Multicore Semantics and Programming

Tim Harris — Amazon

Peter Sewell — University of Cambridge

October – November, 2019

These Lectures

Part 1: Multicore Programming: Concurrent algorithms (Tim Harris, Amazon)

Concurrent programming: simple algorithms, correctness criteria, advanced synchronisation patterns, transactional memory.

Part 2: Multicore Semantics: the concurrency of multiprocessors and programming languages

What concurrency behaviour can you rely on? How can we specify it precisely in semantic models? Linking to usage, microarchitecture, experiment, and semantics. x86, IBM POWER, ARM, Java, C/C++11

Multicore Semantics

- Introduction
- Sequential Consistency
- x86 and the x86-TSO abstract machine
- x86 spinlock example
- Architectures
- Tests and Testing
- ...

Implementing Simple Mutual Exclusion, Naively

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

Implementing Simple Mutual Exclusion, Naively

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

repeated use?
thread symmetry (same code on each thread)?
performance?
fairness?
deadlock, global lock ordering, compositionality?

Let's Try...

```
./runSB.sh
```

Fundamental Question

What is the *behaviour of memory*?

...at the *programmer abstraction*

...when *observed by concurrent code*

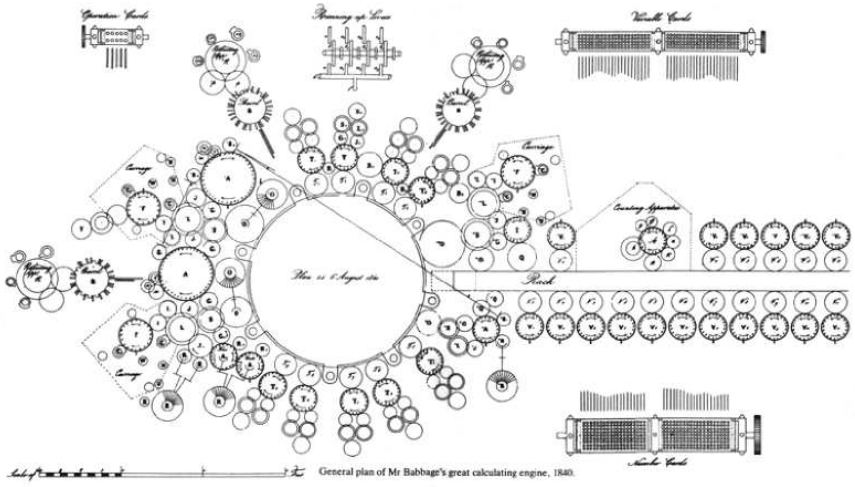
The abstraction of a *memory* goes back some time...

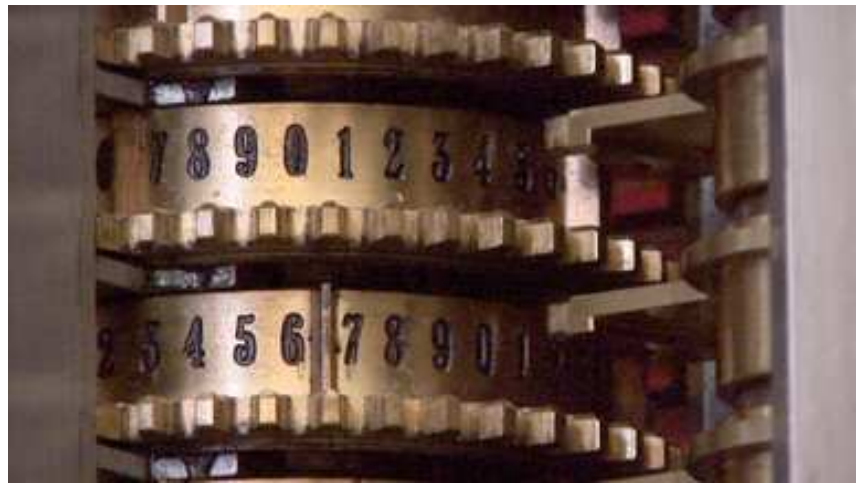
The calculating part of the engine may be divided into two portions

1st *The Mill in which all operations are performed*

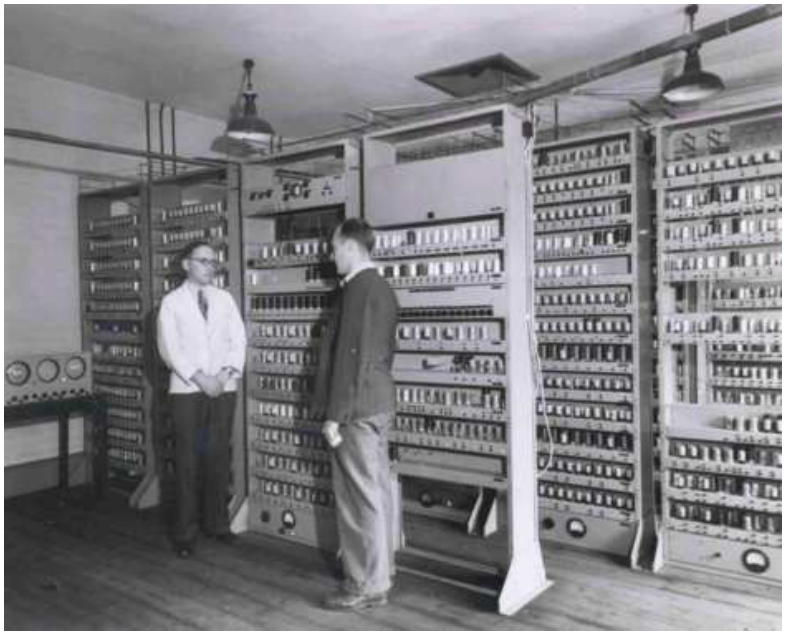
2nd *The Store in which all the numbers are originally placed and to which the numbers computed by the engine are returned.*

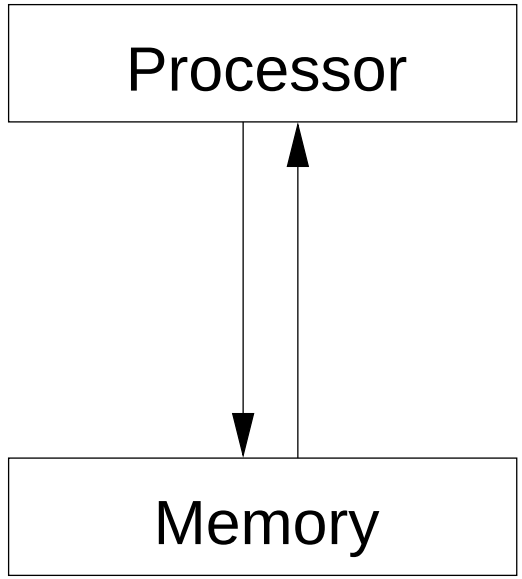
[Dec 1837, *On the Mathematical Powers of the Calculating Engine*, Charles Babbage]





The Golden Age, (1837–) 1945–1962



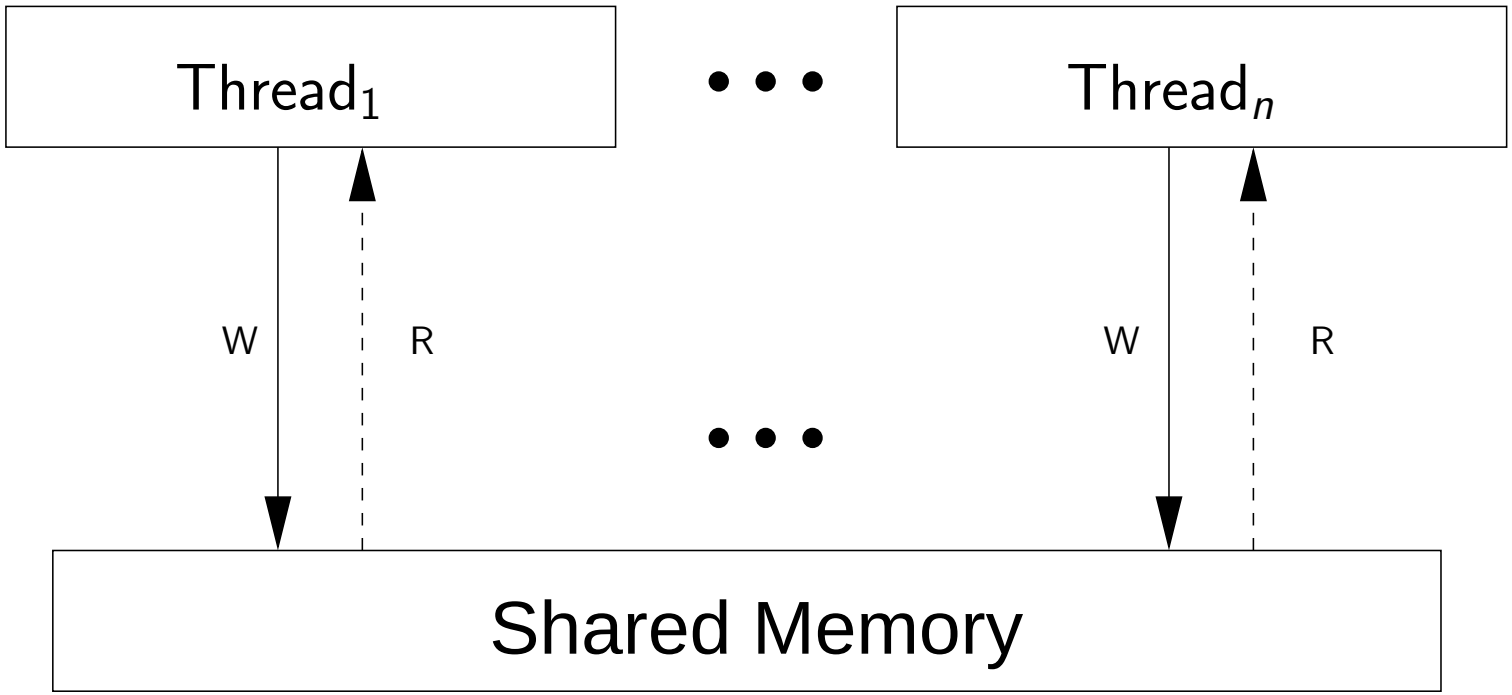


1962: First(?) Multiprocessor

BURROUGHS D825, 1962

"Outstanding features include truly modular hardware with parallel processing throughout"
FUTURE PLANS The complement of compiling languages is to be expanded."

... with Shared-Memory Concurrency



Multiprocessors, 1962–now

Niche multiprocessors since 1962

IBM System 370/158MP in 1972

Mass-market since 2005 (Intel Core 2 Duo).

Multiprocessors, 2019

Intel Xeon E7-8895 v3
36 hardware threads

ARM®

Commonly 8 hardware threads.

IBM®

IBM Power 8 server
(up to 1536 hardware threads)

Why now?

Exponential increases in transistor counts continuing — but not per-core performance

- energy efficiency (computation per Watt)
- limits of instruction-level parallelism

Concurrency finally mainstream — but how to understand, design, and program concurrent systems? Still very hard.

Concurrency everywhere

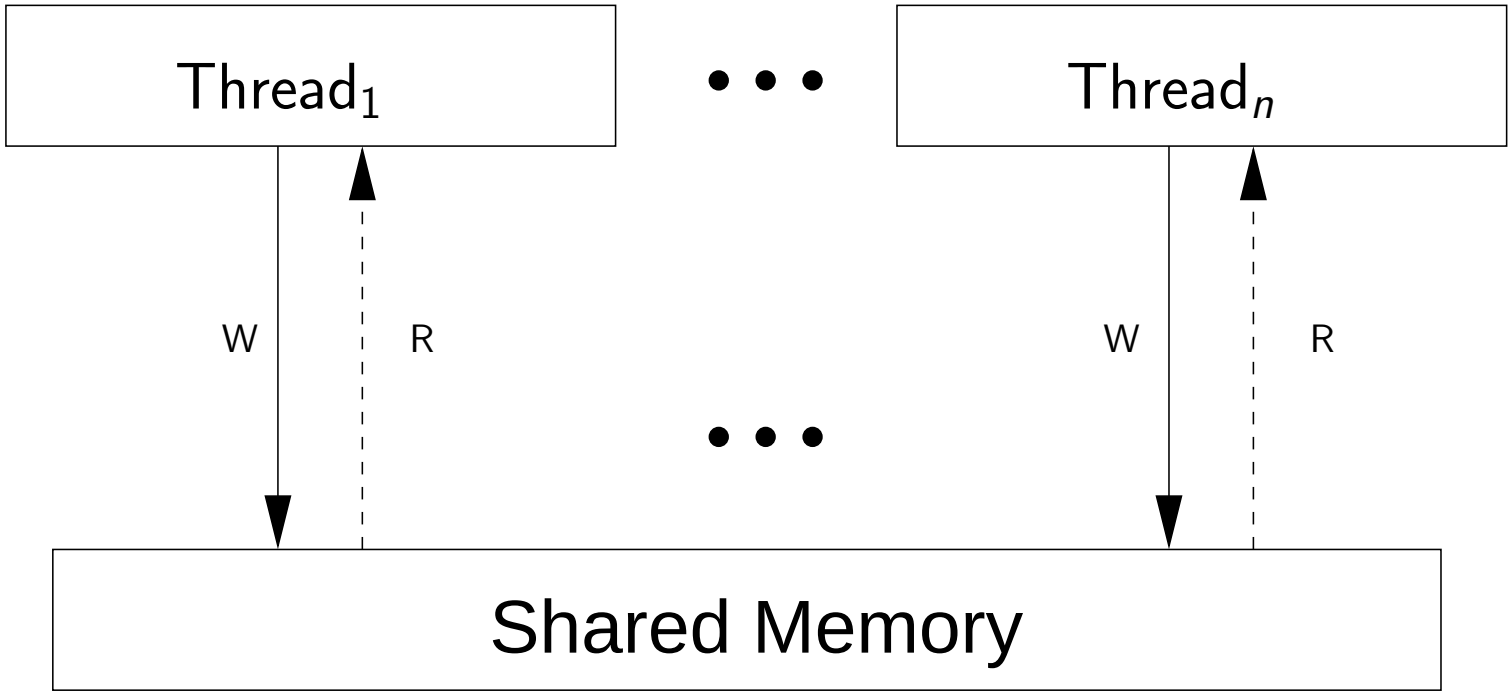
At many scales:

- intra-core
- multicore processors ← our focus
- ...and programming languages ← our focus
- GPU
- datacenter-scale
- internet-scale

explicit message-passing vs shared memory abstractions

Sequential Consistency

Our first model: Sequential Consistency



Multiple threads acting on a *sequentially consistent* (SC) shared memory:

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program *[Lamport, 1979]*

Defining an SC Semantics: SC memory

Define the state of an SC *memory* M to be a function from addresses x to integers n, with M_0 mapping all to 0. Let t range over thread ids.

Describe the interactions between memory and threads with *labels*:

label, l	::=		label
	|	t:W x=n	write
	|	t:R x=n	read
	|	t:τ	internal action (tau)

Define the behaviour of memory as a labelled transition system (LTS): the least set of (M, l, M') triples satisfying these rules.

$\boxed{M \xrightarrow{l} M'}$ memory M does l to become M'

$$\frac{M(x) = n}{M \xrightarrow{t:\mathrm{R}\,x=n} M} \quad \text{M\_read}$$

$$\frac{}{M \xrightarrow{t:\mathrm{W}\,x=n} M \oplus (x \mapsto n)} \quad \text{M\_write}$$

SC, said differently

In any trace $\vec{l} \in \mathrm{traces}(M_0)$ of M_0, i.e. any list of read and write events:

$$l_1,\ l_2,\ \ldots\ l_k$$

such that there are some $M_1, \ldots, M_k$ with

$$M_0 \xrightarrow{l_1} M_1 \xrightarrow{l_2} M_2 \ldots M_k,$$

each read reads from the value of the most recent preceding write to the same address, or from the initial state if there is no such write.

