

Multicore Programming

Peter Sewell

Jaroslav Ševčík

Tim Harris

University of Cambridge

MSR

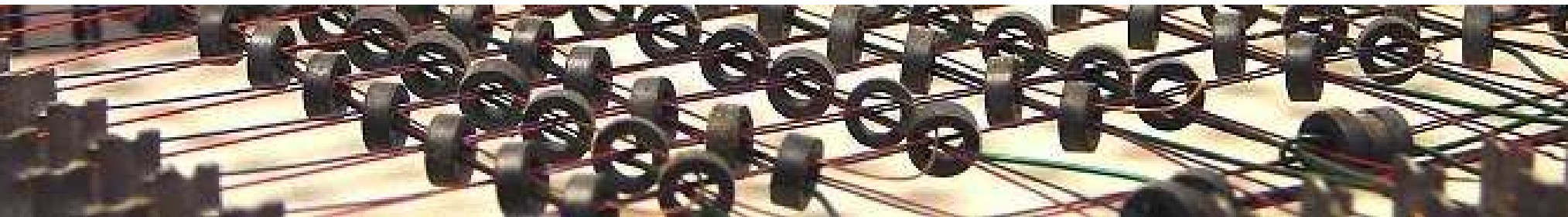
with thanks to

Francesco Zappa Nardelli, Susmit Sarkar, Tom Ridge, Scott Owens,
Magnus O. Myreen, Luc Maranget, Mark Batty, Jade Alglave

October – November, 2010

The Golden Age, 1945–1972

Memory = An array of values

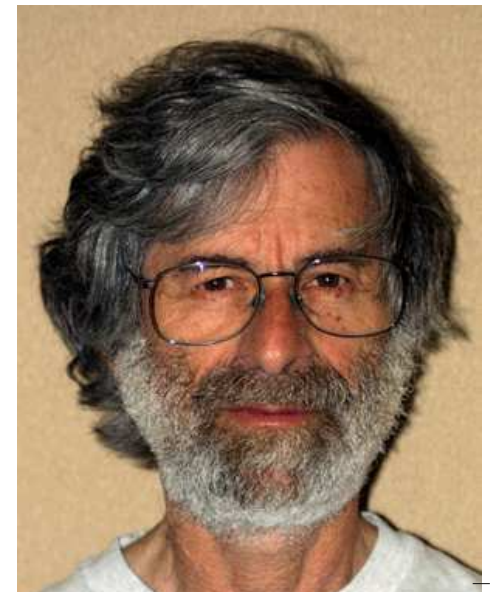
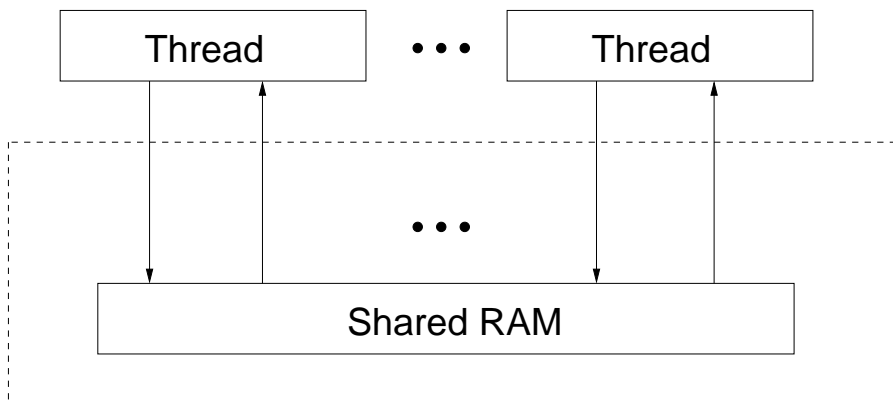


In an Ideal World

Multiprocessors would have *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, and the operations of each individual processor appear in this sequence in the order specified by its program”.

Leslie Lamport, 1979



A Tiny Language

location, x address (or pointer value)

integer, n integer

thread_id, t thread id

<i>expression, e</i>	::=	expression
		<i>n</i> integer literal
		<i>*x</i> read from pointer
		<i>*x = e</i> write to pointer
		<i>e; e'</i> sequential composition

<i>process, p</i>	::=	process
		<i>t:e</i> thread
		<i>p p'</i> parallel composition

... and an SC Semantics

Take a *memory* M to be a function from addresses to integers.

$state, s \quad ::= \quad state$
 $\quad \quad \quad | \quad \langle p, M \rangle \quad \quad \quad process\ p\ paired\ with\ memory\ M$

$label, l \quad ::= \quad label$
 $\quad \quad \quad | \quad W\ x=n \quad \quad \quad write$
 $\quad \quad \quad | \quad R\ x=n \quad \quad \quad read$
 $\quad \quad \quad | \quad \tau \quad \quad \quad internal\ action\ (tau)$

and *thread labels* $l_t \quad ::= \quad W_t\ x=n \quad | \quad R_t\ x=n \quad | \quad \tau_t$

... and an SC Semantics: expressions

$\boxed{e \xrightarrow{l} e'}$ e does l to become e'

$\frac{}{*x \xrightarrow{R\ x=n} n}$ READ

$\frac{}{*x = n \xrightarrow{W\ x=n} n}$ WRITE

$\frac{}{n; e \xrightarrow{\tau} e}$ SEQ

$\frac{e_1 \xrightarrow{l} e'_1}{e_1; e_2 \xrightarrow{l} e'_1; e_2}$ SEQ_CONTEXT

Example: SC Expression Trace

$(*x = *y); *x$

Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R y=7} \xrightarrow{W x=7} \xrightarrow{\tau} \xrightarrow{R x=9} 9$

Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R y=7} \xrightarrow{W x=7} \xrightarrow{\tau} \xrightarrow{R x=9} 9$

$\frac{\frac{\text{---}}{*y} \xrightarrow{R y=7} 7}{*x = *y} \xrightarrow{R y=7} *x = 7$	<p>READ</p> <p>WRITE</p>	
$(*x = *y); *x \xrightarrow{R y=7} (*x = 7); *x$	<p>SEQ_CONTEXT</p>	

Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R y=7} \xrightarrow{W x=7} \xrightarrow{\tau} \xrightarrow{R x=9} 9$

$\frac{\quad}{*x = 7 \xrightarrow{W x=7} 7}$	WRITE	
$(*x = 7); *x \xrightarrow{W x=7} 7; *x$	SEQ_CONTEXT	

Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R y=7} \xrightarrow{W x=7} \xrightarrow{\tau} \xrightarrow{R x=9} 9$

$\frac{}{7; *x \xrightarrow{\tau} *x}$ SEQ

$\frac{}{*x \xrightarrow{R x=9} 9}$ READ

... and an SC Semantics: lifting to processes

$\boxed{p \xrightarrow{l_t} p'}$ p does l_t to become p'

$$\frac{e \xrightarrow{l} e'}{t:e \xrightarrow{l_t} t:e'} \quad \text{THREAD}$$
$$\frac{p_1 \xrightarrow{l_t} p'_1}{p_1|p_2 \xrightarrow{l_t} p'_1|p_2} \quad \text{PAR_CONTEXT_LEFT}$$
$$\frac{p_2 \xrightarrow{l_t} p'_2}{p_1|p_2 \xrightarrow{l_t} p_1|p'_2} \quad \text{PAR_CONTEXT_RIGHT}$$

... and an SC Semantics: SC memory

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

$$\frac{M(x) = n}{M \xrightarrow{R\ x=n} M} \quad \text{MREAD}$$

$$\frac{}{M \xrightarrow{W\ x=n} M \oplus (x \mapsto n)} \quad \text{MWRITE}$$

... and an SC Semantics: whole-system states

$s \xrightarrow{l_t} s'$ s does l_t to become s'

$$\frac{\begin{array}{l} p \xrightarrow{R_t x=n} p' \\ M \xrightarrow{R x=n} M' \end{array}}{\langle p, M \rangle \xrightarrow{R_t x=n} \langle p', M' \rangle} \quad \text{SREAD}$$

$$\frac{\begin{array}{l} p \xrightarrow{W_t x=n} p' \\ M \xrightarrow{W x=n} M' \end{array}}{\langle p, M \rangle \xrightarrow{W_t x=n} \langle p', M' \rangle} \quad \text{SWRITE}$$

$$\frac{p \xrightarrow{\tau_t} p'}{\langle p, M \rangle \xrightarrow{\tau_t} \langle p', M \rangle} \quad \text{STAU}$$

