Store `42` into `b`
- Store `ebx` (`0`) into `b`
- Print `b`: `0` is printed

The horror, the horror... a subtle compiler bug!

Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

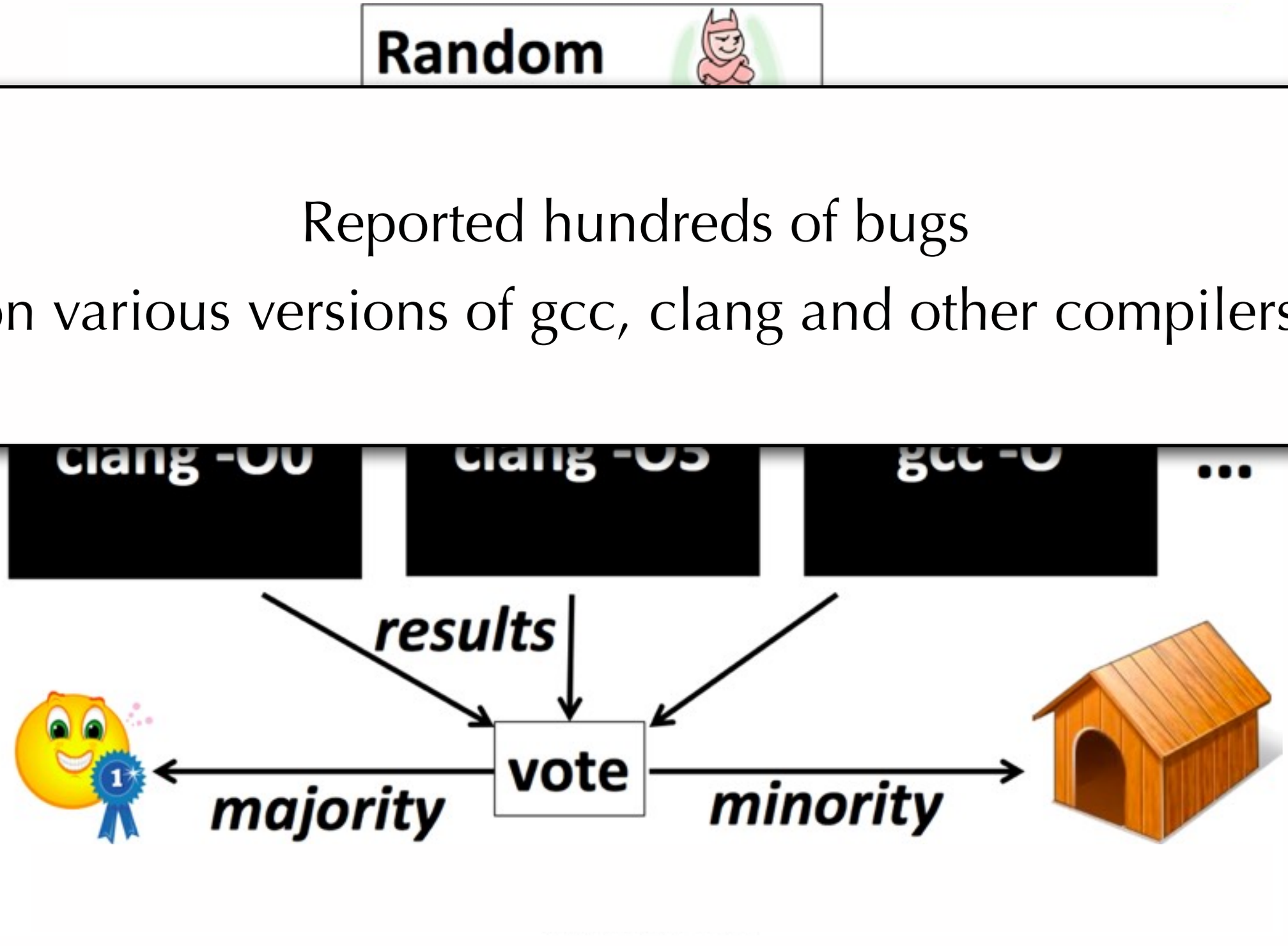


Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