SC, said differently

Making that precise, define an alternative SC memory state L to be a list of labels, most recent at the head. Define *lookup* by:

$$\begin{array}{lcll} \mathit{lookup}\ x\ \mathtt{nil} & = & 0 & \text{initial state value} \\ \mathit{lookup}\ x\ (t{:}\mathrm{W}\,x'{=}n){::}L & = & n & \text{if } x = x' \\ \mathit{lookup}\ x\ l{::}L & = & \mathit{lookup}\ x\ L & \text{otherwise} \end{array}$$

$\boxed{L \xrightarrow{l} L'}$ list memory L does l to become L'

$$\frac{\mathit{lookup}\ x\ L = n}{L \xrightarrow{t:\mathrm{R}\,x=n} (t{:}\mathrm{R}\,x{=}n){::}L}\ \text{Lread}$$

$$\frac{}{L \xrightarrow{t:\mathrm{W}\,x=n} (t{:}\mathrm{W}\,x{=}n){::}L}\ \text{Lwrite}$$

Theorem (?)

M_0 and `nil` have the same traces

Extensional behaviour vs intensional structure

Extensionally, these models have the same behaviour

Intensionally, they have rather different structure – and neither is structured anything like a real hardware implementation.

In defining a model, we're principally concerned with the extensional behaviour: we want to precisely describe the set of allowed behaviours, as clearly as possible. But (see later) sometimes the intensional structure matters too, and we may also care about computability, performance, provability,...

SC, glued onto a tiny PL semantics

In those memory models:

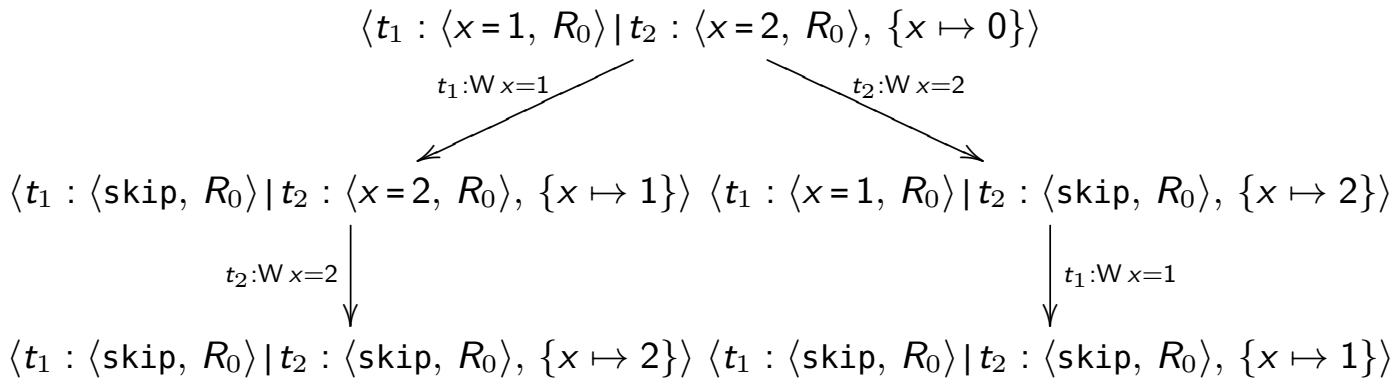
- the events within the trace of each thread were implicitly presumed to be ordered consistently with the *program order* (a control-flow unfolding) of that thread, and
- the values of writes were implicity presumed to be consistent with the thread-local computation specified by the program.

To make these things precise, we could combine the memory model with a threadwise semantics for a tiny concurrent language....

Example system transitions: SC Interleaving

All threads can read and write the shared memory.

Threads execute asynchronously – the semantics allows any interleaving of the thread transitions. Here there are two:



But each interleaving has a linear order of reads and writes to the memory. C.f. Lamport's

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

Back to the naive mutual exclusion example

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

Back to the naive mutual exclusion example

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$; $r_0 = y$	$y = 1$; $r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

Back to the naive mutual exclusion example

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$; $r_0 = y$	$y = 1$; $r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

In other words: is there a trace

$$\langle t_0 : \langle x = 1;\ r_0 = y,\ R_0 \rangle \mid t_1 : \langle y = 1;\ r_1 = x,\ R_0 \rangle,\ \{x \mapsto 0,\ y \mapsto 0\} \rangle$$
$$\xrightarrow{l_1} \ldots \xrightarrow{ln}$$
$$\langle t_0 : \langle \texttt{skip},\ R_0' \rangle \mid t_1 : \langle \texttt{skip},\ R_1' \rangle,\ M' \rangle$$

such that $R_0'(r_0) = 0$ and $R_1'(r_1) = 0$?

Back to the naive mutual exclusion example

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$; $r_0 = y$	$y = 1$; $r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

In other words: is there a trace

$$\langle t_0 : \langle x = 1;\ r_0 = y,\ R_0 \rangle \mid t_1 : \langle y = 1;\ r_1 = x,\ R_0 \rangle,\ \{x \mapsto 0,\ y \mapsto 0\} \rangle$$
$$\xrightarrow{l_1} \ldots \xrightarrow{ln}$$
$$\langle t_0 : \langle \texttt{skip},\ R'_0 \rangle \mid t_1 : \langle \texttt{skip},\ R'_1 \rangle,\ M' \rangle$$

such that $R'_0(r_0) = 0$ and $R'_1(r_1) = 0$?

In this semantics: no

Back to the naive mutual exclusion example

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$; $r_0 = y$	$y = 1$; $r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

In other words: is there a trace

$$\langle t_0 : \langle x = 1;\ r_0 = y,\ R_0 \rangle \mid t_1 : \langle y = 1;\ r_1 = x,\ R_0 \rangle,\ \{x \mapsto 0,\ y \mapsto 0\} \rangle$$
$$\xrightarrow{l_1} \ldots \xrightarrow{ln}$$
$$\langle t_0 : \langle \texttt{skip},\ R_0' \rangle \mid t_1 : \langle \texttt{skip},\ R_1' \rangle,\ M' \rangle$$

such that $R_0'(r_0) = 0$ and $R_1'(r_1) = 0$?

In this semantics: no

But on x86 hardware, we saw it!

Options

1. the hardware is busted (either this instance or in general)
2. the program is bad
3. the model is wrong

Options

1. the hardware is busted (either this instance or in general)
2. the program is bad
3. the model is wrong

SC is not a good model of x86 (or of Power, ARM, Sparc, Itanium...)

Options

1. the hardware is busted (either this instance or in general)
2. the program is bad
3. the model is wrong

SC is not a good model of x86 (or of Power, ARM, Sparc, Itanium...)

Even though most work on verification, and many programmers, assume SC...

Similar Options

1. the hardware is busted
2. the compiler is busted
3. the program is bad
4. the model is wrong

Similar Options

1. the hardware is busted
2. the compiler is busted
3. the program is bad
4. the model is wrong

SC is also not a good model of C, C++, Java,...

Similar Options

1. the hardware is busted
2. the compiler is busted
3. the program is bad
4. the model is wrong

SC is also not a good model of C, C++, Java,...

Even though most work on verification, and many programmers, assume SC...

What's going on? Relaxed Memory

Multiprocessors and compilers incorporate many performance optimisations

(hierarchies of cache, load and store buffers, speculative execution, cache protocols, common subexpression elimination, etc., etc.)

These are:

- unobservable by single-threaded code
- sometimes observable by concurrent code

Upshot: they provide only various *relaxed* (or *weakly consistent*) memory models, not sequentially consistent memory.

New problem?

No: IBM System 370/158MP in 1972, already non-SC

But still a research question!

The mainstream architectures and languages are key interfaces

...but it's been very unclear exactly how they behave.

More fundamentally: it's been (and in significant ways still is) unclear how we can specify that precisely.

As soon as we can do that, we can build above it: explanation, testing, emulation, static/dynamic analysis, model-checking, proof-based verification,....

x86

A Cautionary Tale

Intel 64/IA32 and AMD64 - before Aug. 2007 (Era of Vagueness)

'Processor Ordering' model, informal prose

Example: Linux Kernel mailing list, Nov–Dec 1999 (143 posts)

Keywords: speculation, ordering, cache, retire, causality

A one-instruction programming question, a microarchitectural debate!

1. spin_unlock() Optimization On Intel

20 Nov 1999 - 7 Dec 1999 (143 posts) Archive Link: "spin_unlock optimizatio
Topics: BSD: FreeBSD, SMP
People: Linus Torvalds, Jeff V. Merkey, Erich Boleyn, Manfred Spraul, Peter S
son, Ingo Molnar

Manfred Spraul thought he'd found a way to shave spin_unlock() down fron
22 ticks for the "`lock; btrl $0,%0`" asm code, to 1 tick for a simple "`movl`
instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% sp
in a benchmark test, making the optimization very valuable. Ingo also added t
same optimization cropped up in the FreeBSD mailing list a few days previous
Linus Torvalds poured cold water on the whole thing, saying:

> It does NOT WORK!
> Let the FreeBSD people use it, and let them get faster timings. They
> crash, eventually.
> The window may be small, but if you do this, then suddenly spinlo
> aren't reliable any more.
> The issue is not writes being issued in-order (although all the Intel C
> books warn you NOT to assume that in-order write behaviour - I be
> won't be the case in the long run).
> The issue is that you _have_ to have a serializing instruction in orde
> make sure that the processor doesn't re-order things around the unloc
> For example, with a simple write, the CPU can legally delay a read t
> happened inside the critical region (maybe it missed a cache line), and
> a stale value for any of the reads that _should_ have been serialized by
> spinlock.
> Note that I actually thought this was a legal optimization, and for a w
> I had this in the kernel. It crashed. In random ways.
> Note that the fact that it does not crash now is quite possibly becaus
> either
>
> - we have a lot less contention on our spinlocks these days. Th
> might hide the problem, because the _spinlock_ will be fine (t
> cache coherency still means that the spinlock itself works fine

Resolved only by appeal to an oracle:

don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replie Linus, pointing out a possible misconception in his proposed exploit. Regarding 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 nus:

> Everybody has convinced me that yes, the Intel ordering rules _are_ strong enough that all of this really is legal, and that's what I wanted. I've gotten sane explanations for why serialization (as opposed to just the simple locked access) is required for the lock() side but not the unlock() side, and that lack of symmetry was what bothered me the most.
>
> Oliver made a strong case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.
>
> Thanks, guys, we'll be that much faster due to this..

Erich then argued that serialization was not required for the lock() side either, after a long and interesting discussion he apparently was unable to win people ov In fact, as Peter Samuelson pointed out to me after KT publication (and many th to him for it):

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

IWP and AMD64, Aug. 2007/Oct. 2008 (Era of Causality)

Intel published a white paper (IWP) defining 8 informal-prose principles, e.g.

P1. Loads are not reordered with older loads
P2. Stores are not reordered with older stores

supported by 10 *litmus tests* illustrating allowed or forbidden behaviours, e.g.

Message Passing (MP)

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV EAX←[y] (read y=1)
MOV [y]←1 (write y=1)	MOV EBX←[x] (read x=0)
Forbidden Final State: Thread 1:EAX=1 ∧ Thread 1:EBX=0	

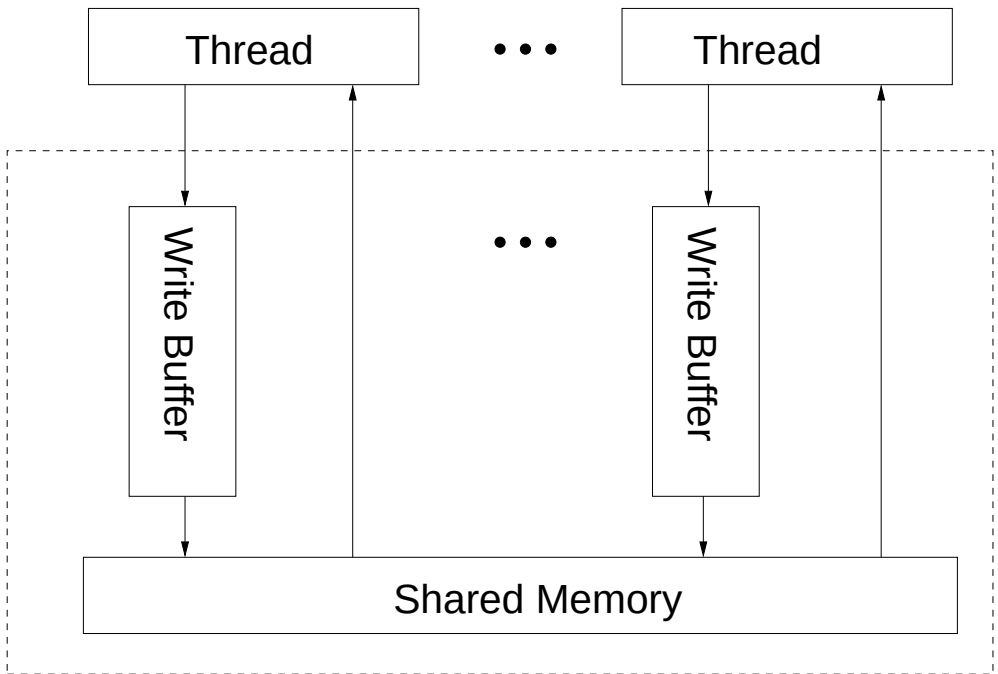
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y=0)	MOV EBX←[x]	(read x=0)
Allowed Final State: Thread 0:EAX=0 ∧ Thread 1:EBX=0			

but not with older stores to the same location

Store Buffer (SB)

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y=0)	MOV EBX←[x]	(read x=0)
Allowed Final State: Thread 0:EAX=0 ∧ Thread 1:EBX=0			

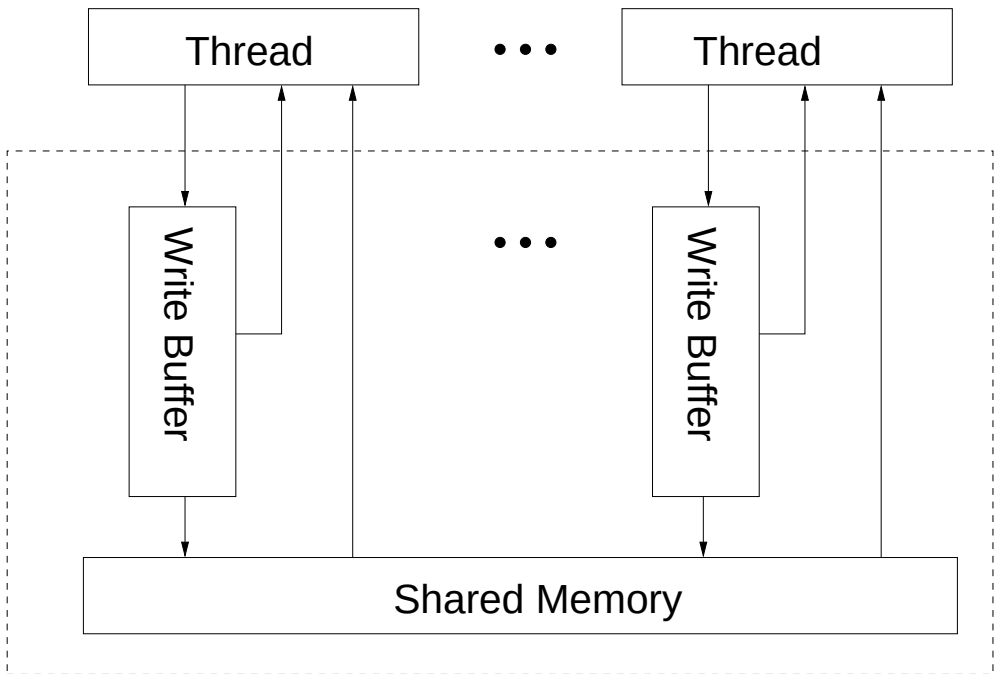


Litmus Test 2.4. Intra-processor forwarding is allowed

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[x]	(read x=1)	MOV ECX←[y]	(read y=1)
MOV EBX←[y]	(read y=0)	MOV EDX←[x]	(read x=0)
Allowed Final State: Thread 0:EBX=0 ∧ Thread 1:EDX=0 Thread 0:EAX=1 ∧ Thread 1:ECX=1			