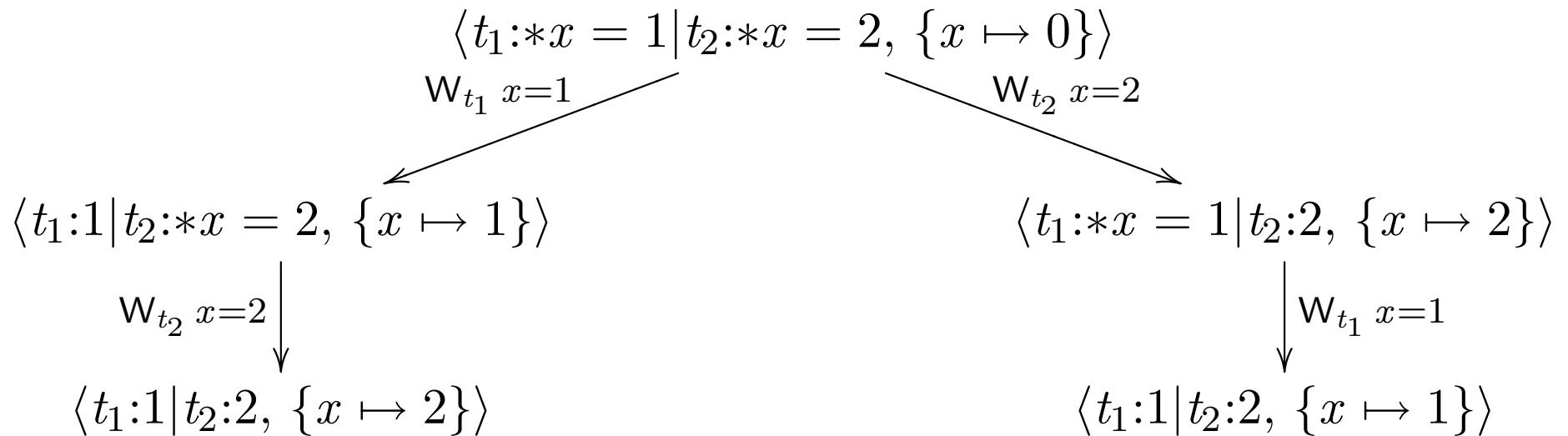
Example: SC Whole-System Trace

$$\langle t_1: (*x = *y); *x, \{x \mapsto 0, y \mapsto 7\} \rangle$$
$$\begin{array}{c} R_{t_1} y=7 \\ \downarrow \end{array}$$
$$\langle t_1: (*x = 7); *x, \{x \mapsto 0, y \mapsto 7\} \rangle$$
$$\begin{array}{c} W_{t_1} x=7 \\ \downarrow \end{array}$$
$$\langle t_1: 7; *x, \{x \mapsto 7, y \mapsto 7\} \rangle$$
$$\begin{array}{c} \tau_{t_1} \\ \downarrow \end{array}$$
$$\langle t_1: *x, \{x \mapsto 7, y \mapsto 7\} \rangle$$
$$\begin{array}{c} R_{t_1} x=7 \\ \downarrow \end{array}$$
$$\langle t_1: 7, \{x \mapsto 7, y \mapsto 7\} \rangle$$

Example: 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.

In an Ideal World

Multiprocessors would have *sequentially consistent* shared memory

Taken for granted, by almost all

- concurrency theorists
- program logics
- concurrent verification tools
- programmers

False, since 1972

IBM System 370/158MP



Mass-market since 2005.



The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$)	MOV $[y] \leftarrow 1$ (write $y=1$)
MOV EAX $\leftarrow [y]$ (read y)	MOV EBX $\leftarrow [x]$ (read x)

What final states are allowed?

What are the possible sequential orders?

The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

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

Thread 0:EAX = 0

Thread 1:EBX=1

The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$)	MOV $[y] \leftarrow 1$ (write $y=1$)
MOV EAX $\leftarrow [y]$ (read $y=1$)	MOV EBX $\leftarrow [x]$ (read $x=1$)

Thread 0: EAX = 1

Thread 1: EBX = 1

The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$)	MOV $[y] \leftarrow 1$ (write $y=1$)
MOV EAX $\leftarrow [y]$ (read $y=1$)	MOV EBX $\leftarrow [x]$ (read $x=1$)

Thread 0: EAX = 1

Thread 1: EBX = 1

The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$)	MOV $[y] \leftarrow 1$ (write $y=1$)
MOV EAX $\leftarrow [y]$ (read $y=1$)	MOV EBX $\leftarrow [x]$ (read $x=1$)

Thread 0:EAX = 1

Thread 1:EBX=1

The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$)	MOV $[y] \leftarrow 1$ (write $y=1$)
MOV EAX $\leftarrow [y]$ (read $y=1$)	MOV EBX $\leftarrow [x]$ (read $x=1$)

Thread 0: EAX = 1

Thread 1: EBX = 1

The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y , initially 0

Thread 0	Thread 1
	MOV $[y] \leftarrow 1$ (write $y=1$)
	MOV EBX $\leftarrow [x]$ (read $x=0$)
MOV $[x] \leftarrow 1$ (write $x=1$)	
MOV EAX $\leftarrow [y]$ (read $y=1$)	

Thread 0:EAX = 1

Thread 1:EBX=0

The First Bizarre Example

Conclusion:

0,1 and 1,1 and 1,0 can happen, but 0,0 is impossible

The First Bizarre Example

Conclusion:

0,1 and 1,1 and 1,0 can happen, but 0,0 is impossible

In fact, in the real world:

we observe 0,0 every 630/100000 runs
(on an Intel Core Duo x86)

(and so Dekker's algorithm will fail)



Weakly Consistent 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.

Problems

- Memory \neq Array of values + Global time

Problems

- Memory \neq Array of values + Global time
- Real processor and language models are subtle & various
- Memory model research mostly idealised
- Concurrency verification research mostly SC
- Industrial processor and language specs?

We've looked at the specs of x86, Power, ARM, Java, and C++. *They're all flawed*

~~Problems~~ Research Opportunities

- Memory \neq Array of values + Global time
- Real processor and language models are subtle & various
- Memory model research mostly idealised
- Concurrency verification research mostly SC
- Industrial processor and language specs?

We've looked at the specs of x86, Power, ARM, Java, and C++. They're *all* flawed

These Lectures

Part 1: Relaxed-memory concurrency, for multiprocessors and programming languages (Peter Sewell, Jaroslav Ševčík, and Mark Batty)

Establish a solid basis for thinking about relaxed-memory executions, linking to usage, microarchitecture, experiment, and semantics.

Part 2: Concurrent algorithms (Tim Harris)

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

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

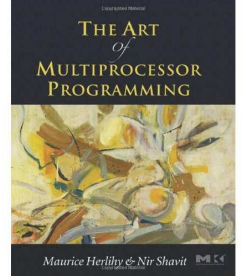
Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

Uses



1. how to code low-level concurrent datastructures
2. how to build concurrency testing and verification tools
3. how to specify and test multiprocessors
4. how to design and express high-level language definitions
5. ... and in general, as an example of mathematically rigorous computer engineering

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

Architectures

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

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



Intel® 64 and IA-32 Architect
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, Nov. 2006, vol 3a, 10-5)

As we shall see, such descriptions sometimes are:

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

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

A Cautionary Tale

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

A model called *Processor Ordering*, informal prose

Example: Linux Kernel mailing list, 20 Nov 1999 - 7 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_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 spinlocks down from about 22 ticks for the "lock; btr1 \$0,%0" a to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. he reported that Ingo Molnar noticed a 4% speed-up in mark test, making the optimization very valuable. I added that the same optimization cropped up in the mailing list a few days previously. But Linus Torvalds poured water on the whole thing, saying:

It does NOT WORK!

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

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more. - p. 26

Resolved only by appeal to an oracle:

that the pipelines are no longer invalid and the code should be blown out.

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

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

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

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

The reason for this is that stores can only be observed when all prior instructions have completed (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 stores, etc absolutely have to have completed before a cache-miss or not.

He went on:

Since the instructions for the store in the spin

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	

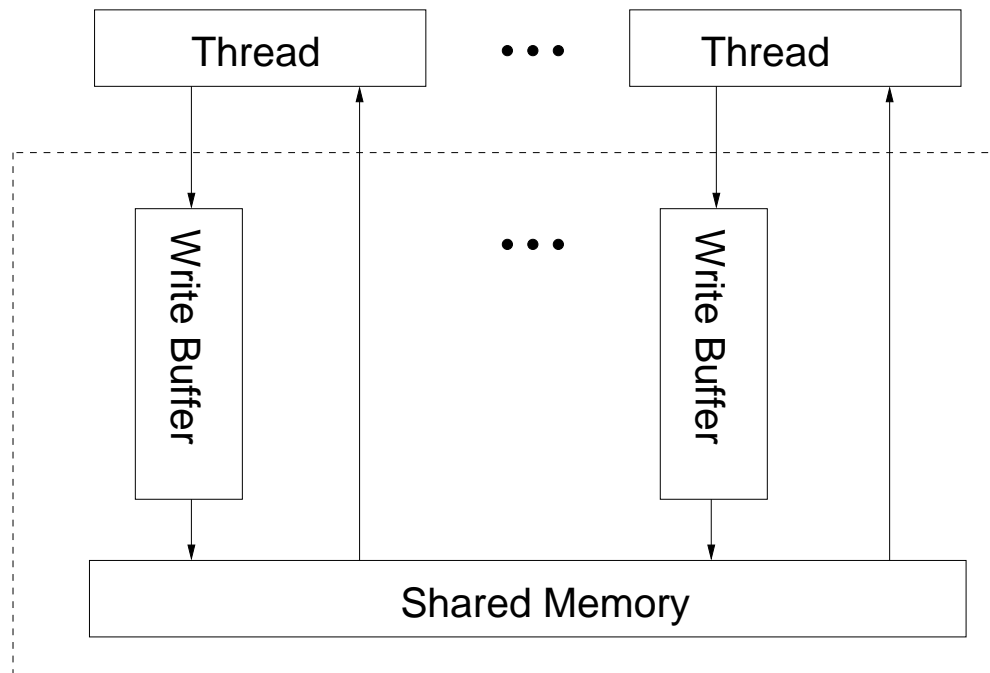
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

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 \wedge Thread 1:EBX=0	

P3. Loads may be reordered with older stores to different locations 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 \wedge Thread 1:EBX=0	