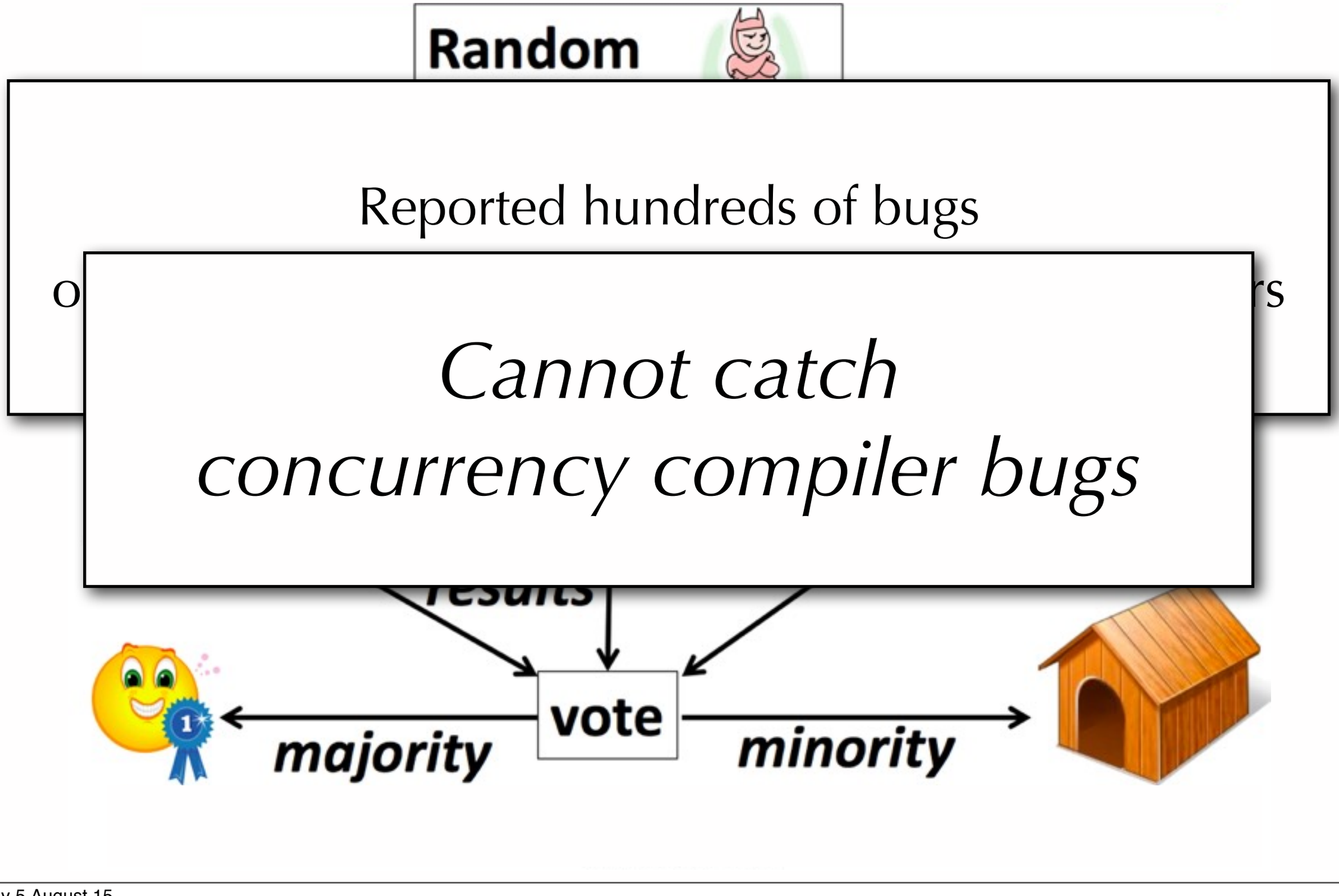
Reported hundreds of bugs

on various versions of gcc, clang and other compilers



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Hunting *concurrency* compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours:

how to test for correctness?