Litmus Test 2.4. Intra-processor forwarding is allowed

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[x]	(read x=1)	MOV ECX←[y]	(read y=1)
MOV EBX←[y]	(read y=0)	MOV EDX←[x]	(read x=0)
Allowed Final State: Thread 0:EBX=0 ∧ Thread 1:EDX=0 ∧ Thread 0:EAX=1 ∧ Thread 1:ECX=1			



Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

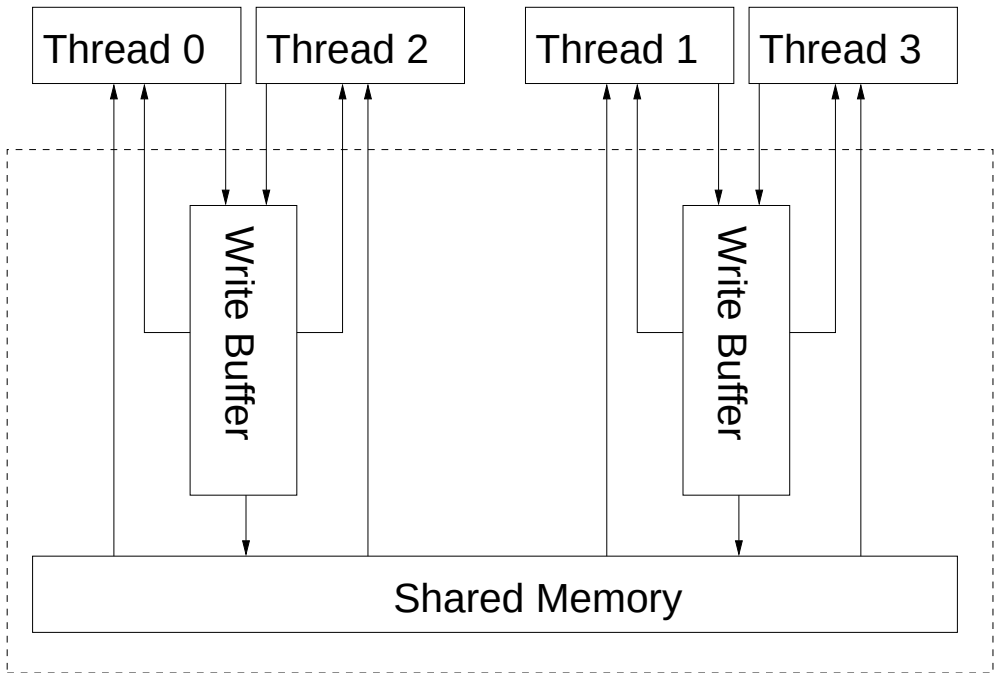
Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

Microarchitecturally plausible? yes, e.g. with shared store buffers



Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

- AMD3.14: Allowed
- IWP: ???
- Real hardware: unobserved
- Problem for normal programming: ?

Weakness: adding memory barriers does not recover SC, which was assumed in a Sun implementation of the JMM

Problem 2: Ambiguity

P1–4. ...may be reordered with...

P5. Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation

Write-to-Read Causality (WRC) (Litmus Test 2.5)

Thread 0	Thread 1	Thread 2
MOV [x]←1 (W x=1)	MOV EAX←[x] (R x=1)	MOV EBX←[y] (R y=1)
	MOV [y]←1 (W y=1)	MOV ECX←[x] (R x=0)
Forbidden Final State: Thread 1:EAX=1 ∧ Thread 2:EBX=1 ∧ Thread 2:ECX=0		

Problem 3: Unsoundness!

Example from Paul Loewenstein:
n6

Thread 0		Thread 1	
MOV [x]←1	(a:W x=1)	MOV [y]←2	(d:W y=2)
MOV EAX←[x]	(b:R x=1)	MOV [x]←2	(e:W x=2)
MOV EBX←[y]	(c:R y=0)		
Allowed Final State: Thread 0:EAX=1 $\wedge$ Thread 0:EBX=0 $\wedge$ x=1			

Observed on real hardware, but not allowed by (any interpretation we can make of) the IWP 'principles', if one reads 'ordered' as referring to a single per-execution partial order.

(can see allowed in store-buffer microarchitecture)

Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0		Thread 1	
MOV [x]←1	(a:W x=1)	MOV [y]←2	(d:W y=2)
MOV EAX←[x]	(b:R x=1)	MOV [x]←2	(e:W x=2)
MOV EBX←[y]	(c:R y=0)		
Allowed Final State: Thread 0:EAX=1 ∧ Thread 0:EBX=0 ∧ x=1			

In the view of Thread 0:
a→b by *P4: Reads may [...] not be reordered with older writes to the same location.*
b→c by *P1: Reads are not reordered with other reads.*
c→d, otherwise c would read 2 from d
d→e by *P3. Writes are not reordered with older reads.*
so a:Wx=1 → e:Wx=2

But then that should be respected in the final state, by *P6: In a multiprocessor system, stores to the same location have a total order*, and it isn't.

(can see allowed in store-buffer microarchitecture)

Problem 3: Unsoundness!

Example from Paul Loewenstein:
n6

Thread 0	Thread 1
MOV [x]←1 (a:W x=1) MOV EAX←[x] (b:R x=1) MOV EBX←[y] (c:R y=0)	MOV [y]←2 (d:W y=2) MOV [x]←2 (e:W x=2)
Allowed Final State: Thread 0:EAX=1 ∧ Thread 0:EBX=0 ∧ x=1	

Observed on real hardware, but not allowed by (any interpretation we can make of) the IWP 'principles'.

(can see allowed in store-buffer microarchitecture)

So spec unsound (and also our POPL09 model based on it).

Intel SDM and AMD64, Nov. 2008 – Oct. 2015

Intel SDM rev. 29–55 and AMD 3.17–3.25

Not unsound in the previous sense

Explicitly exclude IRIW, so not weak in that sense. New principle:

> Any two stores are seen in a consistent order by processors other than those performing the stores

But, still ambiguous, and the *view by those processors is left entirely unspecified*

Intel:

`https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf`

(rev. 35 on 6/10/2010, rev. 55 on 3/10/2015, rev. 70 on 1/11/2019).
See especially SDM Vol. 3A, Ch. 8, Sections 8.1–8.3

AMD:

`http://support.amd.com/TechDocs/24593.pdf`

(rev. 3.17 on 6/10/2010, rev. 3.25 on 3/10/2015, rev. 3.32 on 1/11/2019).
See especially APM Vol. 2, Ch. 7, Sections 7.1–7.2

Inventing a Usable Abstraction

Have to be:

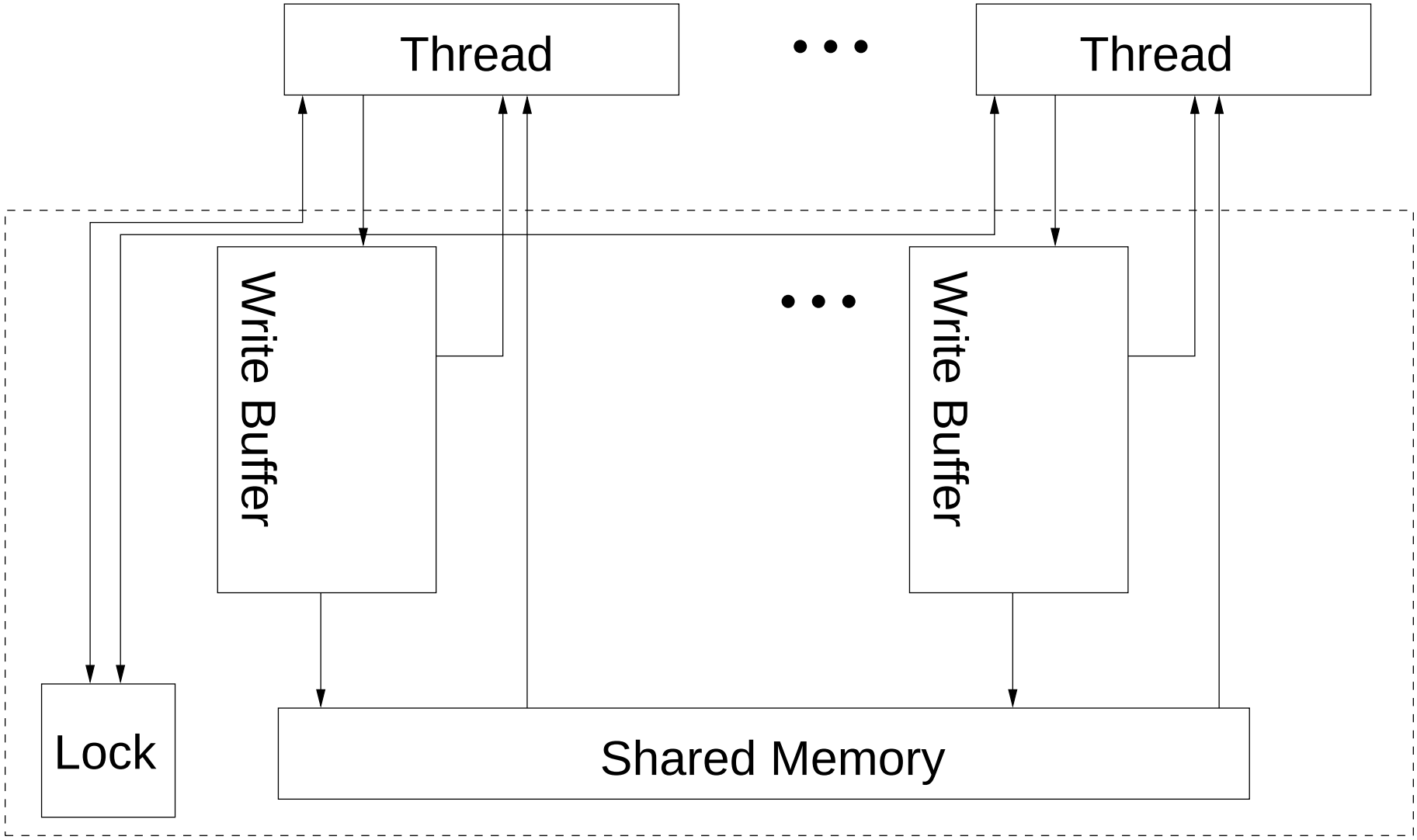
- Unambiguous
- Sound w.r.t. experimentally observable behaviour
- Easy to understand
- Consistent with what we know of vendors intentions
- Consistent with expert-programmer reasoning

Key facts:

- Store buffering (with forwarding) is observable
- IRIW is not observable, and is forbidden by the recent docs
- Various other reorderings are not observable and are forbidden

These suggest that x86 is, in practice, like SPARC TSO.

x86-TSO Abstract Machine



x86-TSO Abstract Machine

As for Sequential Consistency, we separate the programming language (here, really the *instruction semantics*) and the x86-TSO *memory model*.

(the memory model describes the behaviour of the stuff in the dotted box)

Put the instruction semantics and abstract machine in parallel, exchanging read and write messages (and lock/unlock messages).

x86-TSO Abstract Machine: Interface

Labels

l	::=	t:W x=v	a write of value v to address x by thread t
	|	t:R x=v	a read of v from x by t
	|	t:τ	an internal action of the thread
	|	t:$\tau_{x=v}$	an internal action of the abstract machine, moving $x = v$ from the write buffer on t to shared memory
	|	t:B	an MFENCE memory barrier by t
	|	t:L	start of an instruction with LOCK prefix by t
	|	t:U	end of an instruction with LOCK prefix by t

where

- t is a hardware thread id, of type *tid*,
- x and y are memory addresses, of type *addr*
- v and w are machine words, of type *value*

x86-TSO Abstract Machine: Machine States

An *x86-TSO abstract machine state* m is a record

$$m : \langle\!\lbrack \begin{array}{l} M : \mathit{addr} \to \mathit{value}; \\ B : \mathit{tid} \to (\mathit{addr} \times \mathit{value})\ \mathit{list}; \\ L : \mathit{tid}\ \mathit{option} \rbrack\!\rangle \end{array}$$

Here:

- $m.M$ is the shared memory, mapping addresses to values
- $m.B$ gives the store buffer for each thread, most recent at the head
- $m.L$ is the global machine lock indicating when a thread has exclusive access to memory

Write m_0 for the initial state with $m.M = M_0$, $s.B$ empty for all threads, and $m.L = \mathit{None}$ (lock not taken).

x86-TSO Abstract Machine: Auxiliary Definitions

Say there are *no pending* writes in t's buffer $m.B(t)$ for address x if there are no (x, v) elements in $m.B(t)$.

Say t is *not blocked* in machine state s if either it holds the lock ($m.L = \text{Some}\, t$) or the lock is not held ($m.L = \text{None}$).

x86-TSO Abstract Machine: Behaviour

RM: Read from memory

$$\frac{\begin{array}{l}\mathrm{not\_blocked}(m, t)\\ m.M(x) = v\\ \mathrm{no\_pending}(m.B(t), x)\end{array}}{m \xrightarrow{t:\mathrm{R}\,x=v} m}$$

Thread t can read v from memory at address x if t is not blocked, the memory does contain v at x, and there are no writes to x in t's store buffer.

x86-TSO Abstract Machine: Behaviour

RB: Read from write buffer

$$\frac{\begin{array}{l}\mathrm{not\_blocked}(m, t) \\ \exists b_1\ b_2.\ m.B(t) = b_1 +\!\!+ [(x, v)] +\!\!+ b_2 \\ \mathrm{no\_pending}(b_1, x)\end{array}}{m \xrightarrow{t:\mathrm{R}\,x=v} m}$$

Thread t can read v from its store buffer for address x if t is not blocked and has v as the newest write to x in its buffer;

x86-TSO Abstract Machine: Behaviour

WB: Write to write buffer

$$m \xrightarrow{t:\mathrm{W}\,x=v} m \oplus \langle\!\langle B := m.B \oplus (t \mapsto ([(x, v)] +\!\!+ m.B(t))) \rangle\!\rangle$$

Thread t can write v to its store buffer for address x at any time;

x86-TSO Abstract Machine: Behaviour

WM: Write from write buffer to memory

$$\frac{\begin{array}{c}\mathrm{not\_blocked}(m, t) \\ m.B(t) = b +\!\!+ [(x, v)]\end{array}}{m \xrightarrow{t:\tau\ x=v} m \oplus \langle\!\langle M := m.M \oplus (x \mapsto v) \rangle\!\rangle \oplus \langle\!\langle B := m.B \oplus (t \mapsto b) \rangle\!\rangle}$$