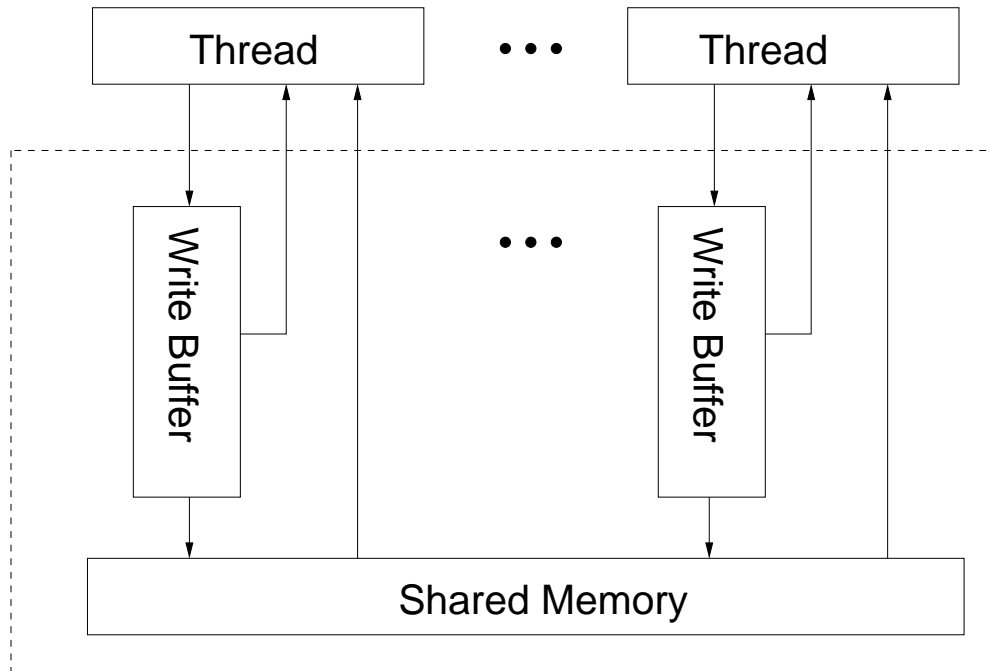


Litmus Test 2.4. Intra-processor forwarding is allowed

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 \wedge Thread 1:EDX=0 Thread 0:EAX=1 \wedge 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 \wedge Thread 1:EDX=0 Thread 0:EAX=1 \wedge 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=1)
		(read y=0)	(read x=0)
Allowed or Forbidden?			



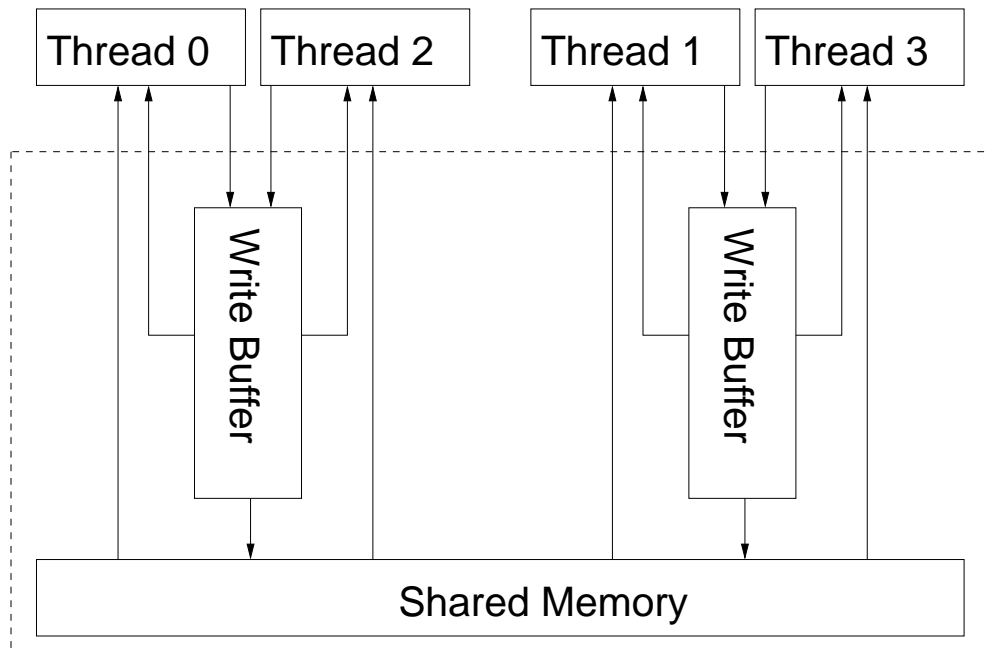
Problem 1: Weakness

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=1)
		(read y=0)	(read x=0)

Allowed or Forbidden?

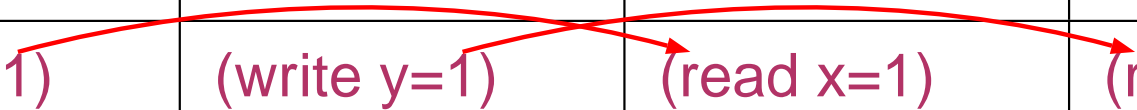
Microarchitecturally plausible? yes, e.g. with shared store buffers



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=1)
		(read y=0)	(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 \wedge Thread 2:EBX=1 \wedge 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’.

(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 \wedge Thread 0:EBX=0 \wedge 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 [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	

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 – now

Intel SDM rev. 29–35 and AMD3.17

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:

<http://www.intel.com/products/processor/manuals/index.htm>
(rev. 35 on 6/10/2010).

See especially SDM Vol. 3A, Ch. 8.

AMD:

<http://developer.amd.com/documentation/guides/Pages/default.aspx>

(rev. 3.17 on 6/10/2010).

See especially APM Vol. 2, Ch. 7.

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

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*.

Aside: x86 ISA, Locked Instructions

Thread 0	Thread 1
INC x	INC x

Aside: x86 ISA, Locked 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)

Aside: x86 ISA, Locked 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	

Aside: x86 ISA, Locked 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

Aside: x86 ISA, Locked Instructions

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...

Aside: x86 ISA, Memory Barriers

MFENCE memory barrier

(also SFENCE and LFENCE)

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

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

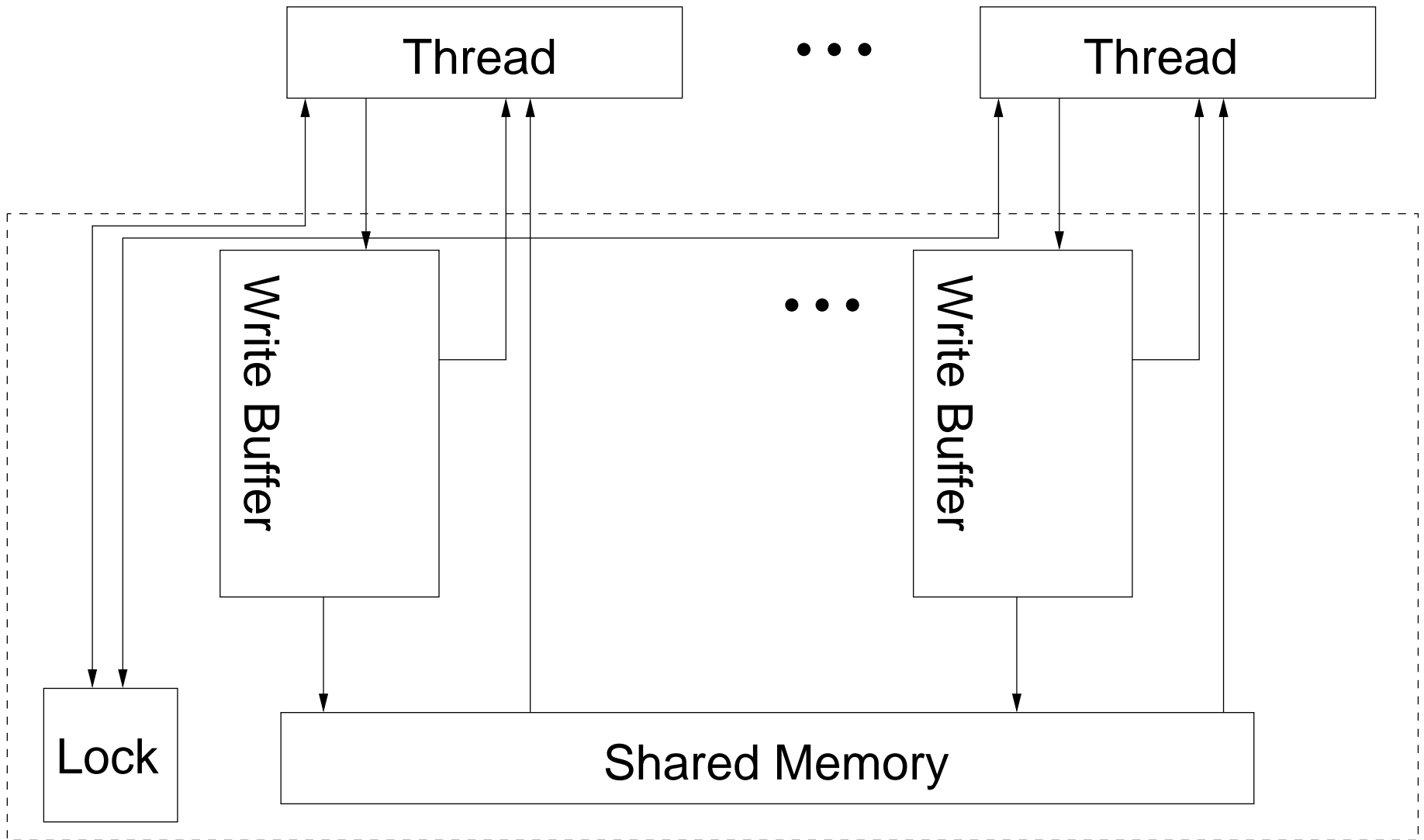
Separate *instruction semantics* and *memory model*

Define the memory model in two (provably equivalent) styles:

- an abstract machine (or operational model)
- an axiomatic model

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

x86-TSO Abstract Machine



x86-TSO Abstract Machine: Interface

Events

$e ::=$	$W_t x=v$	a write of value v to address x by thread t
	$R_t x=v$	a read of v from x by t
	B_t	an MFENCE memory barrier by t
	L_t	start of an instruction with LOCK prefix by t
	U_t	end of an instruction with LOCK prefix by t
	$\tau_t x=v$	an internal action of the machine, moving $x = v$ from the write buffer on t to shared memory

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

A *machine state* s is a record

$$s : \langle \begin{array}{l} M : \text{addr} \rightarrow \text{value}; \\ B : \text{tid} \rightarrow (\text{addr} \times \text{value}) \text{ list}; \\ L : \text{tid option} \end{array} \rangle$$

Here:

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

x86-TSO Abstract Machine: Auxiliary Definitions

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

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

x86-TSO Abstract Machine: Behaviour

RM: Read from memory

$\text{not_blocked}(s, t)$

$s.M(x) = v$

$\text{no_pending}(s.B(t), x)$

$$\frac{}{s \xrightarrow{R_t x=v} s}$$

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

$\text{not_blocked}(s, t)$

$\exists b_1 b_2. s.B(t) = b_1 \ ++ \ [(x, v)] \ ++ \ b_2$

$\text{no_pending}(b_1, x)$

$$s \xrightarrow{R_t \ x=v} s$$

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

$$s \xrightarrow{W_t x=v} s \oplus \langle [B := s.B \oplus (t \mapsto ([x, v] ++ s.B(t)))] \rangle$$

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

$\text{not_blocked}(s, t)$

$s.B(t) = b \text{ ++ } [(x, v)]$

$s \xrightarrow{\tau_t x=v}$

$s \oplus \langle [M := s.M \oplus (x \mapsto v)] \rangle \oplus \langle [B := s.B \oplus (t \mapsto b)] \rangle$

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

L: Lock

$$s.L = \text{NONE}$$

$$s.B(t) = []$$

$$s \xrightarrow{L_t} s \oplus \langle [L := \text{SOME}(t)] \rangle$$

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 $\tau_t x=v$ steps to empty the buffer and then proceed.

x86-TSO Abstract Machine: Behaviour

U: Unlock

$$s.L = \text{SOME}(t)$$

$$s.B(t) = []$$

$$s \xrightarrow{U_t} s \oplus \langle [L := \text{NONE}] \rangle$$

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

x86-TSO Abstract Machine: Behaviour

B: Barrier

$$\frac{s.B(t) = []}{s \xrightarrow{B_t} s}$$

If t 's store buffer is empty, it can execute an MFENCE.

Notation Reference

SOME and NONE construct optional values

(\cdot, \cdot) builds tuples

$[\]$ builds lists

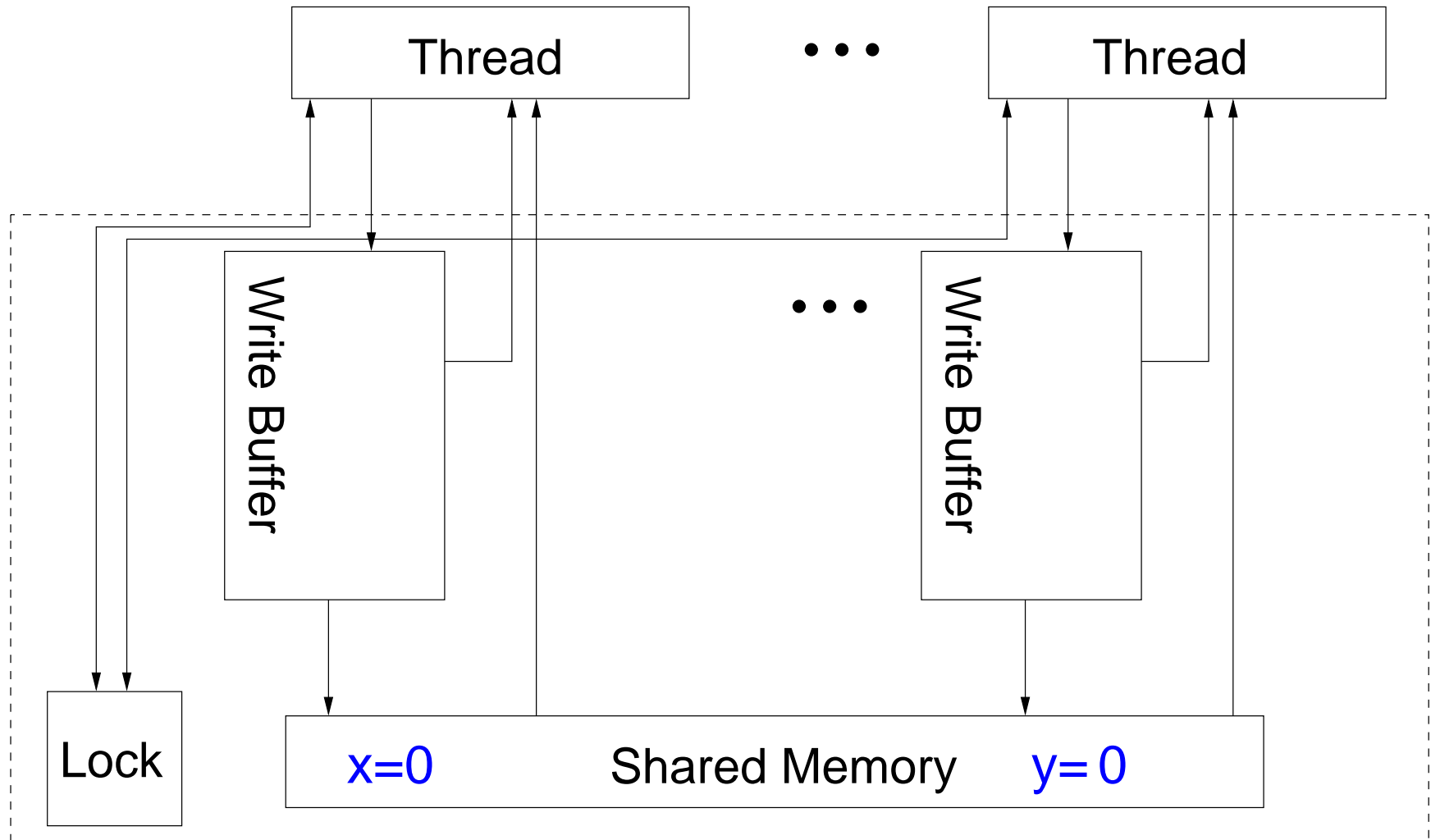
++ appends lists

$\cdot \oplus \langle \cdot := \cdot \rangle$ updates records

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

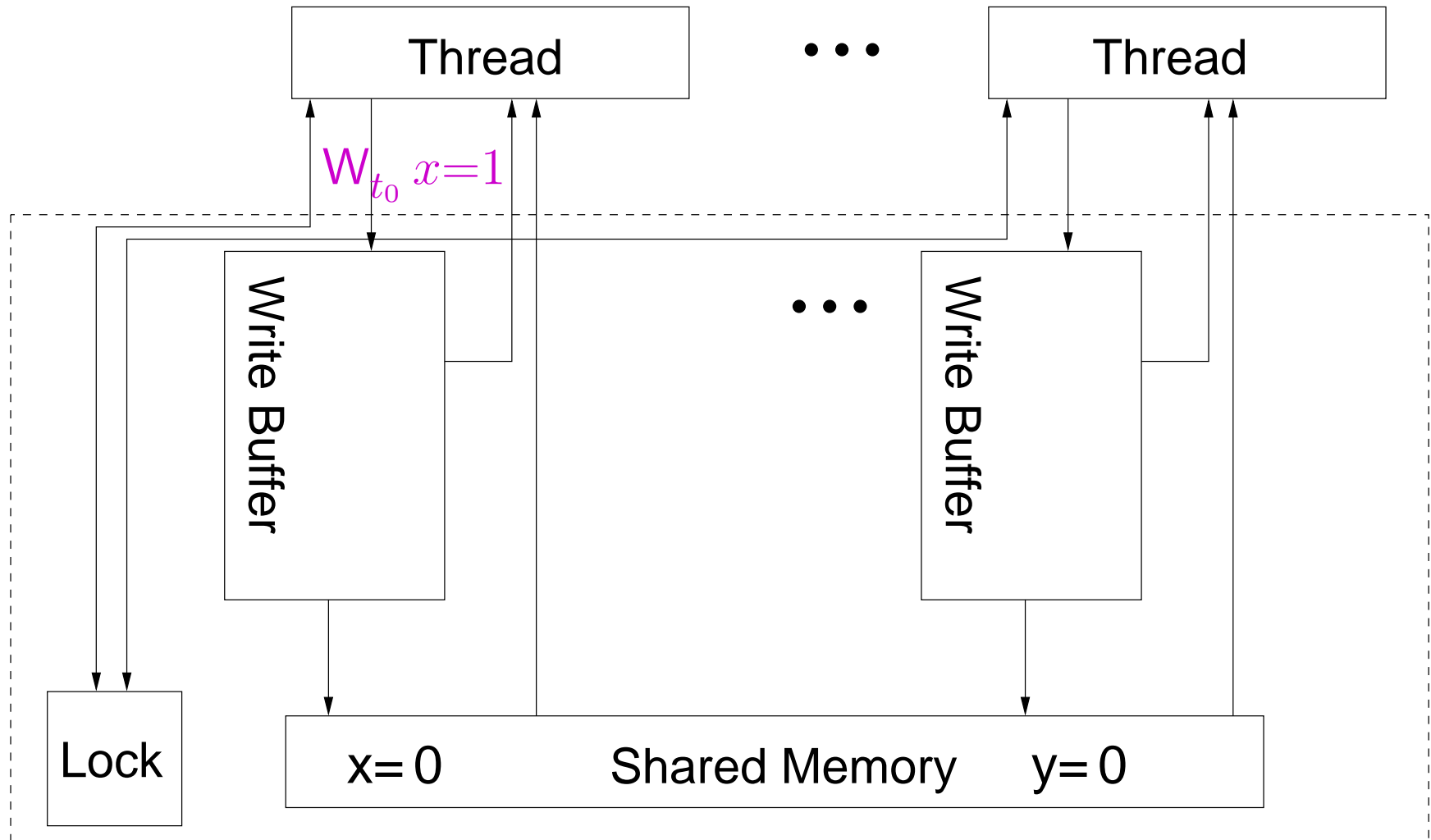
First Example, Revisited

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)