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

Idea

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

↓

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

Hunt concurrency compiler bugs

=

search for transformations of sequential code not sound in an arbitrary non-racy context

Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

tracing

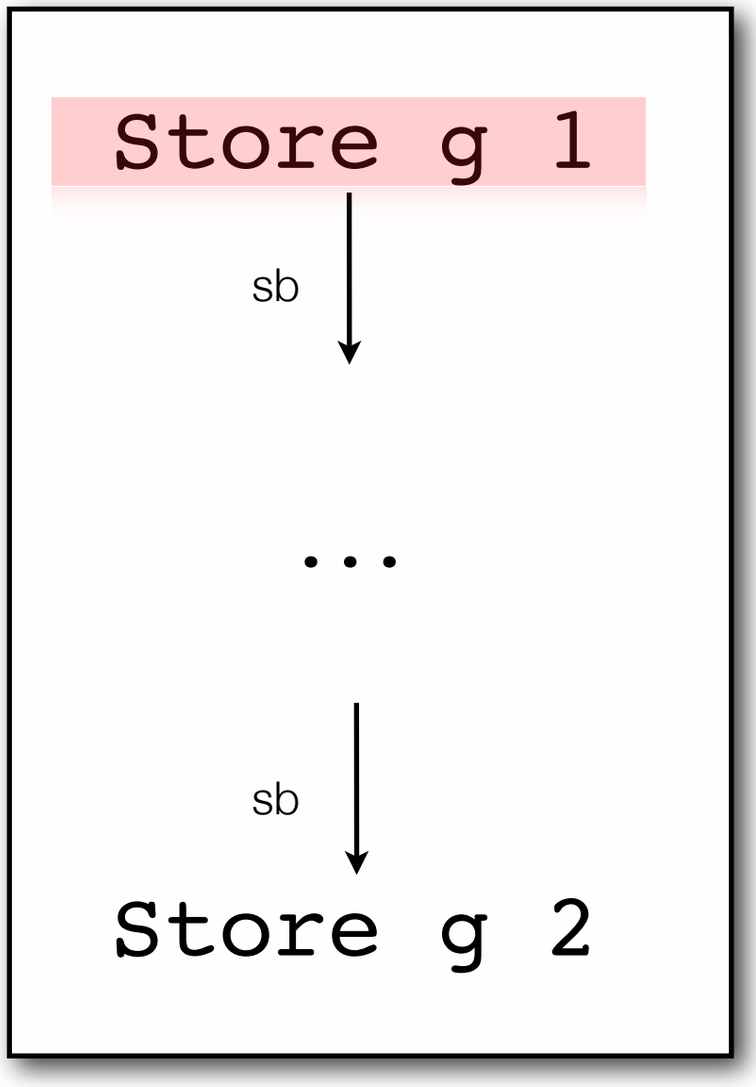
REFERENCE MEMORY TRACE

MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context

Soundness of compiler optimisations in the C11/C++11 memory model

Elimination of *overwritten writes*



Under which conditions is it correct to eliminate the first store?

A **same-thread release-acquire pair** is a pair of a release action followed by an acquire action in program order.

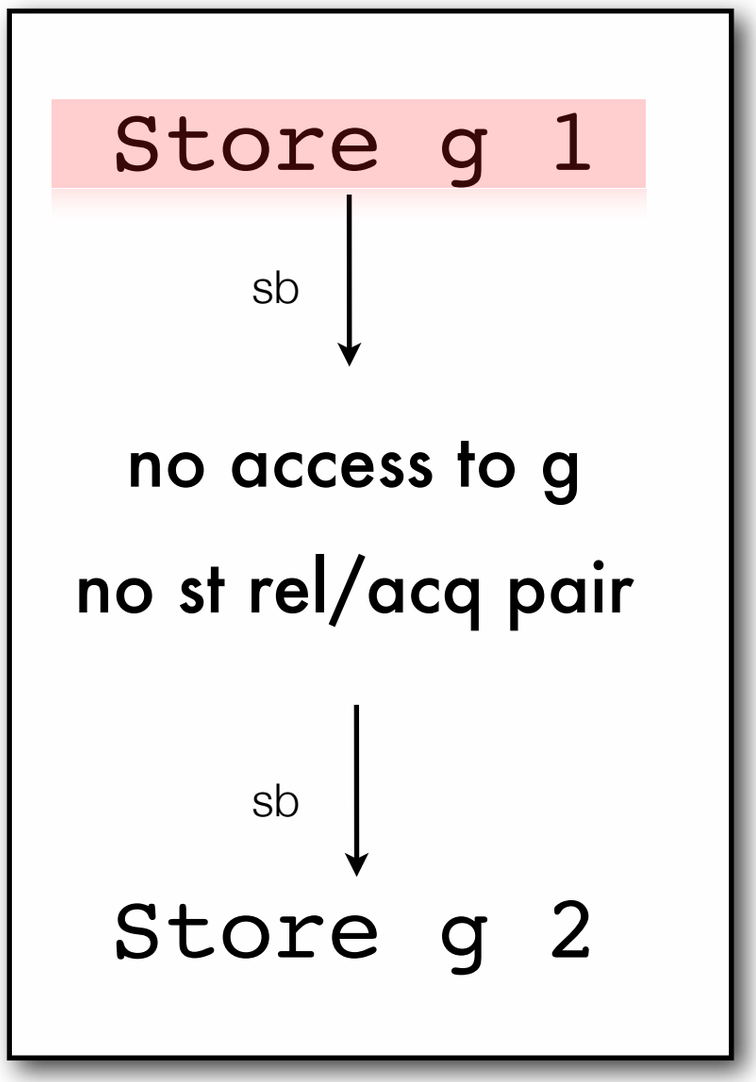
An action is a *release* if it is a possible source of a synchronisation