If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread

x86-TSO Abstract Machine: Behaviour

...rules for lock, unlock, and mfence later

Notation Reference

Some and None construct optional values

$(\cdot, \cdot)$ builds tuples

$[\,]$ builds lists

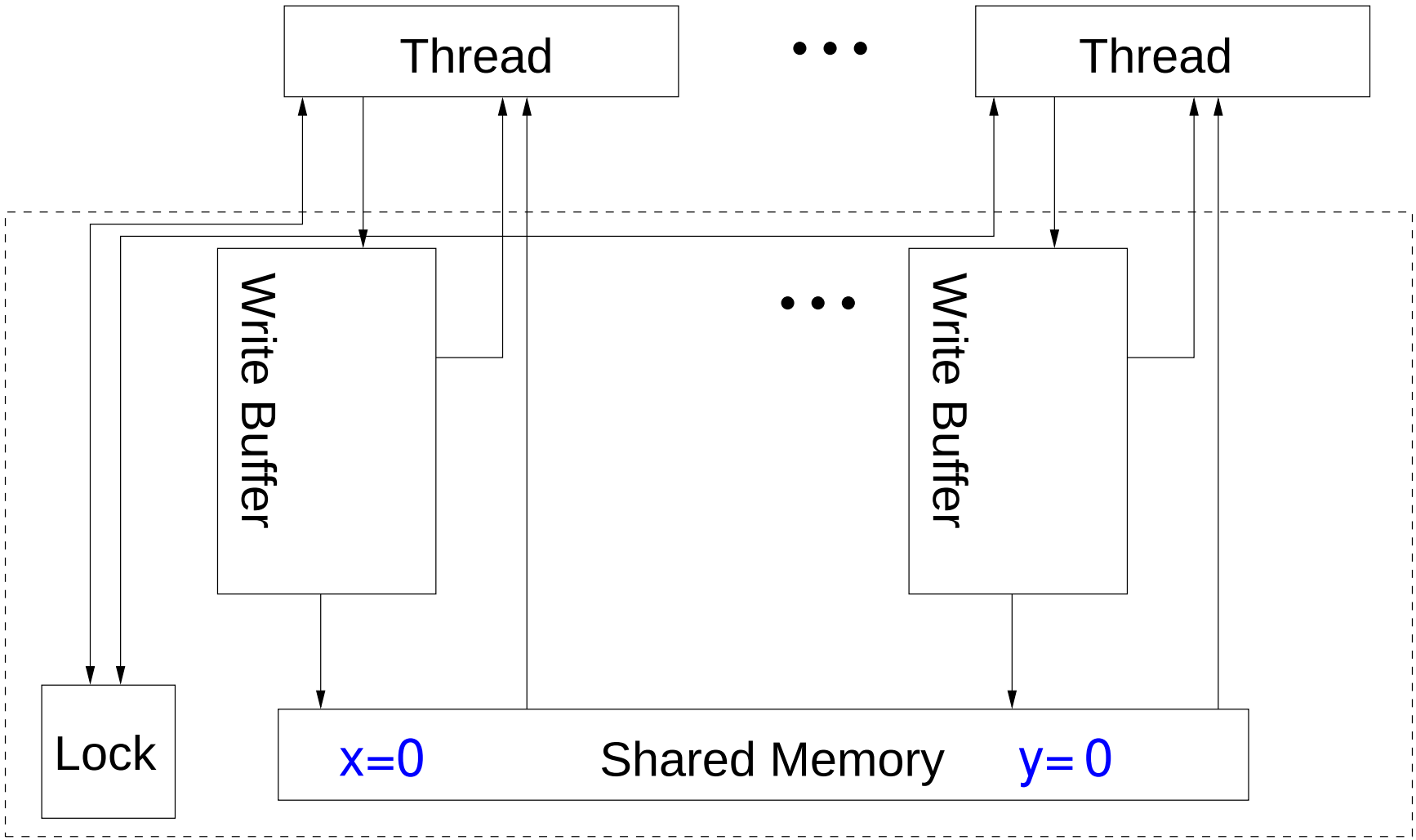
$+\!\!+$ appends lists

$\cdot \oplus \langle\!\![\cdot := \cdot]\!\!\rangle$ updates records

$\cdot(\cdot \mapsto \cdot)$ updates functions.

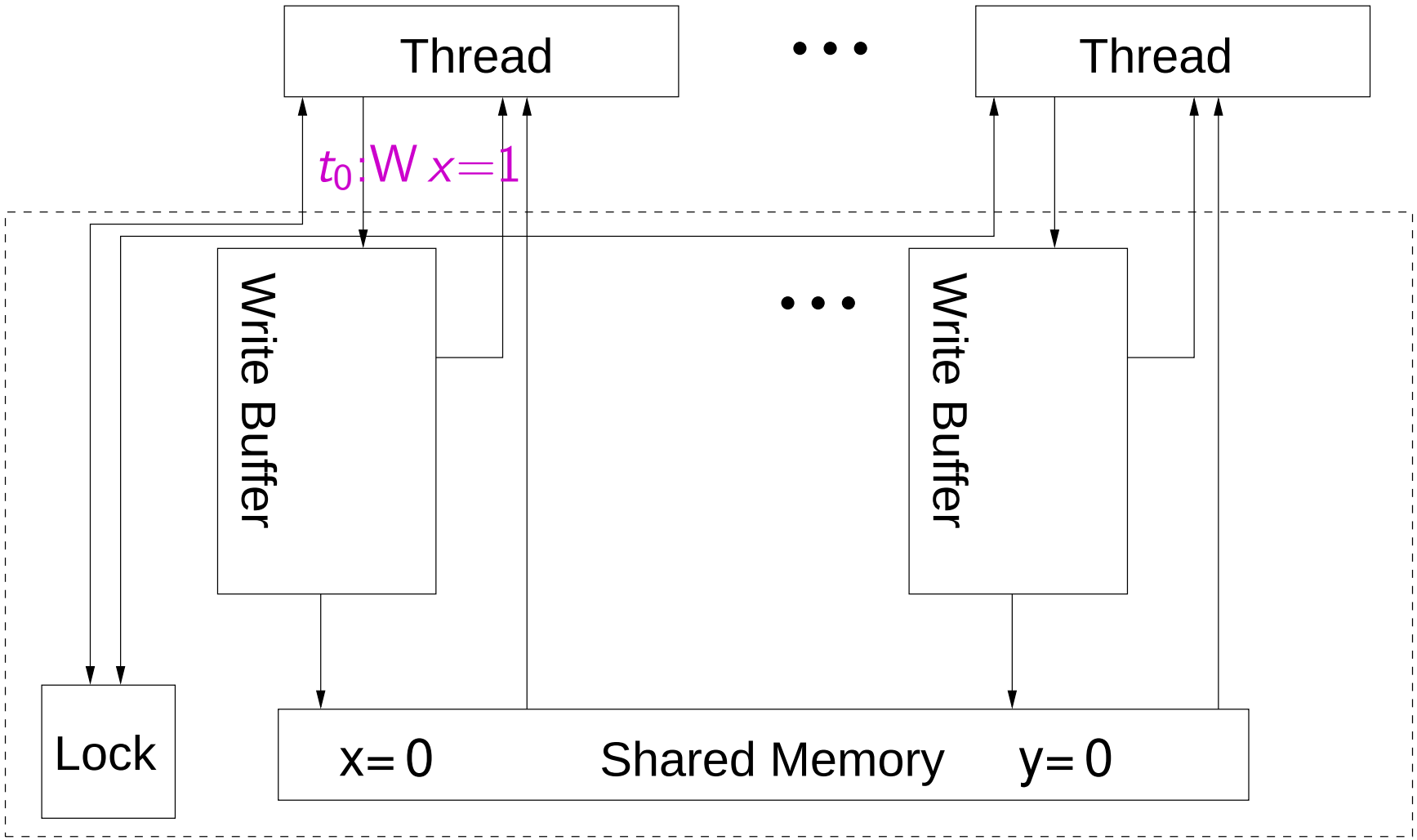
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



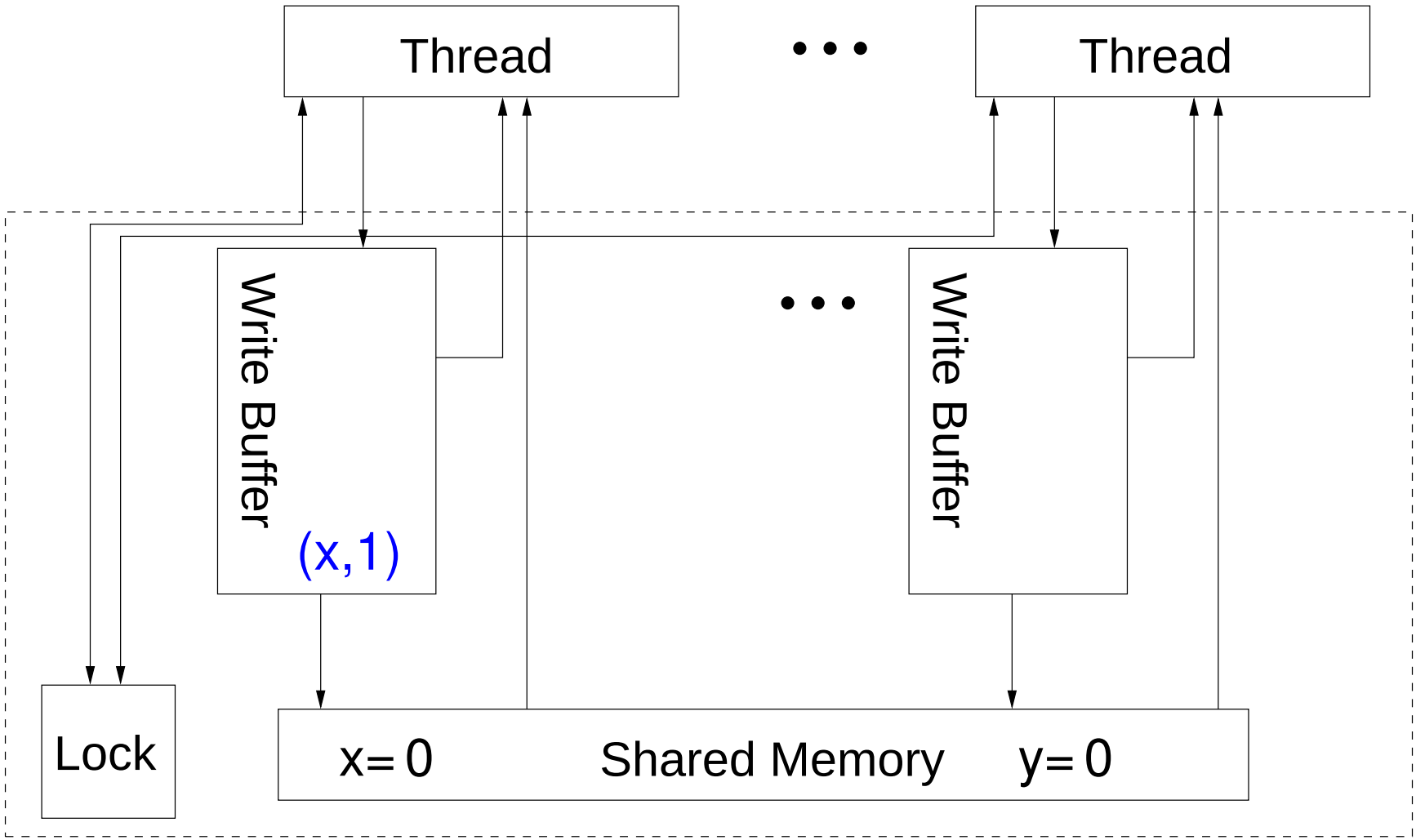
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



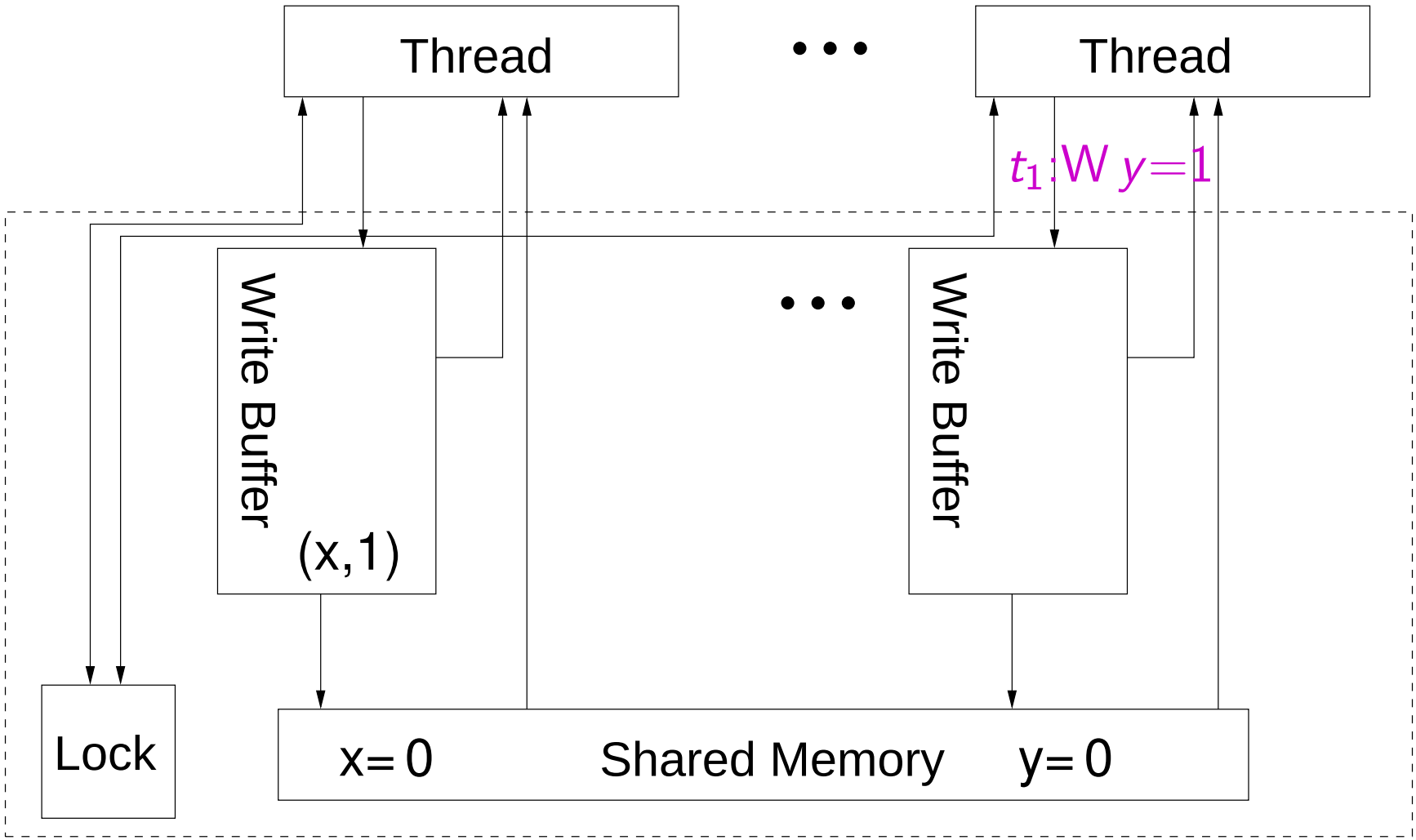
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



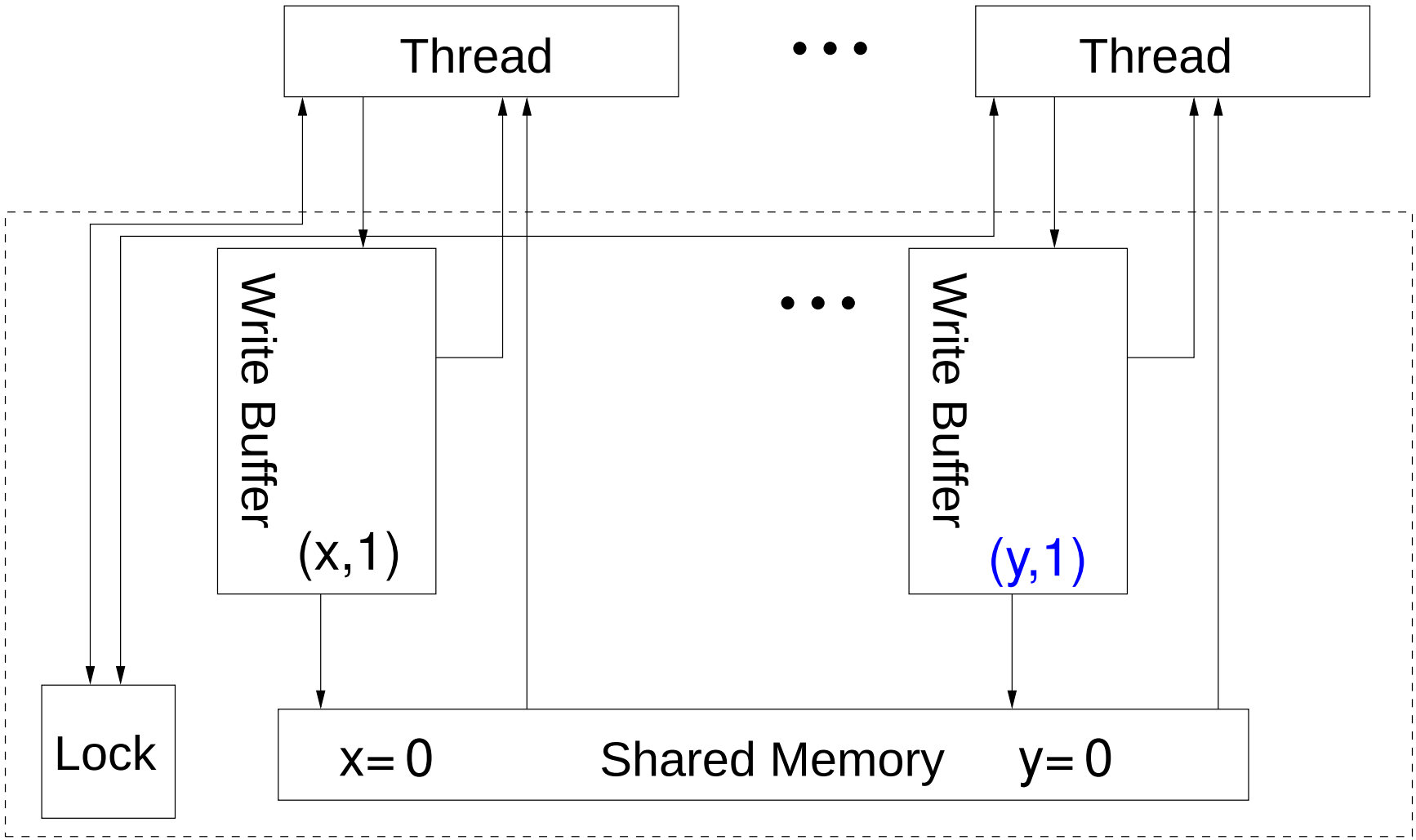
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