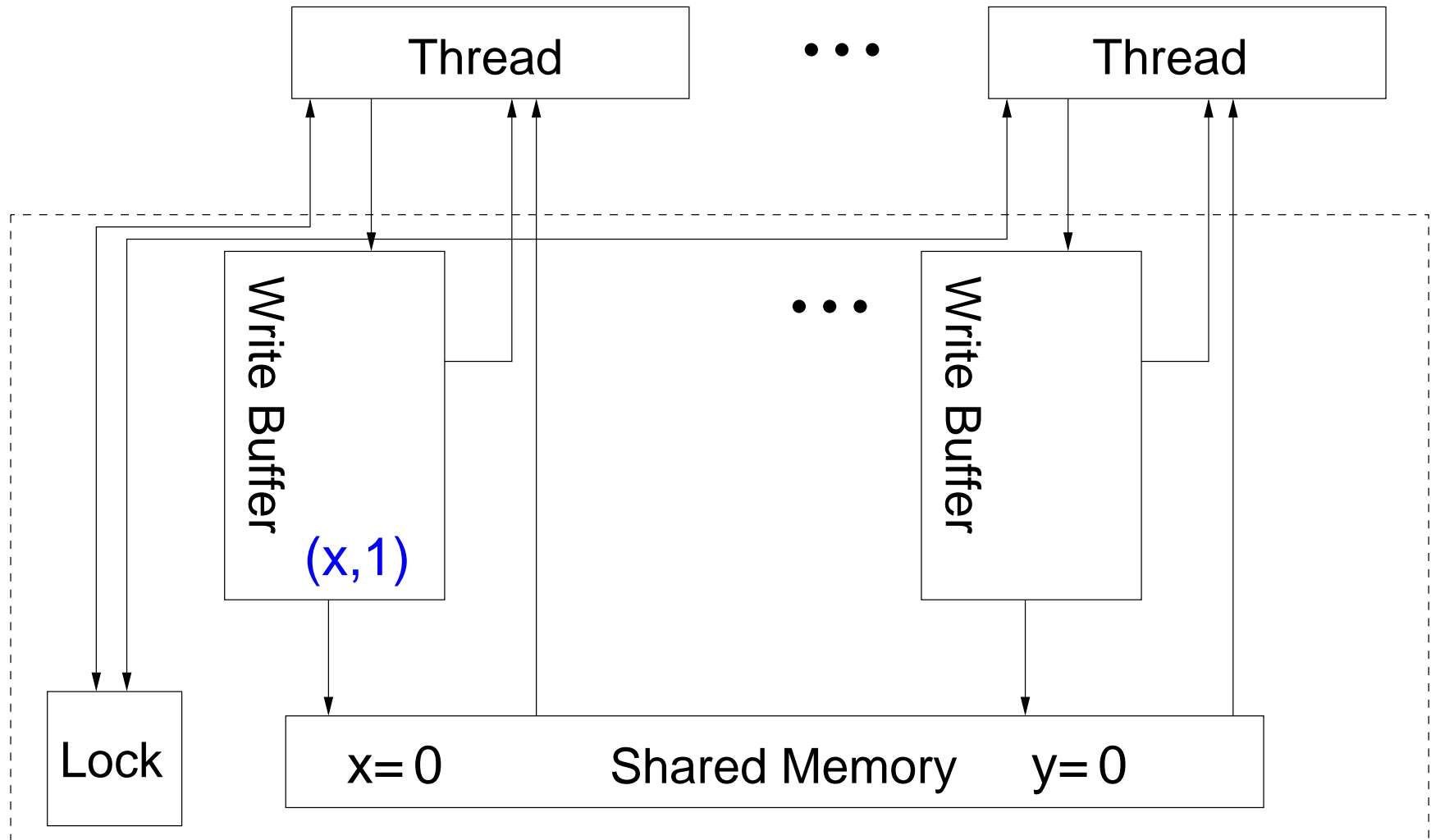
First Example, Revisited

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)



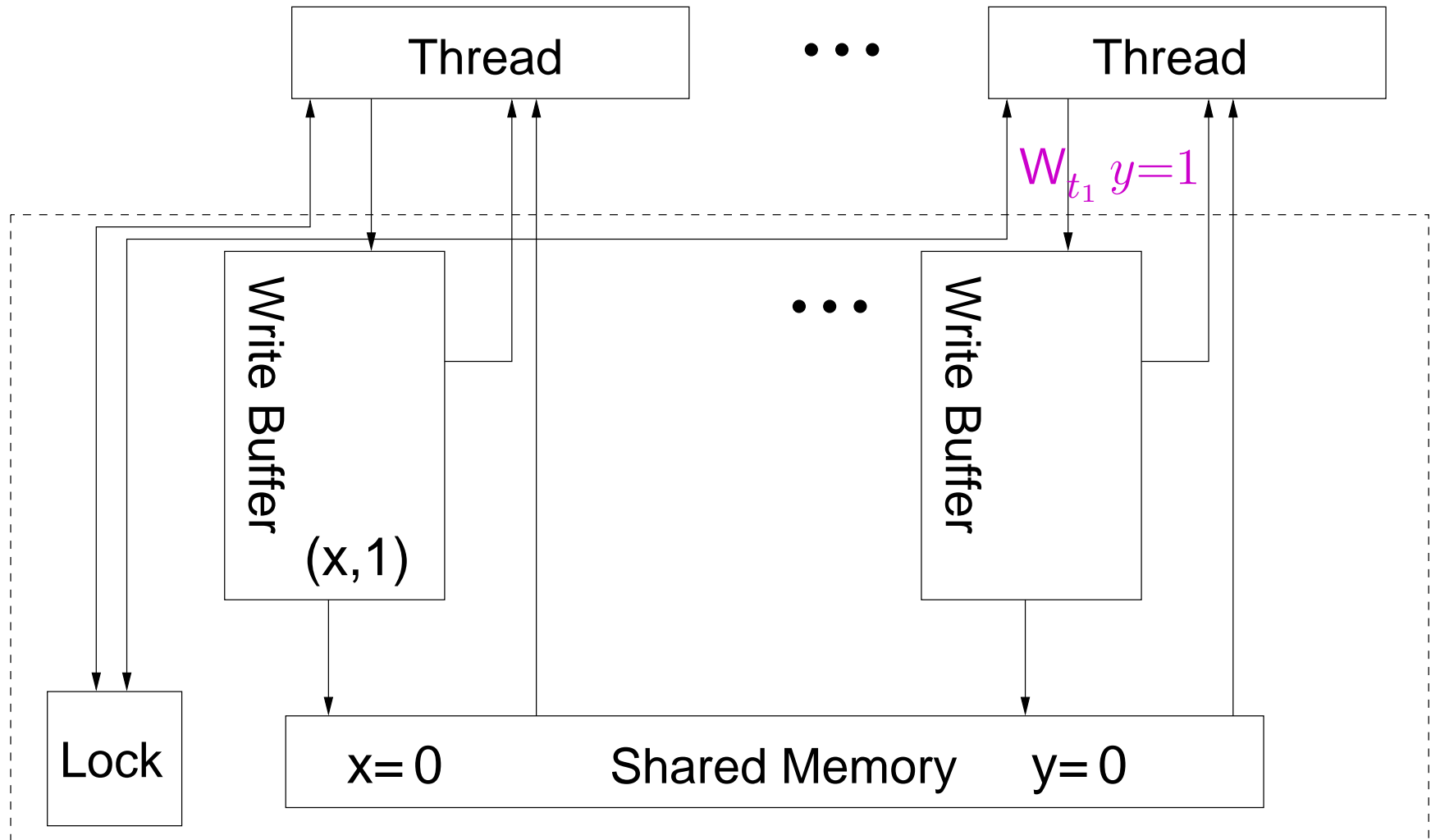
First Example, Revisited

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)