unlock mutex, release or seq_cst atomic write

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

lock mutex, acquire or seq_cst atomic read

Elimination of *overwritten writes*



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

1. no intervening accesses to g
2. no intervening
   same-thread release-acquire pair

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

candidate overwritten write

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

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

candidate overwritten write

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

same-thread release-acquire pair

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

Thread 2 is non-racy

The soundness condition

Shared memory

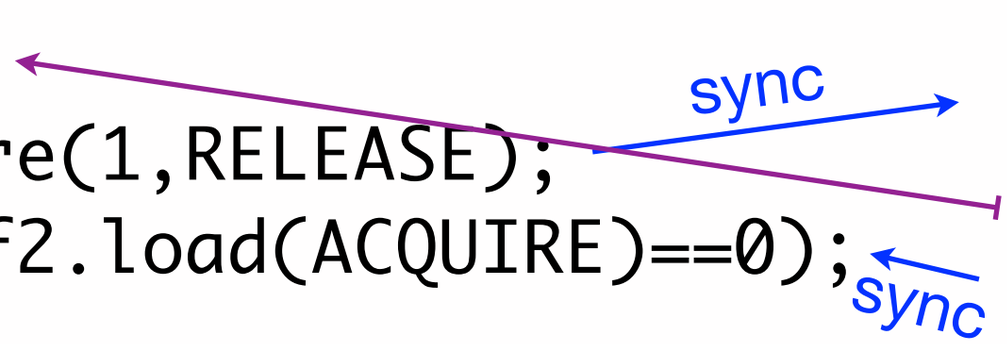
```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```



Thread 2 is non-racy

The program should only print 1

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

sync

sync

Thread 2 is non-racy

The program should only print 1

If we perform overwritten write elimination it prints 0

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

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

sync →

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);

g = 2;
```

sync →

Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
f1.store(1,RELEASE);