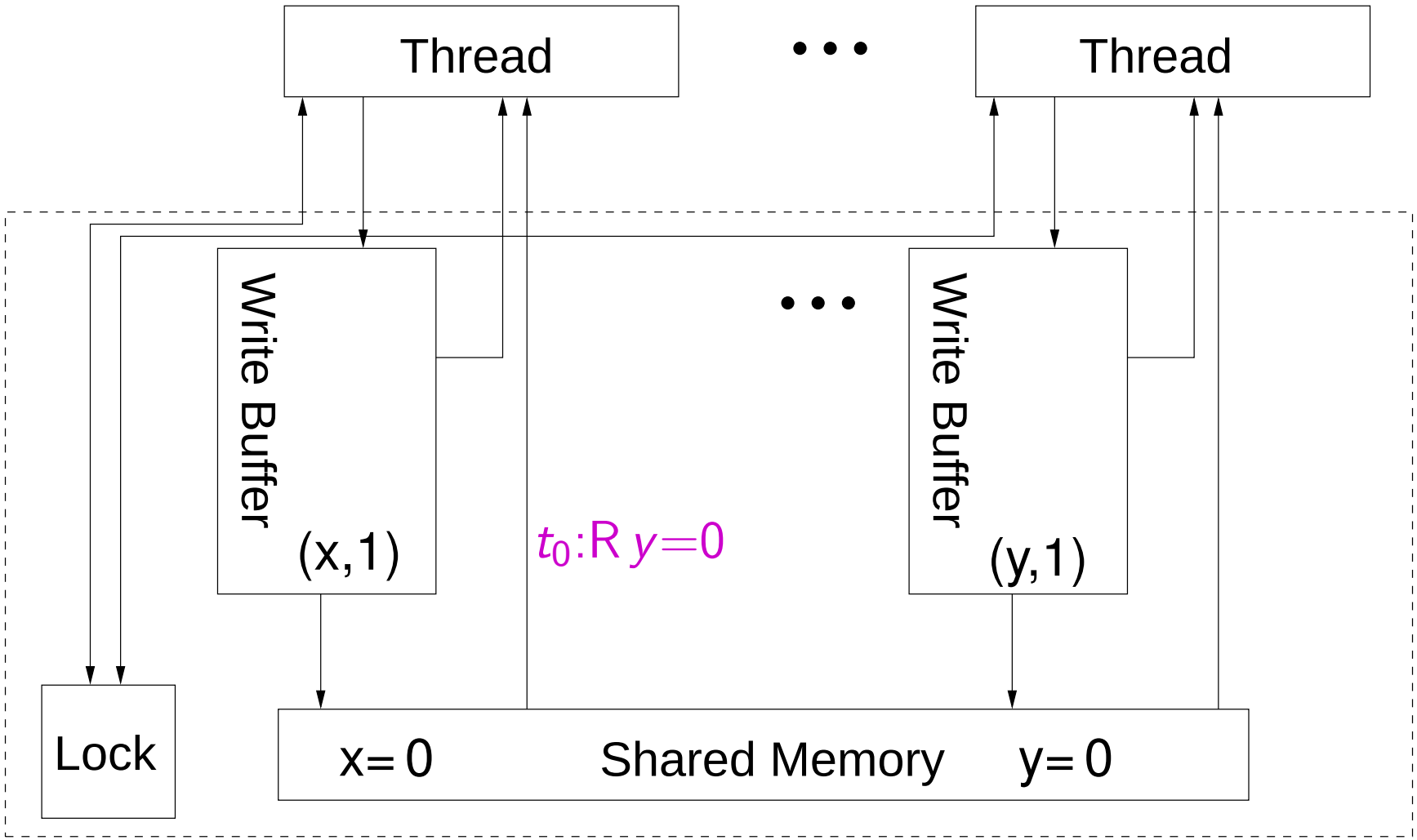
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



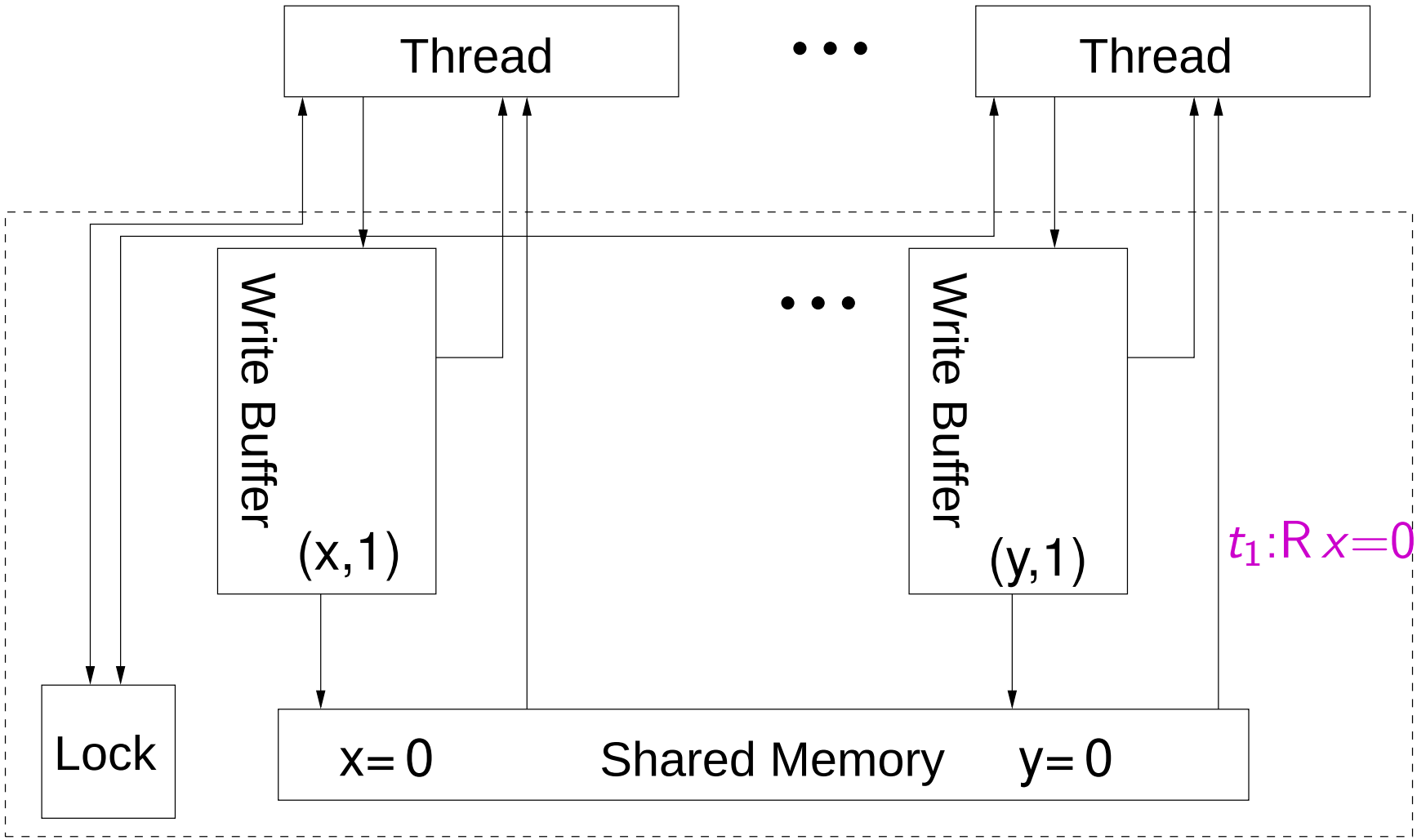
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



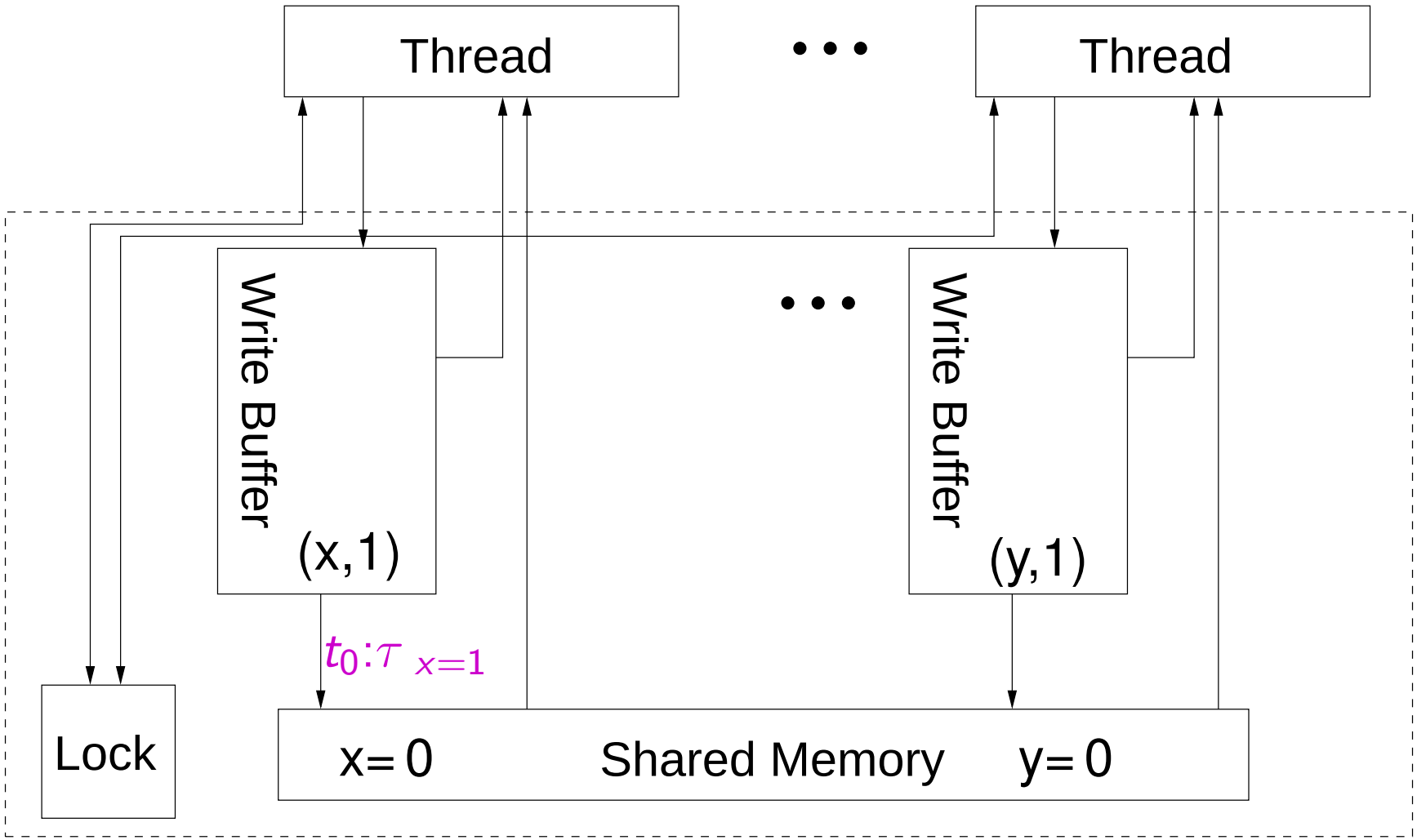
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



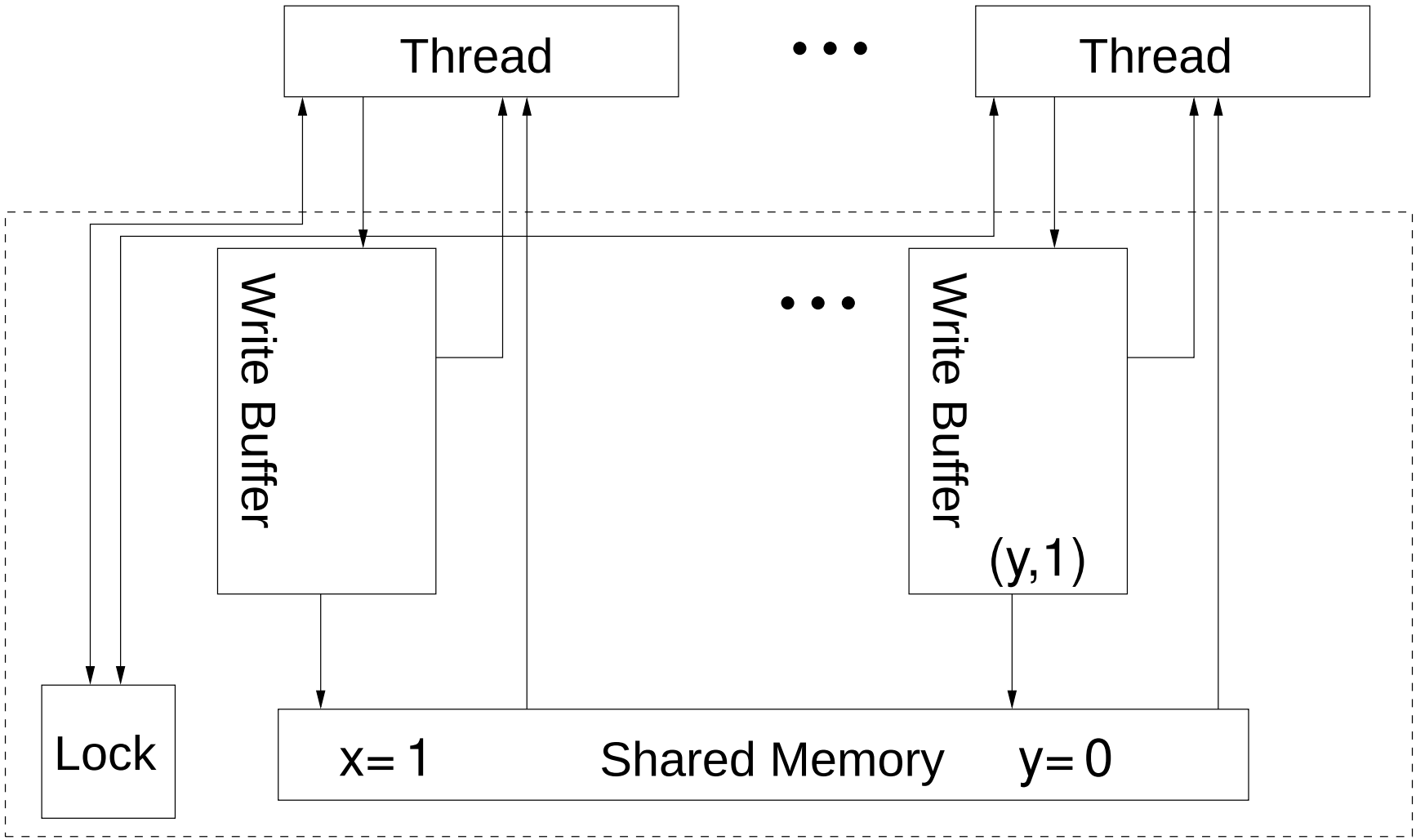
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