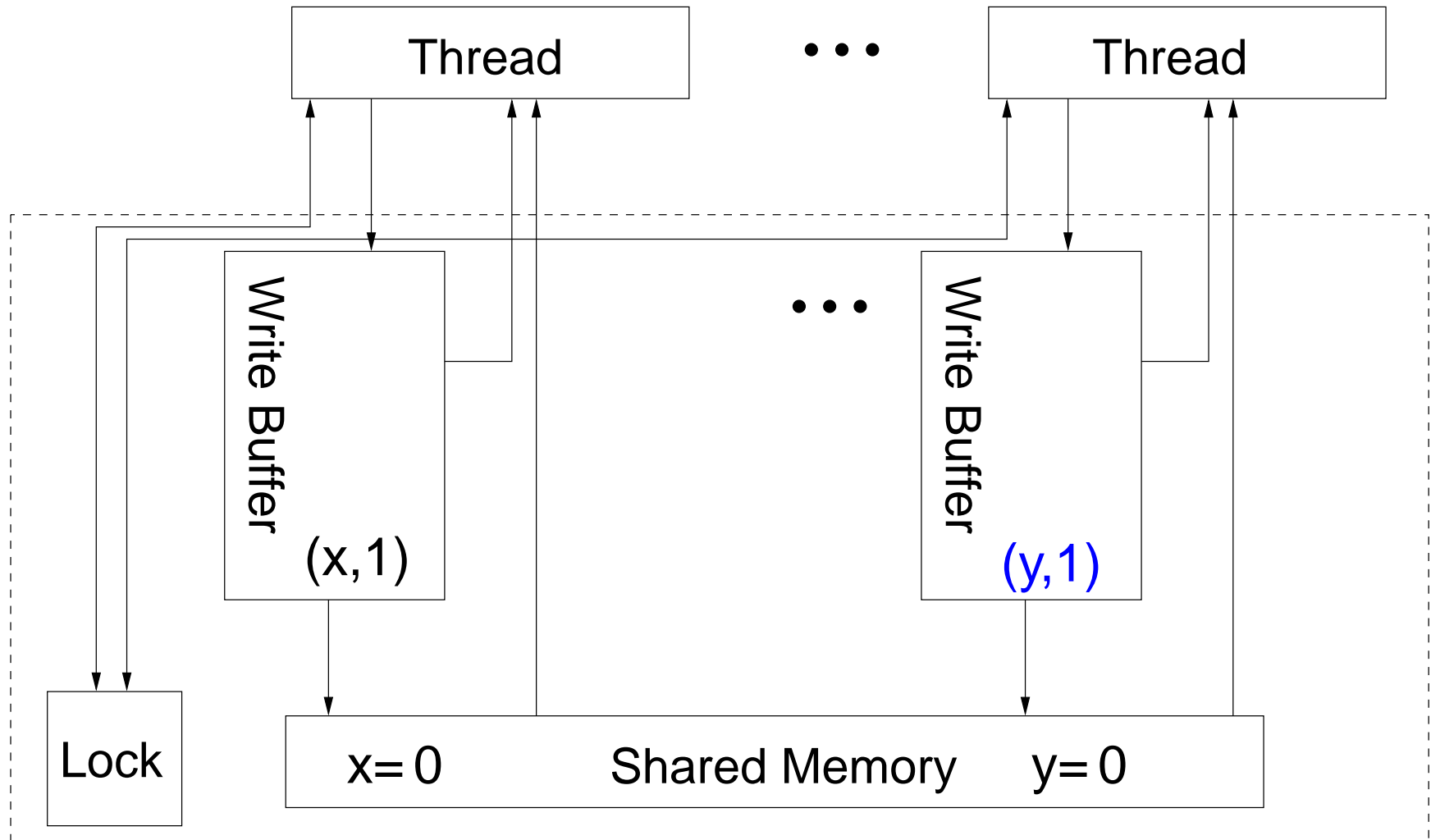
First Example, Revisited

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)



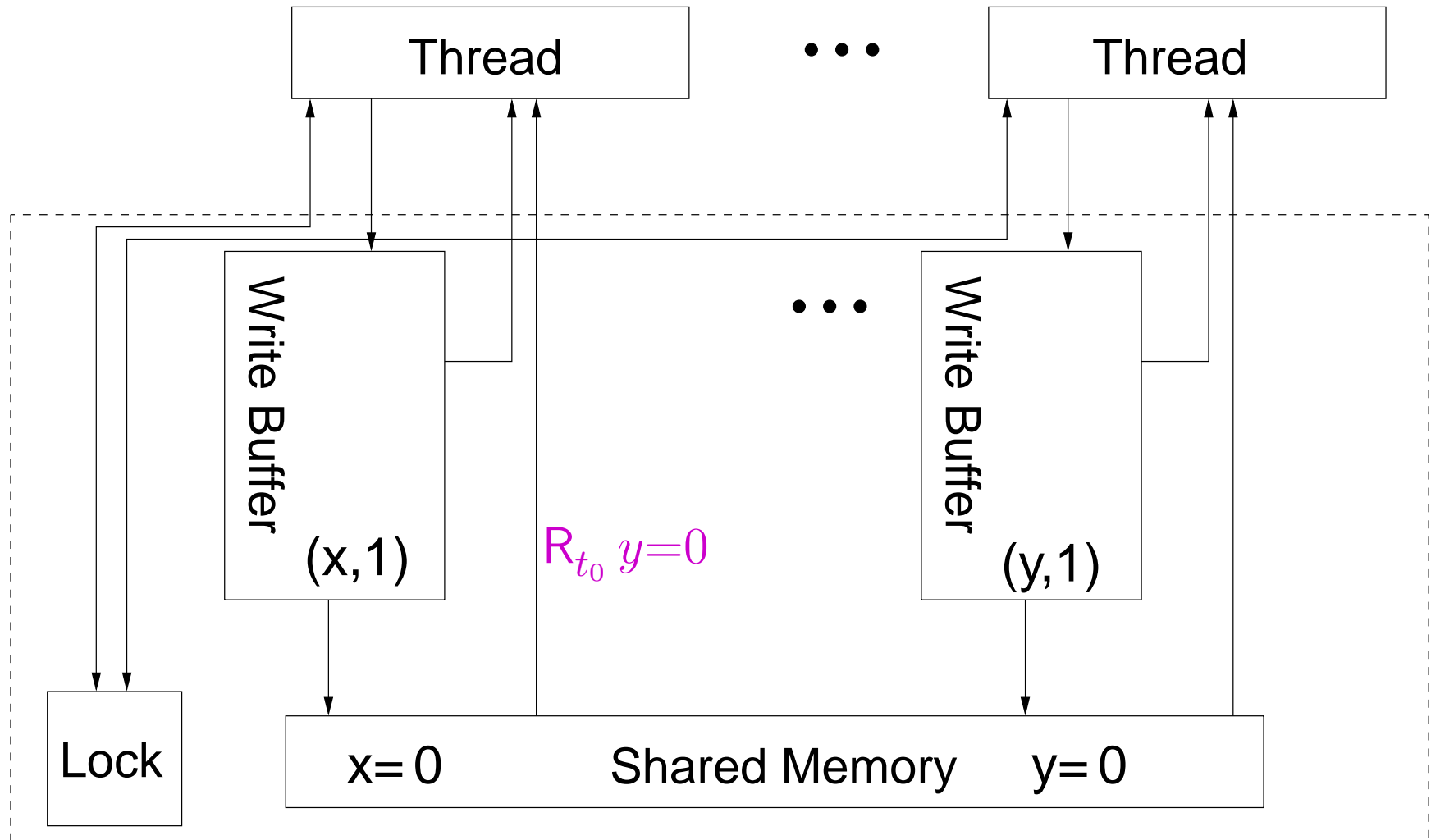
First Example, Revisited

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)



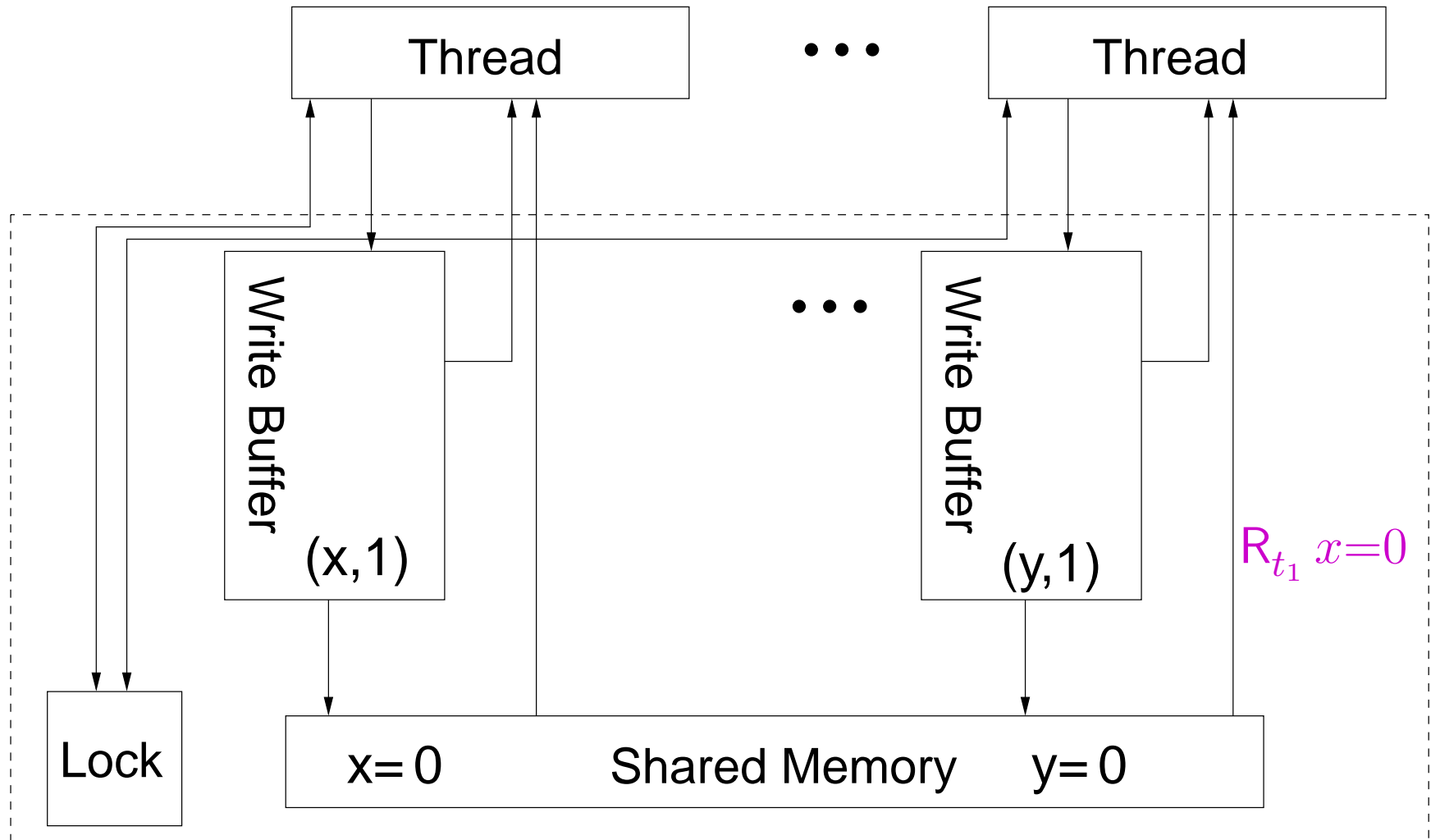
First Example, Revisited

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)