g = 2;
```

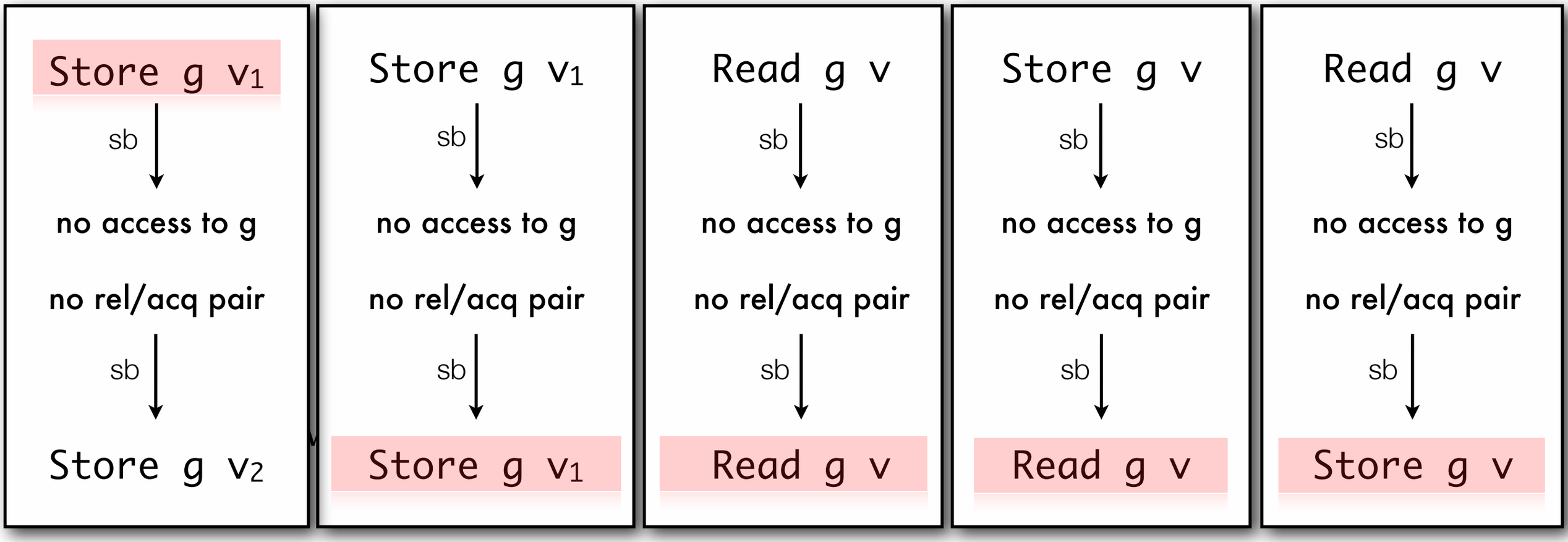
Thread 2

```
while(f1.load(ACQUIRE)==0);
printf("%d", g);
f2.store(1,RELEASE);
```

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

It is sound to optimise the overwritten write.

Eliminations: bestiary

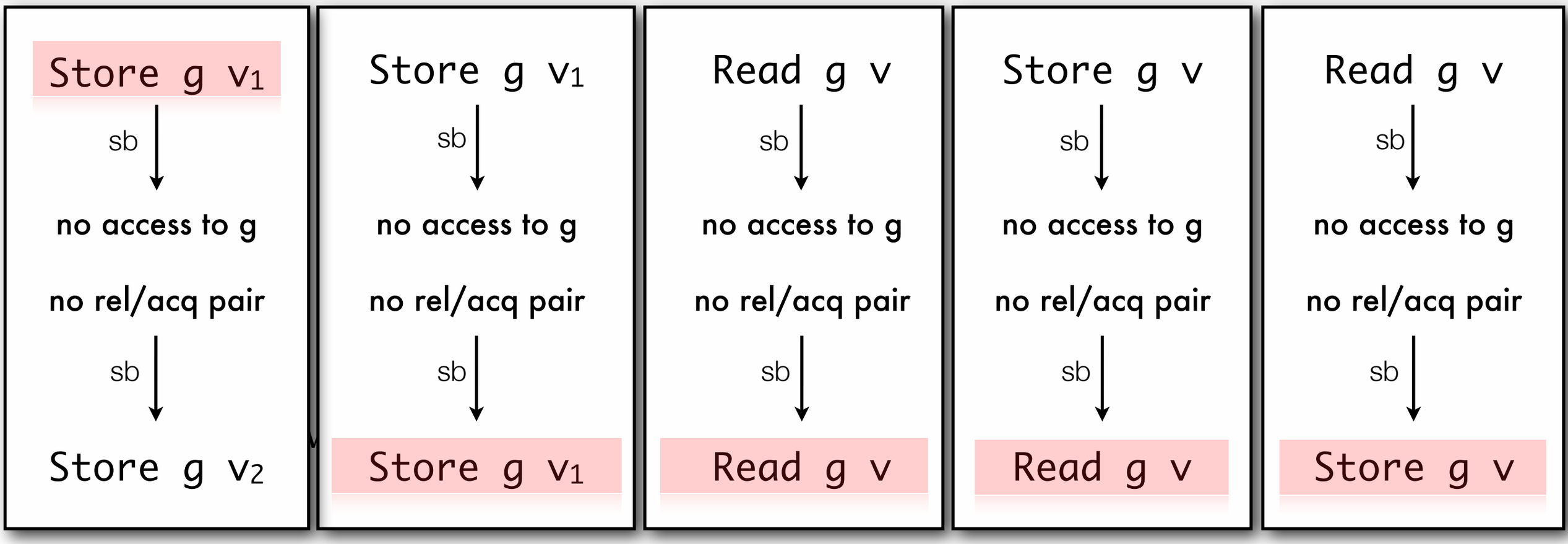


Overwritten-Write | Write-after-Write | Read-after-Read | Read-after-Write | Write-after-Read

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (*irrelevant reads*).

Also correctness statements for

reorderings, **merging**, and **introductions** of events.



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

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (*irrelevant reads*).

From theory to the Cmmtest tool

Random Generator

SEQUENTIAL PROGRAM

optimising compiler under test

reference semantics

EXECUTABLE

tracing

REFERENCE MEMORY TRACE

MEMORY TRACE

Check: only transformations sound in any concurrent non-racy context

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

optimising
compiler
under test

reference
semantics

EXECUTABLE

tracing

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

Check: only transformations sound
in any concurrent non-racy context

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

optimising
compiler
under test

reference
semantics

EXECUTABLE

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

Check: only transformations sound
in any concurrent non-racy context

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

gcc/clang -O0

optimising
compiler
under test

EXECUTABLE

EXECUTABLE

binary
instrumentation

REFERENCE
MEMORY
TRACE

binary
instrumentation

MEMORY
TRACE

Check: only transformations sound
in any concurrent non-racy context

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

gcc/clang -O0

optimising
compiler
under test

EXECUTABLE

EXECUTABLE

binary
instrumentation

binary
instrumentation

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

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

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

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

Start with a randomly generated well-defined program