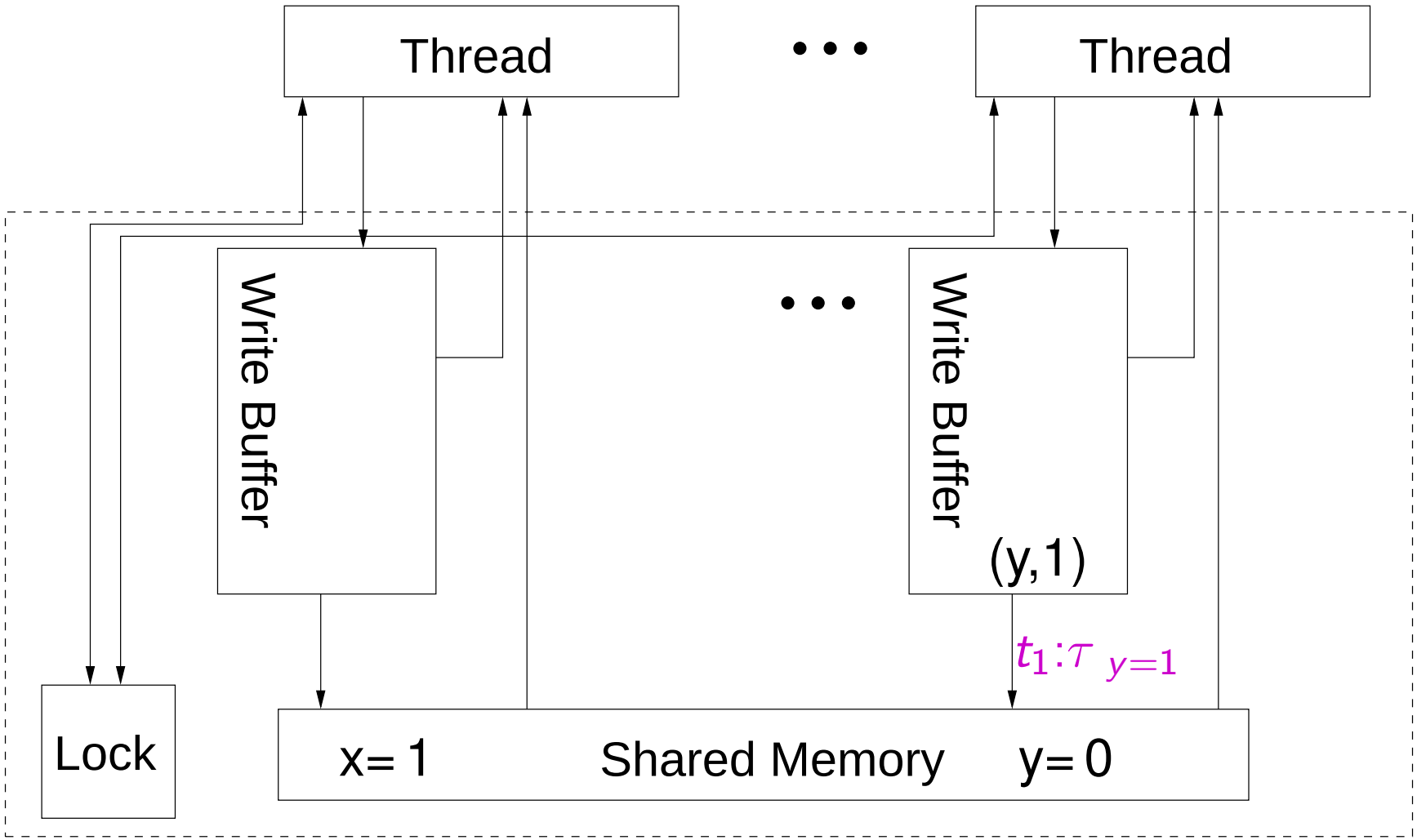
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



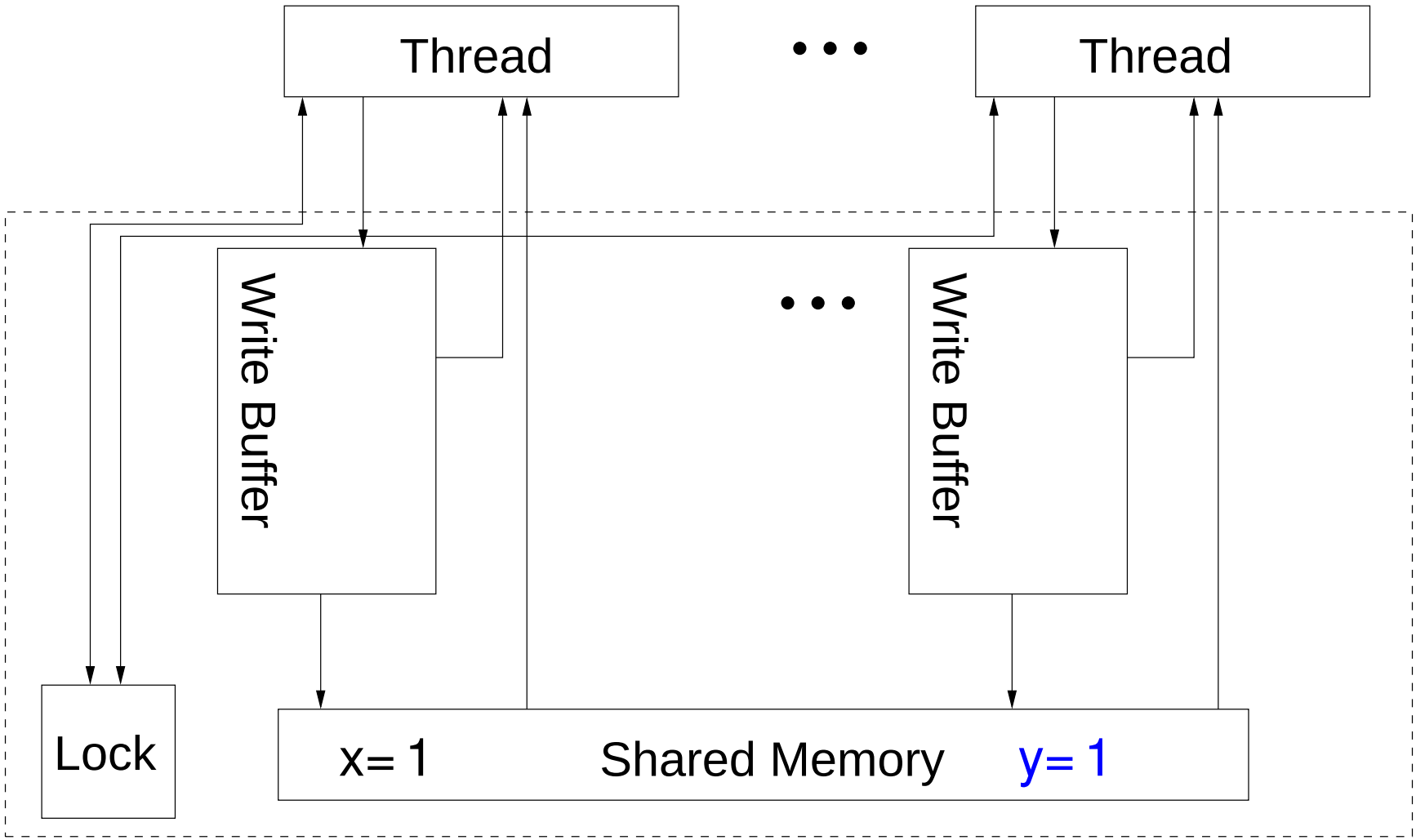
First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



Strengthening the model: the MFENCE memory barrier

MFENCE: an x86 assembly instruction

...waits for local write buffer to drain (or forces it – is that an observable distinction?)

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MFENCE	MFENCE
MOV EAX←[y] (read y=0)	MOV EBX←[x] (read x=0)
Forbidden Final State: Thread 0:EAX=0 ∧ Thread 1:EBX=0	

NB: no inter-thread synchronisation

x86-TSO Abstract Machine: Behaviour

B: Barrier

$$\frac{m.B(t) = [\,]}{m \xrightarrow{t:\mathrm{B}} m}$$

If t's store buffer is empty, it can execute an MFENCE (otherwise the MFENCE blocks until that becomes true).

Does MFENCE restore SC?

For any process P, define $\mathrm{insert\_fences}(P)$ to be the process with all $s_1\,;\,s_2$ replaced by $s_1\,;\,\texttt{mfence}\,;\,s_2$ (formally define this recursively over statements, threads, and processes).

For any trace $l_1, \ldots, l_k$ of an x86-TSO system state, define $\mathrm{erase\_flushes}(l_1, \ldots, l_k)$ to be the trace with all $t{:}\tau_{\,x=v}$ labels erased (formally define this recursively over the list of labels).

Theorem (?)

For all processes P,

$$\mathrm{traces}(\langle P,\, m_0\rangle) = \mathrm{erase\_flushes}(\mathrm{traces}(\langle \mathrm{insert\_fences}(P),\, m_{\mathrm{tso}0}\rangle))$$

Adding Read-Modify-Write instructions

x86 is not RISC – there are many instructions that read and write memory, e.g.

Thread 0	Thread 1
INC x	INC x

Adding Read-Modify-Write instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Adding Read-Modify-Write instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Thread 0	Thread 1
LOCK;INC x	LOCK;INC x
Forbidden Final State: [x]=1	

Adding Read-Modify-Write instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Thread 0	Thread 1
LOCK;INC x	LOCK;INC x
Forbidden Final State: [x]=1	

Also LOCK'd ADD, SUB, XCHG, etc., and CMPXCHG

Being able to do that atomically is important for many low-level algorithms. On x86 can also do for other sizes, including for 8B and 16B adjacent-doublesize quantities

CAS

Compare-and-swap (CAS):

CMPXCHG dest←src

compares EAX with dest, then:

- if equal, set ZF=1 and load src into dest,
- otherwise, clear ZF=0 and load dest into EAX

All this is one *atomic* step.

Can use to solve *consensus* problem...

Adding LOCK'd instructions to the model

1. extend the tiny language syntax
2. extend the tiny language semantics so that whatever represents a LOCK;INC x will (in thread t) do
   - 2.1 t:L
   - 2.2 t:R $x{=}v$ for an arbitrary v
   - 2.3 t:W $x{=}(v+1)$
   - 2.4 t:U
3. extend the x86-TSO abstract machine with rules for the LOCK and UNLOCK transitions

(this lets us reuse the semantics for INC for LOCK;INC, and to do so uniformly for all RMWs)

x86-TSO Abstract Machine: Behaviour

L: Lock

$$\frac{\begin{array}{c} m.L = \text{None} \\ m.B(t) = [\,] \end{array}}{m \xrightarrow{t:\mathsf{L}} m \oplus \langle\!\langle L := \text{Some}(t) \rangle\!\rangle}$$

If the lock is not held and its buffer is empty, thread t can begin a LOCK'd instruction.

Note that if a hardware thread t comes to a LOCK'd instruction when its store buffer is not empty, the machine can take one or more $t{:}\tau_{\,x=v}$ steps to empty the buffer and then proceed.

x86-TSO Abstract Machine: Behaviour

U: Unlock

$$\frac{\begin{array}{l} m.L = \mathrm{Some}(t) \\ m.B(t) = [\,] \end{array}}{m \xrightarrow{t:\mathrm{U}} m \oplus \langle\!\langle L := \mathrm{None} \rangle\!\rangle}$$

If t holds the lock, and its store buffer is empty, it can end a LOCK'd instruction.

Restoring SC with RMWs

CAS cost

From Paul McKenney (`http://www2.rdrop.com/~paulmck/RCU/`):

Want to be here!

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**	95.9	266.4

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot, plus suffer from contention

NB: Processors, Hardware Threads, and Threads

Our ‘Threads’ are hardware threads.

Some processors have *simultaneous multithreading* (Intel: hyperthreading): multiple hardware threads/core sharing resources.

If the OS flushes store buffers on context switch, software threads should have the same semantics.

NB: Not *All* of x86

Coherent write-back memory (almost all code), but assume

- no exceptions
- no misaligned or mixed-size accesses
- no 'non-temporal' operations
- no device memory
- no self-modifying code
- no page-table changes

Also no fairness properties: finite executions only, in this course.

x86-TSO vs SPARC TSO

x86-TSO based on SPARC TSO

SPARC defined

- TSO (Total Store Order)
- PSO (Partial Store Order)
- RMO (Relaxed Memory Order)

But as far as we know, only TSO has really been used (implementations have not been as weak as PSO/RMO or software has turned them off).

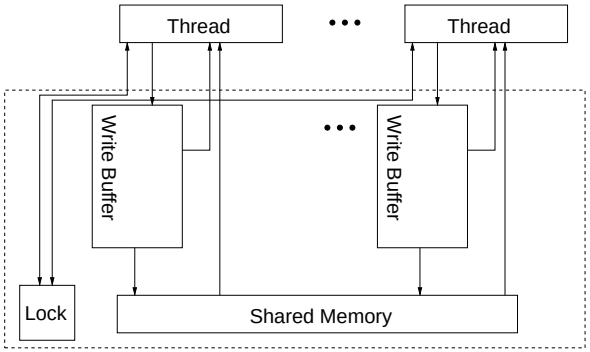
The SPARC Architecture Manual, Version 8, 1992. `http://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz` App. K defines TSO and PSO.

Version 9, Revision SAV09R1459912. 1994 `http://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz` Ch. 8 and App. D define TSO, PSO, RMO

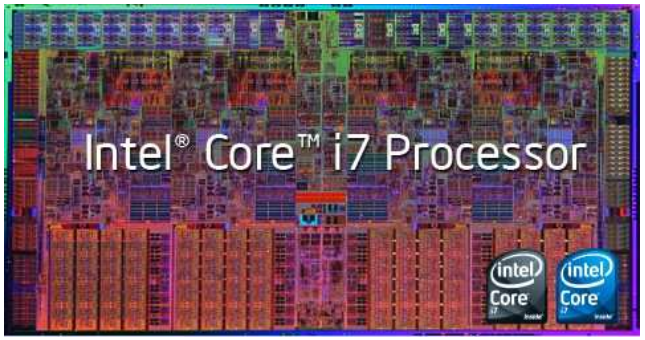
(in an axiomatic style – see later)

NB: This is an *Abstract* Machine

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



$\supseteq_{\text{beh}}$
$\neq_{\text{hw}}$



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

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

x86 spinlock example

Adding primitive mutexes to our source language

Statements $s ::= \ldots \mid \texttt{lock}\, x \mid \texttt{unlock}\, x$

Say lock free if it holds 0, taken otherwise.

Don't mix locations used as locks and other locations.

Semantics (outline): $\texttt{lock}\, x$ has to *atomically* (a) check the mutex is currently free, (b) change its state to taken, and (c) let the thread proceed.
$\texttt{unlock}\, x$ has to change its state to free.

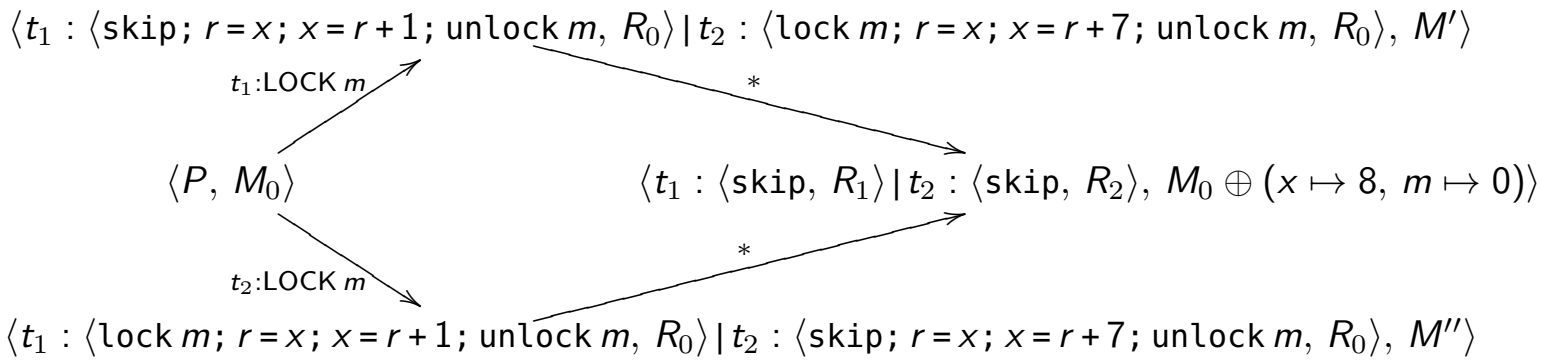
Record of which thread is holding a locked lock? Re-entrancy?