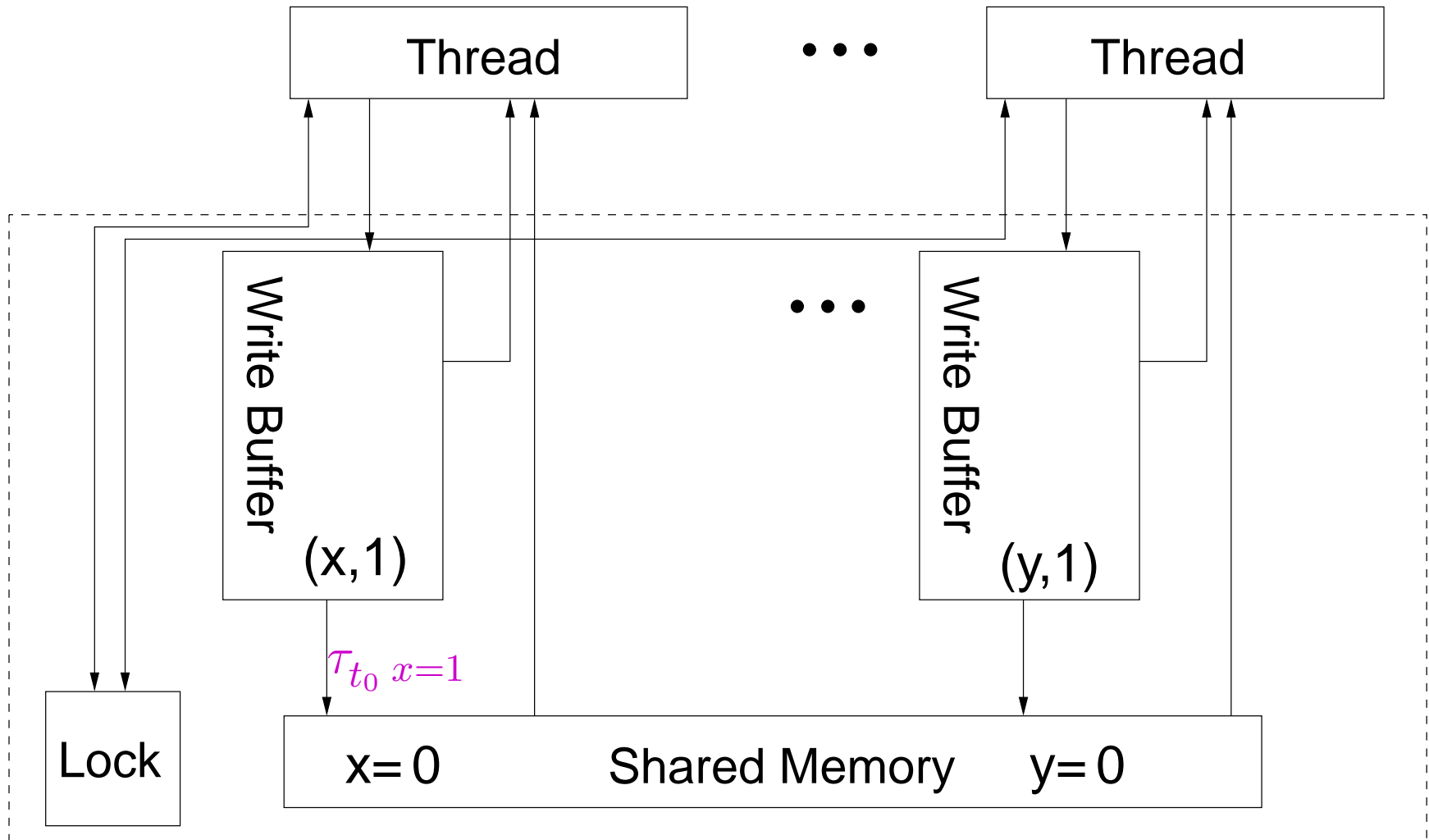
First Example, Revisited

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)



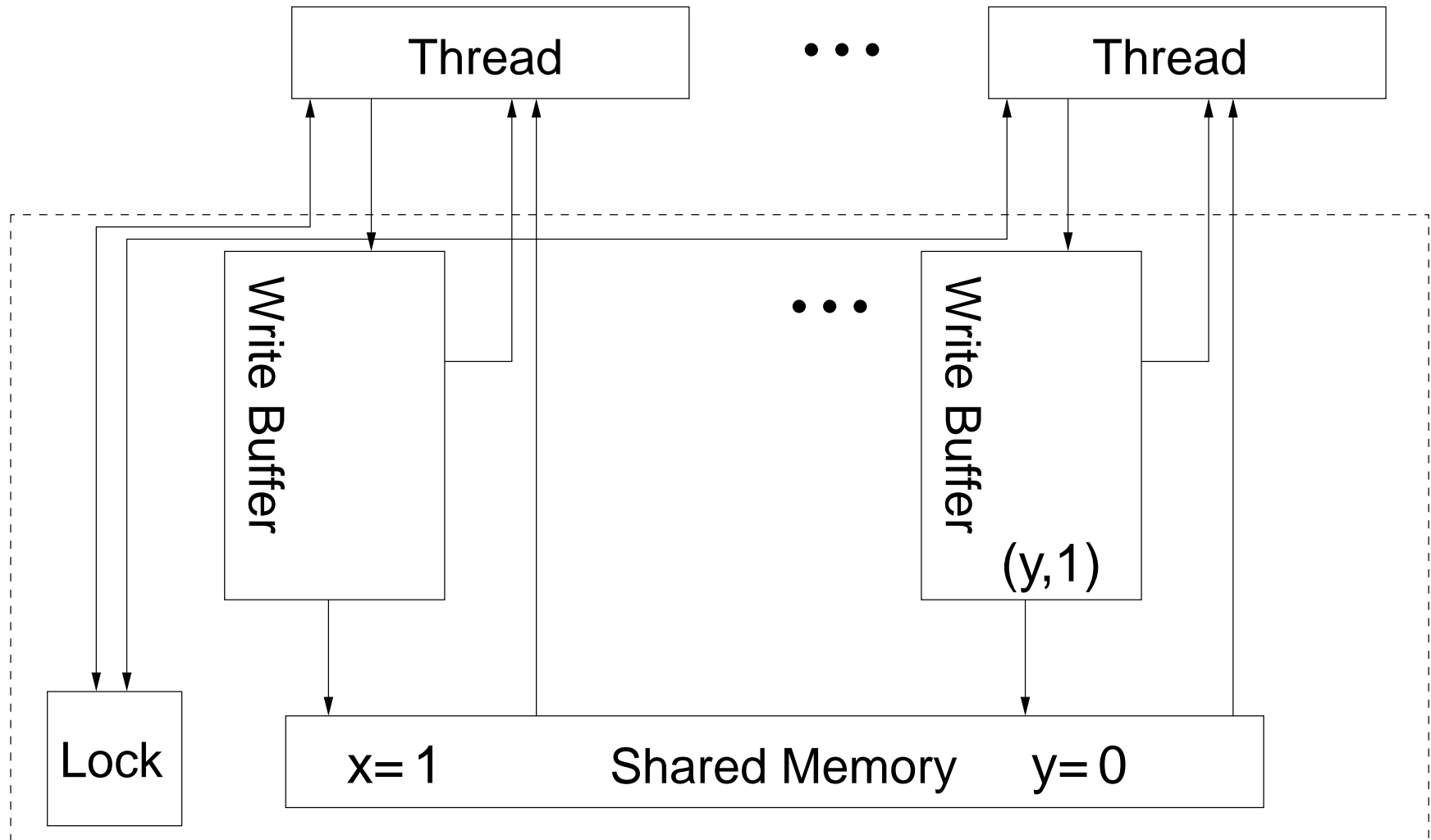
First Example, Revisited

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)