```
const unsigned int g3 = 0UL;  void func_1(void){
long long g4 = 0x1;                 int *l8 = &g6;
int g6 = 6L;                        int l36 = 0x5E9D070FL;
volatile unsigned int g5 = 1UL;     unsigned int l107 = 0xAA37C3ACL;
                                    g4 &= g3;
                                    g5++;
                                    int *l102 = &l36;
                                    for (g6 = 4; g6 < (-3); g6 += 1);
                                    l102 = &g6;
                                    *l102 = ((*l8) && (l107 << 7)*(*l102));
                                }
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

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

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

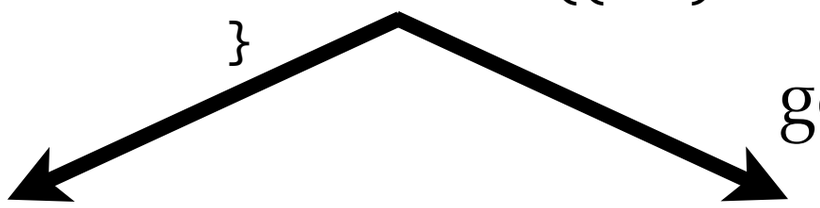
```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

reference semantics

```
Load  g4 1
Store g4 0
Load  g5 1
Store g5 2
Store g6 4
Load  g6 4
Load  g6 4
Load  g6 4
Store g6 1
Load  g4 0
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

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



reference semantics

gcc -O2 memory trace

```
Load  g4 1
Store g4 0
Load  g5 1
Store g5 2
Store g6 4
Load  g6 4
Load  g6 4
Load  g6 4
Store g6 1
Load  g4 0
```

```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

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

reference semantics

gcc -O2 memory trace

```
RaW* Load  g4 1
     Store g4 0
RaW* Load  g5 1
     Store g5 2
OW*  Store g6 4
RaW* Load  g6 4
RaR* Load  g6 4
RaR* Load  g6 4
     Store g6 1
RaW* Load  g4 0
```

```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

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

reference semantics

gcc -O2 memory trace

```
RaW* Load  g4 1
     Store g4 0
RaW* Load  g5 1
     Store g5 2
OW*  Store g6 4
RaW* Load  g6 4
RaR* Load  g6 4
RaR* Load  g6 4
     Store g6 1
RaW* Load  g4 0
```

```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

```
Init g3 0
```

```
void func_1(void){
    int *l8 = &g6;
```

Can match applying
only correct eliminations and reorderings

```
    *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

reference semantics

```
RaW* Load  g4 1
     Store g4 0
RaW* Load  g5 1
     Store g5 2
OW*  Store g6 4
RaW* Load  g6 4
RaR* Load  g6 4
RaR* Load  g6 4
     Store g6 1
RaW* Load  g4 0
```

gcc -O2 memory trace

```
Load  g5 1
Store g4 0
Store g6 1
Store g5 2
Load  g4 0
```

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

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

If we focus on the miscompiled initial example...