Using a Mutex

Consider

$$\begin{array}{rcl} P & = & t_1 : \langle \texttt{lock}\, m;\ r = x;\ x = r + 1;\ \texttt{unlock}\, m,\ R_0 \rangle \\ & \mid & t_2 : \langle \texttt{lock}\, m;\ r = x;\ x = r + 7;\ \texttt{unlock}\, m,\ R_0 \rangle \end{array}$$

in the initial store M_0:



where $M' = M_0 \oplus (m \mapsto 1)$

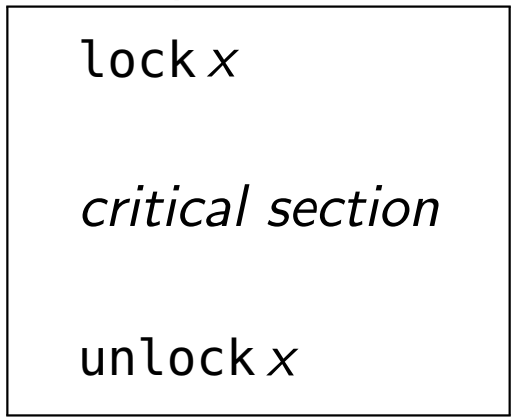
Deadlock

`lock` m can block (that's the point). Hence, you can *deadlock*.

$$
\begin{aligned}
P = \quad & t_1 : \langle \texttt{lock}\, m_1 \texttt{;}\ \texttt{lock}\, m_2 \texttt{;}\ x = 1 \texttt{;}\ \texttt{unlock}\, m_1 \texttt{;}\ \texttt{unlock}\, m_2,\ R_0 \rangle \\
\mid \quad & t_2 : \langle \texttt{lock}\, m_2 \texttt{;}\ \texttt{lock}\, m_1 \texttt{;}\ x = 2 \texttt{;}\ \texttt{unlock}\, m_1 \texttt{;}\ \texttt{unlock}\, m_2,\ R_0 \rangle
\end{aligned}
$$

Implementing mutexes with simple x86 spinlocks

Implementing the language-level mutex with x86-level simple spinlocks



Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    skip
}

critical section

unlock(x)
```

Invariant:

lock taken if x $\leq$ 0

lock free if x=1

(NB: different internal representation from high-level semantics)

Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}

critical section

unlock(x)
```

Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}

critical section

x ←1        OR        atomic_write(x, 1)
```

Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}

critical section

x ←1
```

Simple x86 Spinlock

The address of x is stored in register eax.

```
acquire:    LOCK DEC [eax]
            JNS enter
spin:       CMP [eax],0
            JLE spin
            JMP acquire
enter:

        critical section

release:    MOV [eax]←1
```

From Linux v2.6.24.7

NB: don't confuse levels — we're using x86 atomic (LOCK'd) instructions in a Linux spinlock implementation.

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
$x = 1$		

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
$x = 1$		
$x = 0$	acquire	

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
$x = 1$		
$x = 0$	acquire	
$x = 0$	critical	

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	
x = 1		read x

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	
x = 1		read x
x = 0		acquire

Spinlock SC Data Race

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0		
x = 0		
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

Spinlock SC Data Race

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0		
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

Spinlock SC Data Race

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x

Spinlock Example (x86-TSO)

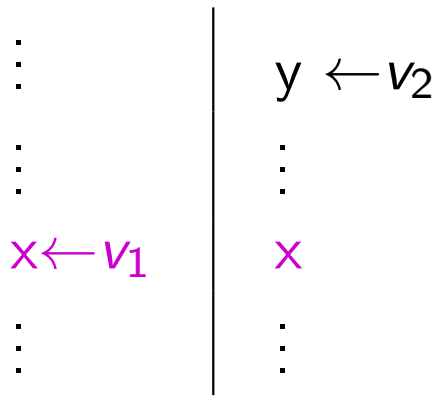
```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x
x = 0		acquire

Triangular Races (Owens)

- Read/write data race
- Only if there is a bufferable write preceding the read

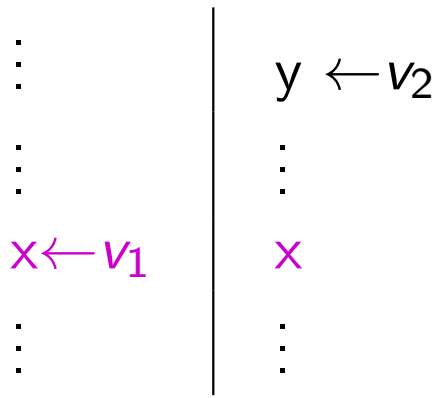
Triangular race



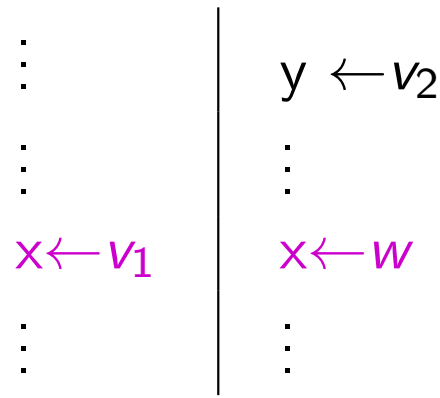
Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



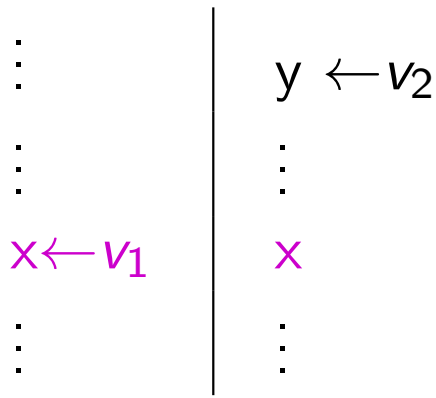
Not triangular race



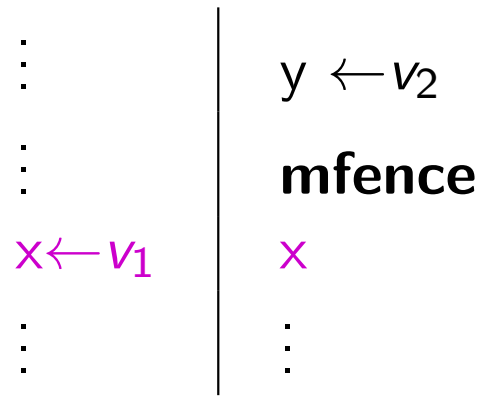
Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



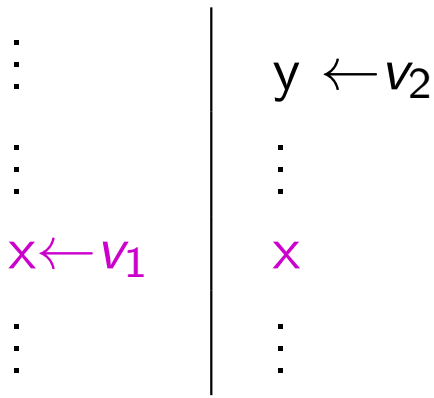
Not triangular race



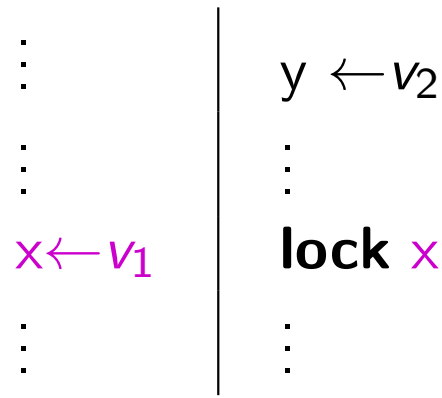
Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



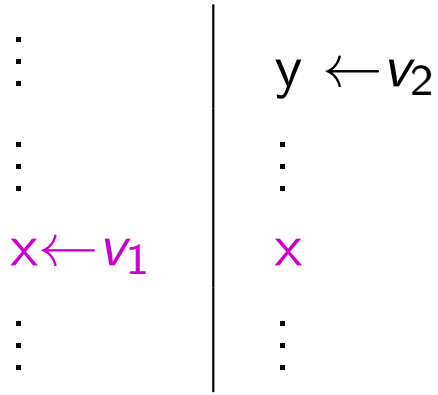
Not triangular race



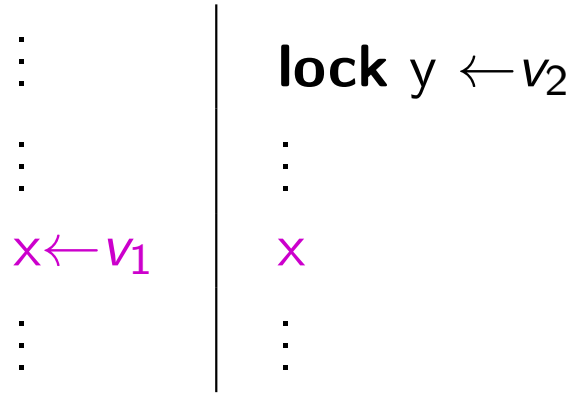
Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



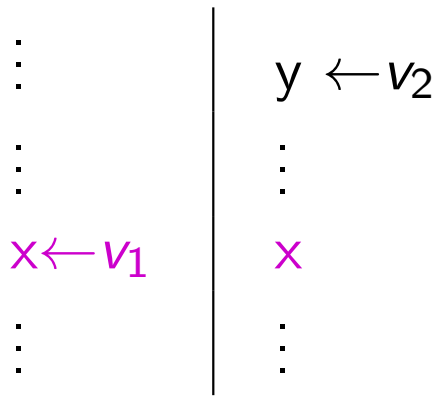
Not triangular race



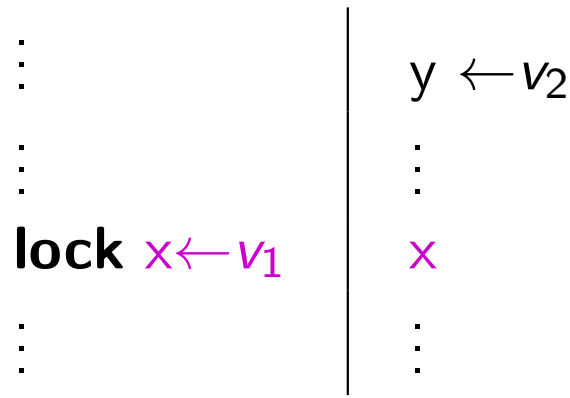
Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



Triangular race



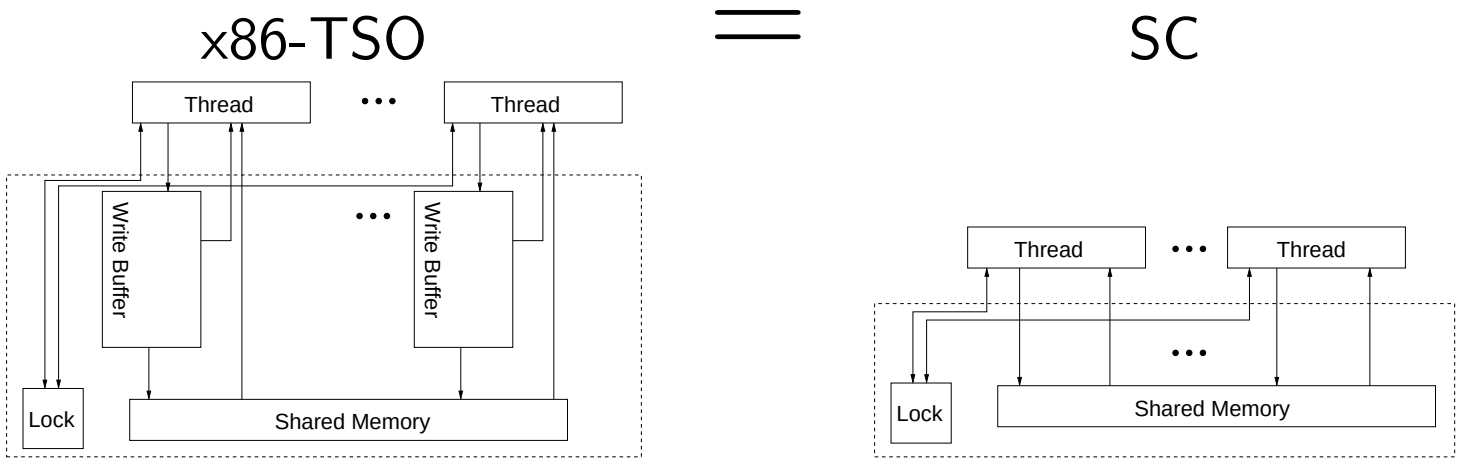
TRF Principle for x86-TSO

Say a program is *triangular race free (TRF)* if no SC execution has a triangular race.

Theorem (TRF)

If a program is TRF then any x86-TSO execution is equivalent to some SC execution.

If a program has no triangular races when run on a sequentially consistent memory, then



Spinlock Data Race

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

- acquire's writes are locked

Program Correctness

Theorem

Any well-synchronized program that uses the spinlock correctly is TRF.

Theorem

Spinlock-enforced critical sections provide mutual exclusion.

Other Applications of TRF

A concurrency bug in the HotSpot JVM

- Found by Dave Dice (Sun) in Nov. 2009
- `java.util.concurrent.LockSupport` ('Parker')
- Platform specific C++
- Rare hung thread
- Since "day-one" (missing MFENCE)
- Simple explanation in terms of TRF

Also: Ticketed spinlock, Linux SeqLocks, Double-checked locking

Architectures

What About the Specs?

Hardware manufacturers document *architectures*:

Intel 64 and IA-32 Architectures Software Developer's Manual
AMD64 Architecture Programmer's Manual
Power ISA specification

ARM Architecture Reference Manual

and programming languages (at best) are defined by *standards*:

ISO/IEC 9899:1999 Programming languages – C

J2SE 5.0 (September 30, 2004)

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

What About the Specs?

Hardware manufacturers document *architectures*:

Intel 64 and IA-32 Architectures Software Developer's Manual
AMD64 Architecture Programmer's Manual
Power ISA specification

ARM Architecture Reference Manual

and programming languages (at best) are defined by *standards*:

ISO/IEC 9899:1999 Programming languages – C

J2SE 5.0 (September 30, 2004)

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

Flawed. Always confusing, sometimes wrong.

What About the Specs?

"all that horrible horribly incomprehensible and confusing [...] text that no-one can parse or reason with — not even the people who wrote it"

Anonymous Processor Architect, 2011

Why all these problems?

Recall that the vendor *architectures* are:

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

Architectures should:

- reveal enough for effective programming;
- without revealing sensitive IP; and
- without unduly constraining future processor design.

There's a big tension between these, compounded by internal politics and inertia.

Fundamental Problem

Architecture texts: *informal prose* attempts at subtle loose specifications

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

(Intel SDM, Nov. 2006, vol 3a, 10-5)

Fundamental Problem

Architecture texts: *informal prose* attempts at subtle loose specifications

Fundamental problem: prose specifications cannot be used

- to *test programs against*, or
- to *test processor implementations*, or
- to *prove* properties of either, or even
- to *communicate precisely*.

(in a real sense, the architectures don't *exist*).

The models we're developing here can be used for all these things. An 'architecture' should be such a precisely defined mathematical artifact.

Validating the models?

We are inventing new abstractions, not just formalising existing clear-but-non-mathematical specs. So why should anyone believe them?

- some aspects of existing arch specs *are* clear (a few concurrency examples, much of ISA spec)
- experimental testing
  - models should be *sound* w.r.t. experimentally observable behaviour of existing h/w (modulo h/w bugs)
  - but the architectural intent may be (often is) looser
- discussion with architects
- consistency with expert-programmer intuition
- formalisation (at least mathematically consistent)
- proofs of metatheory

Tests and Testing

‘Empirical Science of the Artificial’

Treating these human-made artifacts as objects of empirical science

In principle (modulo manufacturing defects): their structure and behaviour are completely known.

In practice: the structure is too complex for anyone to fully understand, the emergent behaviour is not well-understood, and there are commercial confidentiality issues.

Litmus Testing

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$;	$y = 1$;
$r_0 = y$	$r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

Litmus Testing

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$;	$y = 1$;
$r_0 = y$	$r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

Step 1: Get the compiler out of the way, writing tests in assembly:

`SB.litmus`:

```
X86 SB ""
{x = 0; y = 0};

        P0       |        P1       ;
  mov [x], 1     |  mov [y], 1     ;
  mov EAX, [y]   |  mov EBX, [x]   ;

exists (P0:EAX = 0 /\ P1:EBX = 0);
```

Litmus Testing

Step 2: Want to run that test

- starting in a wide range of the processor's internal states (cache-line states, store-buffer states, pipeline states, ...),
- with the threads roughly synchronised, and
- with a wide range of timing and interfering activity.

Our `litmus` tool takes a test and compiles it to a program (C with embedded assembly) that does that.

Basic idea: have an array for each location (x, y) and the observed results; run many instances of test in a randomised order.

First version: Braibant, Sarkar, Zappa Nardelli [x86-CC, POPL09].
Now mostly Maranget: [TACAS11]

Litmus Testing

Install via opam, or download `litmus`:
`http://diy.inria.fr/sources/litmus.tar.gz`

Untar, edit the `Makefile` to set the install `PREFIX` (e.g. to the untar'd directory).

`make all` (needs OCaml) and `make install`

```
./litmus -mach corei7.cfg testsuite/X86/SB.litmus
```

Docs at `http://diy.inria.fr/doc/litmus.html`

More tests on course web page.

Litmus Output (1/2)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for ../../../sem/WeakMemory/litmus.new/x86/SB.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
X86 SB
"Loads may be reordered with older stores to different locations"

{x=0; y=0;}

 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EBX,[x] ;

exists (0:EAX=0 /\ 1:EBX=0)
Generated assembler
#START _litmus_P1
movl $1,(%rdi,%rcx)
movl (%rdx,%rcx),%eax
#START _litmus_P0
movl $1,(%rsi,%rdx)
movl (%rdi,%rdx),%eax
```

Litmus Output (2/2)

```
Test SB Allowed
Histogram (4 states)
11    *>0:EAX=0; 1:EBX=0;
499985:>0:EAX=1; 1:EBX=0;
499991:>0:EAX=0; 1:EBX=1;
13    :>0:EAX=1; 1:EBX=1;
Ok

Witnesses
Positive: 11, Negative: 999989
Condition exists (0:EAX=0 /\ 1:EBX=0) is validated
Hash=d907d5adfff1644c962c0d8ecb45bbff
Observation SB Sometimes 11 999989
Time SB 0.17
```

...and logging `/proc/cpuinfo`, litmus options, and gcc options

Good practice: the litmus file condition identifies a particular outcome of interest (often enough to completely determine the reads-from and coherence relations of an execution), but does *not* say whether that outcome is allowed or forbidden in any particular model; that's kept elsewhere.

What's a Test?

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$;	$y = 1$;
$r_0 = y$	$r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

What's a Test?

Initial state: x=0 and y=0	
Thread 0	Thread 1
$x = 1$; $r_0 = y$	$y = 1$; $r_1 = x$
Allowed? Thread 0's $r_0 = 0 \wedge$	Thread 1's $r_1 = 0$

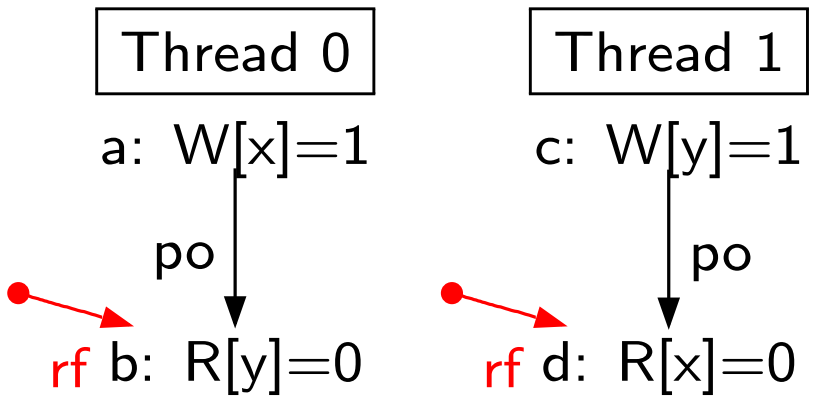
In the operational model, is there a trace

$$\langle t_0 : \langle x = 1;\ r_0 = y,\ R_0\rangle \mid t_1 : \langle y = 1;\ r_1 = x,\ R_0\rangle,\ \{x \mapsto 0,\ y \mapsto 0\}\rangle$$
$$\xrightarrow{l_1} \ldots \xrightarrow{ln}$$
$$\langle t_0 : \langle \texttt{skip},\ R_0'\rangle \mid t_1 : \langle \texttt{skip},\ R_1'\rangle,\ M'\rangle$$

such that $R_0'(r_0) = 0$ and $R_1'(r_1) = 0$?

Candidate Execution Diagrams

That final condition identifies a set of executions, with particular read and write events; we can abstract from the threadwise semantics and just draw those:



Test SB

- in these diagrams, the events are organised by threads, we elide the thread ids, but we give each event a unique id a, b, ...
- we draw *program order* (po) edges within each thread;
- we draw *reads-from* (rf) edges from each write (or a red dot for the initial state) to all reads that read from it;

Coherence

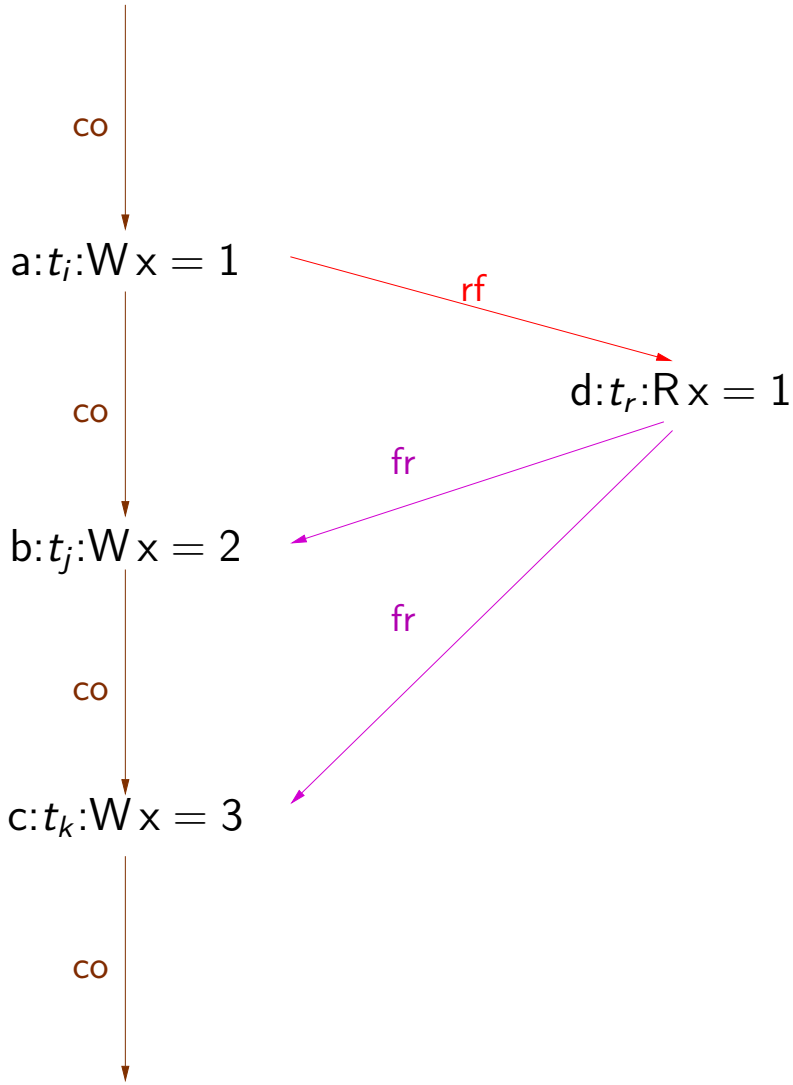
Conventional hardware architectures guarantee *coherence*:

- in any execution, for each location, there is a total order over all the writes to that location, and for each thread the order is consistent with the thread's program-order for its reads and writes to that location; or (loosely)
- in any execution, for each location, the execution restricted to just the reads and writes to that location is SC.

In simple hardware implementations, that's the order in which the processors gain write access to the cache line.

From-reads

Given that, we can think of a read event as "before" the coherence-successors of the write it reads from.



From-reads

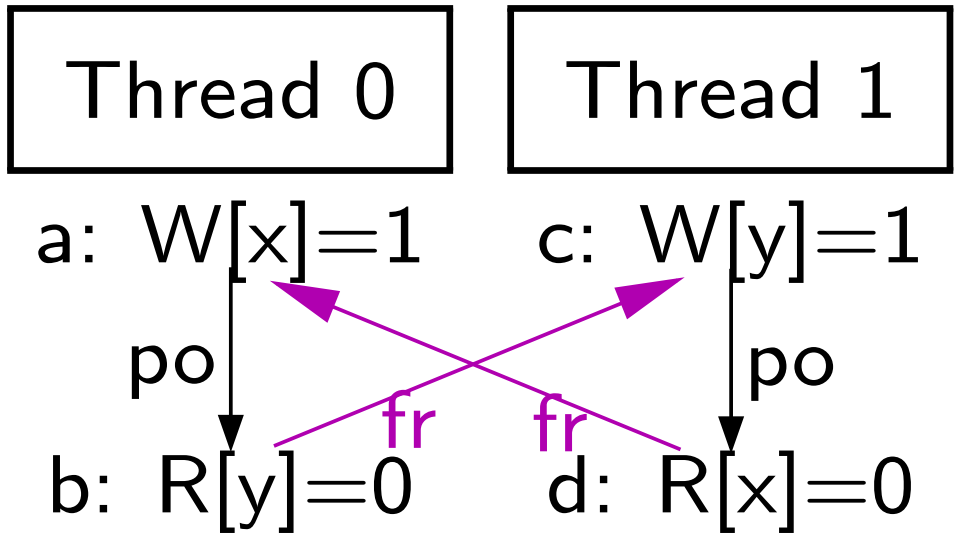
Given that, we can think of a read event as "before" the coherence-successors of the write it reads from.

Given a candidate execution with a coherence order co over the writes to x, and a reads-from relation rf from writes to x to the reads that read from them, define the *from-reads* relation fr to relate each read to the co-successors of the write it reads from (or to all writes to x if it reads from the initial state).

$$r \xrightarrow{\text{fr}} w \quad \text{iff} \quad (\exists w_0.\ w_0 \xrightarrow{\text{co}} w \quad \wedge \quad w_0 \xrightarrow{\text{rf}} r) \quad \vee \quad (\neg\exists w_0.\ w_0 \xrightarrow{\text{rf}} r)$$

(co is an irreflexive transitive relation)

The SB cycle



Test SB

A more abstract characterisation of why this execution is non-SC?

Candidate Executions, more precisely

Forget the memory states M_i and focus just on the read and write events. Give them ids $a, b, \ldots$ (unique within an execution): $a : t : \mathrm{R}\, x{=}n$ and $a : t : \mathrm{W}\, x{=}n$.
Say a *candidate pre-execution* E consists of

- a finite set E of such events
- *program order* (po), an irreflexive transitive relation over E

  [intuitively, from a control-flow unfolding and choice of arbitrary memory read values of the source program]

Say a *candidate execution witness for* E, X, consists of with

- *reads-from* (rf), a relation over E relating writes to the reads that read from them (with same address and value)

  [note this is intensional: it identifies *which write*, not just the value]
- *coherence* (co), an irreflexve transitive relation over E relating only writes that are to the same address; total when restricted to the writes of each address separately

  [intuitively, the hardware coherence order for each address]

SC, said differently again: pre-executions

Say a candidate pre-execution E is SC-L if there exists a total order sc over all its events such that for all read events $e_r = (a : t : \mathrm{R}\, x{=}n) \in E$, either n is the value of the most recent (w.r.t. sc) write to x, if there is one, or 0, otherwise.

Theorem (?)

E is SC-L iff there exists a trace $\vec{l} \in \mathrm{traces}(M_0)$ of M_0 such that the events of E are the labels of $\vec{l}$ (with a choice of unique id for each) and po is the union of the order of $\vec{l}$ restricted to each thread.

Say a candidate pre-execution E is consistent with the threadwise semantics of process P if there exists a trace $\vec{l} \in \mathrm{traces}(P)$ of P such that the events of E are the labels of $\vec{l}$ (with a choice of unique id for each) and po is the union of the order of $\vec{l}$ restricted to each thread.

SC, said differently again: "Axiomatically"

Say a candidate pre-execution E and execution witness X are SC-A if

$$\mathrm{acyclic}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})$$

Theorem (?)

E is SC-L iff there exists an execution witness X (satisfying the well-formedness conditions of the last-but-one slide) such that E, X is SC-A.

This characterisation of SC is existentially quantifying over irrelevant order...

How to generate good tests?

- hand-crafted test programs [RAPA, Collier]
- hand-crafted litmus tests
- exhaustive or random small program generation
- from executions that (minimally?) violate
  $\mathrm{acyclic}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})$

  ...given such an execution, construct a litmus test program and final condition that picks out that execution

  [`diy` tool of Alglave and Maranget `http://diy.inria.fr/doc/gen.html`; and Shasha and Snir, TOPLAS 1988]
- systematic families of those (see periodic table, later)

Accumulated library of 1000's of litmus tests.

How to compare test results and models?

Need model to be *executable as a test oracle*: given a litmus test, want to compute the set of *all* results the model permits.

Then compare that set with the set of all results observed running test (with `litmus` harness) on actual hardware.

model	experiment	conclusion
Y	Y	
Y	–	model is looser (or testing not aggressive)
–	Y	model not sound (or hardware bug)
–	–	

The SC semantics as executable test oracles

Given P, either:

1. enumerate entire graph of $\langle P, M_0 \rangle$ transition system

   (maybe with some partial-order reduction), or

2. 2.1 enumerate all pre-executions E, by enumerating entire graph of P threadwise semantics transition system;
   2.2 for each E, enumerate all pairs of relations over the events (for rf and co, to make a well-formed execution witness X); and
   2.3 discard those that don't satisfy the SC-A acyclicity predicate of E, X.

(actually for (1), use an inductive-on-syntax characterisation of the set of all pre-executions of a process)

These are *operational* and *axiomatic* styles of defining relaxed memory models.

References

- Reasoning About Parallel Architectures (RAPA), William W. Collier, Prentice-Hall, 1992. `http://www.mpdiag.com`
- The Semantics of x86-CC Multiprocessor Machine Code. Sarkar, Sewell, Zappa Nardelli, Owens, Ridge, Braibant, Myreen, Alglave. POPL 2009
- A Better x86 Memory Model: x86-TSO. Owens, Sarkar, Sewell. TPHOLs 2009.
- Fences in Weak Memory Models. Alglave, Maranget, Sarkar, Sewell. CAV 2010.
- Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. Scott Owens. ECOOP 2010.
- x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors, Sewell, Sarkar, Owens, Zappa Nardelli, Myreen. Communications of the ACM (Research Highlights) 2010 No.7.
- Litmus: Running Tests Against Hardware. Alglave, Maranget, Sarkar, Sewell. TACAS 2011 (Tool Demonstration Paper).