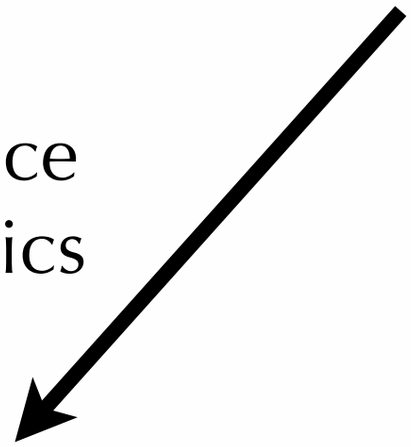
```
int a = 1;
int b = 0;
```

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

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

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

reference semantics



```
Load a 1
```

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

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

reference semantics

```
Load a 1
```

gcc -O2 memory trace

```
Load  a 1
Load  b 0
Store b 0
```

Cannot match some events ⟶ detect compiler bug

```
;
}
```

reference semantics

```
Load a 1
```

gcc -O2 memory trace

```
Load  a 1
Load  b 0
Store b 0
```

Applications

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

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

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

Remark: these bugs break the Posix thread model too.

All promptly fixed.

2. Checking compiler invariants

GCC internal invariant: never reorder with an atomic access

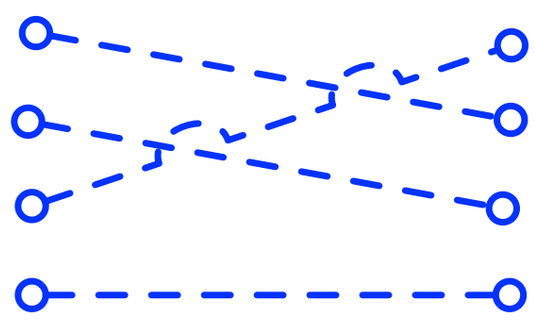
Baked this invariant into the tool and found a counterexample...

...not a bug, but fixed anyway

```
atomic_uint a;
int32_t g1, g2;

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

```
ALoad   a   0
ALoad   a   0
Load    g1  0
Store   g2  0
```



```
Load    g1  0
ALoad   a   0
ALoad   a   0
Store   g2  0
```

3. Detecting unexpected behaviours

```
uint16_t g

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

```
uint16_t g

g=0;
```

Correct or not?

3. Detecting unexpected behaviours

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

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

```
Load   g  0        )——?——(        Store  g  0
```

The introduced store cannot be observed by a non-racy context.

Still, *arguable if a compiler should do this or not*.

3. Detecting unexpected behaviours

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

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

```
Load   g   0        )——?——(        Store   g   0
```

False positives in Thread Sanitizer

The formalisation of the C11 memory model enables compiler testing… what else?

Proving the correctness of mappings for atomics

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

C/C++11 Operation	ARM implementation
Load Relaxed:	ldr
Load Consume:	ldr + preserve dependencies until next `kill_dependency` *OR* ldr; teq; beq; isb *OR* ldr; dmb
Load Acquire:	ldr; teq; beq; isb *OR* ldr; dmb
Load Seq Cst:	ldr; dmb
Store Relaxed:	str
Store Release:	dmb; str
Store Seq Cst:	dmb; str; dmb
Cmpxchg Relaxed (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop
Cmpxchg Acquire (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Cmpxchg Release (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop;
Cmpxchg AcqRel (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Cmpxchg SeqCst (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; dmb
Acquire Fence:	dmb
Release Fence:	dmb
AcqRel Fence:	dmb
SeqCst Fence:	dmb

Inform new optimisations

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

```
while (flag.load(acquire))
  {}
```

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

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

Take-up in Industrial Concurrency Community?

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

- Fixed up ISO C/C++11 Standard
  standard text and our maths in sync
- Fixed and verified C/C++11 to POWER compilation scheme
  compilers have to agree on this
- Clarified POWER and ARM architectural intent
  ongoing dialogues with the architects
- Found concurrency bugs in gcc, proposing optimisation schemes
  ongoing dialogue with gcc developers

The memory models of modern hardware are better understood

Programming languages attempt to specify and implement reasonable memory models.

Researchers and programmers are now interested in these problems.

The memory models of modern hardware are better understood

Still, many open problems...

problems.

The memory models of modern hardware are better understood

Still, many research opportunities!

problems.

Thank you! Questions?