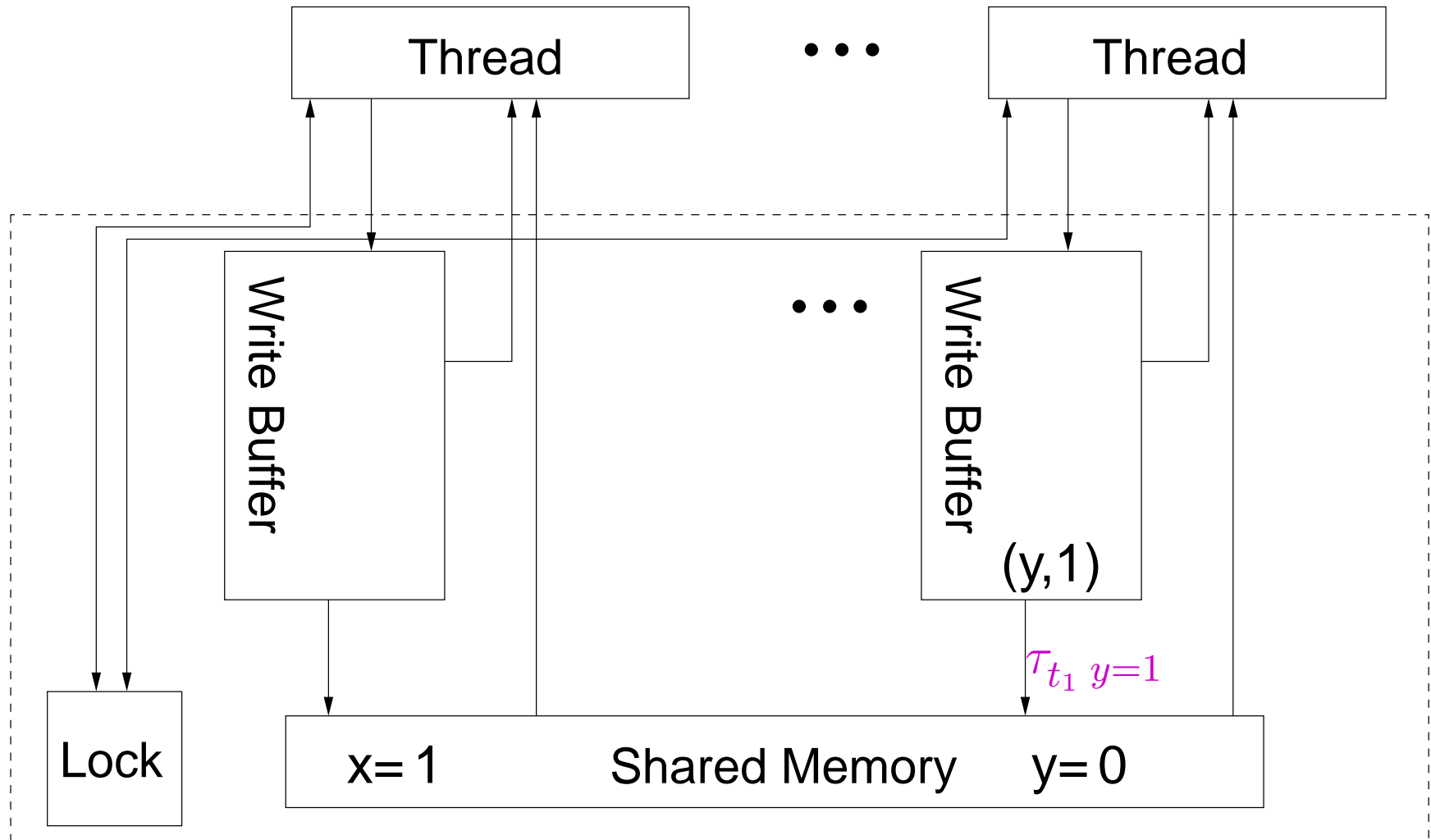
First Example, Revisited

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)



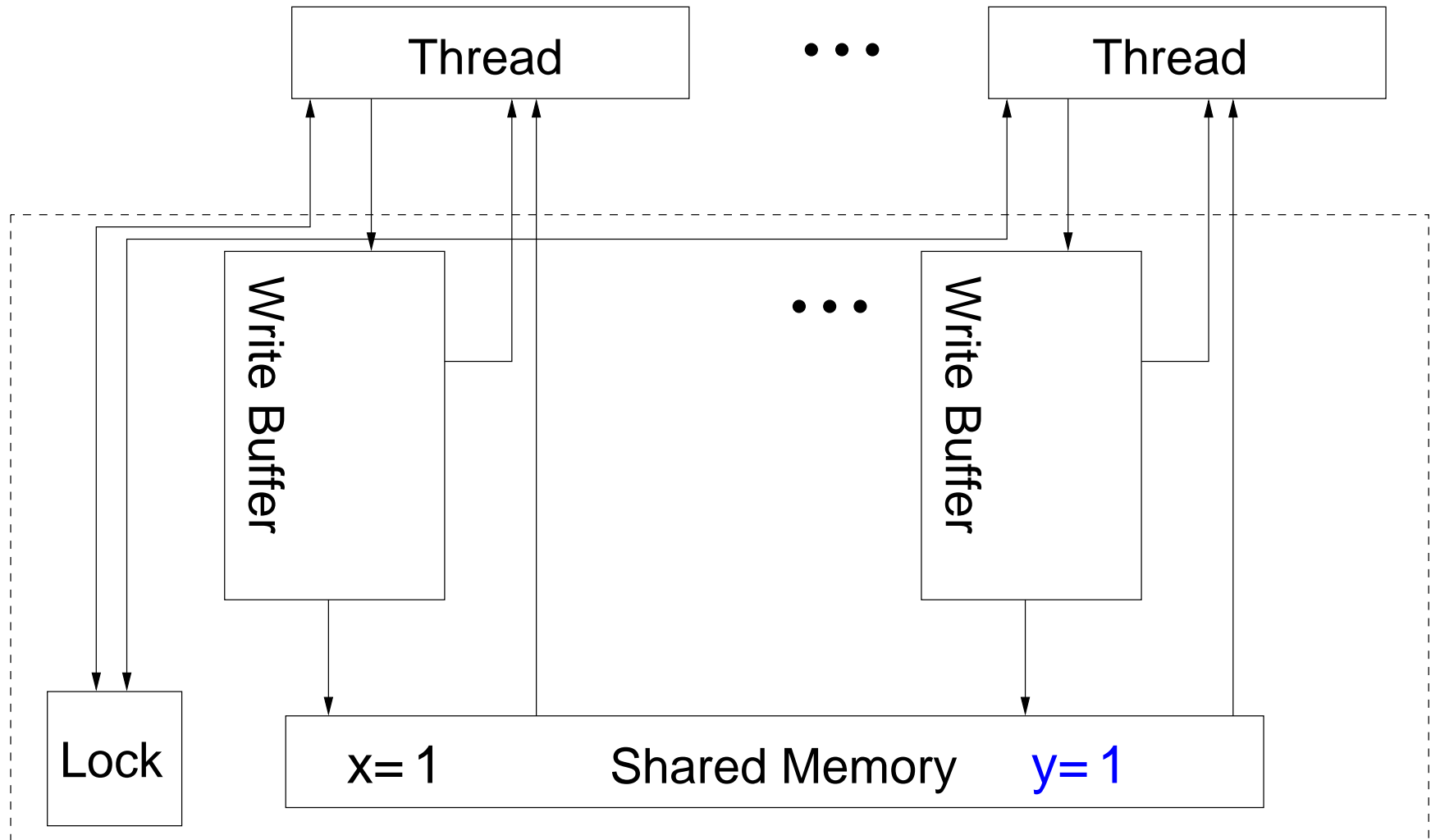
First Example, Revisited

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)



Barriers and LOCK'd Instructions, recap

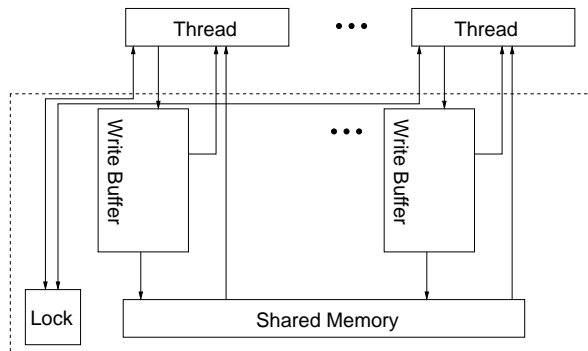
- MFENCE memory barrier
 - flushes local write buffer
- LOCK'd instructions (atomic INC, ADD, CMPXCHG, etc.)
 - flush local write buffer
 - globally locks memory

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 \wedge Thread 1:EBX=0	

NB: both are *expensive*

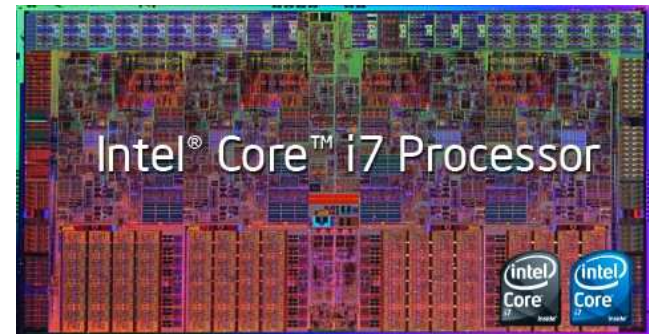
NB: This is an *Abstract Machine*

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



\supseteq beh

\neq 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

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.

Validating the Semantics

Testing tools:

- LITMUS, runs litmus tests on real h/w
- MEMEVENTS, symbolically finds all possible results
- EMUL, daemon emulator

(Also modelling & testing instruction semantics)

Informal vendor support

Formalized in theorem prover (HOL4)

One reasonable model

Liveness

Question: is every memory write guaranteed to eventually propagate from store buffer to shared memory?

We tentatively assume so (with a progress condition on machine traces).

AMD: yes

Intel: unclear

(ARM: yes)

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

x86-TSO: The Axiomatic Model

The abstract machine generates x86-TSO executions stepwise.

The axiomatic model says whether a complete candidate execution is admitted by x86-TSO.

Events: $i:W_t x=v$, $i:R_t x=v$ and $i:B_t$ as before, but with unique ids i .

Event structures E :

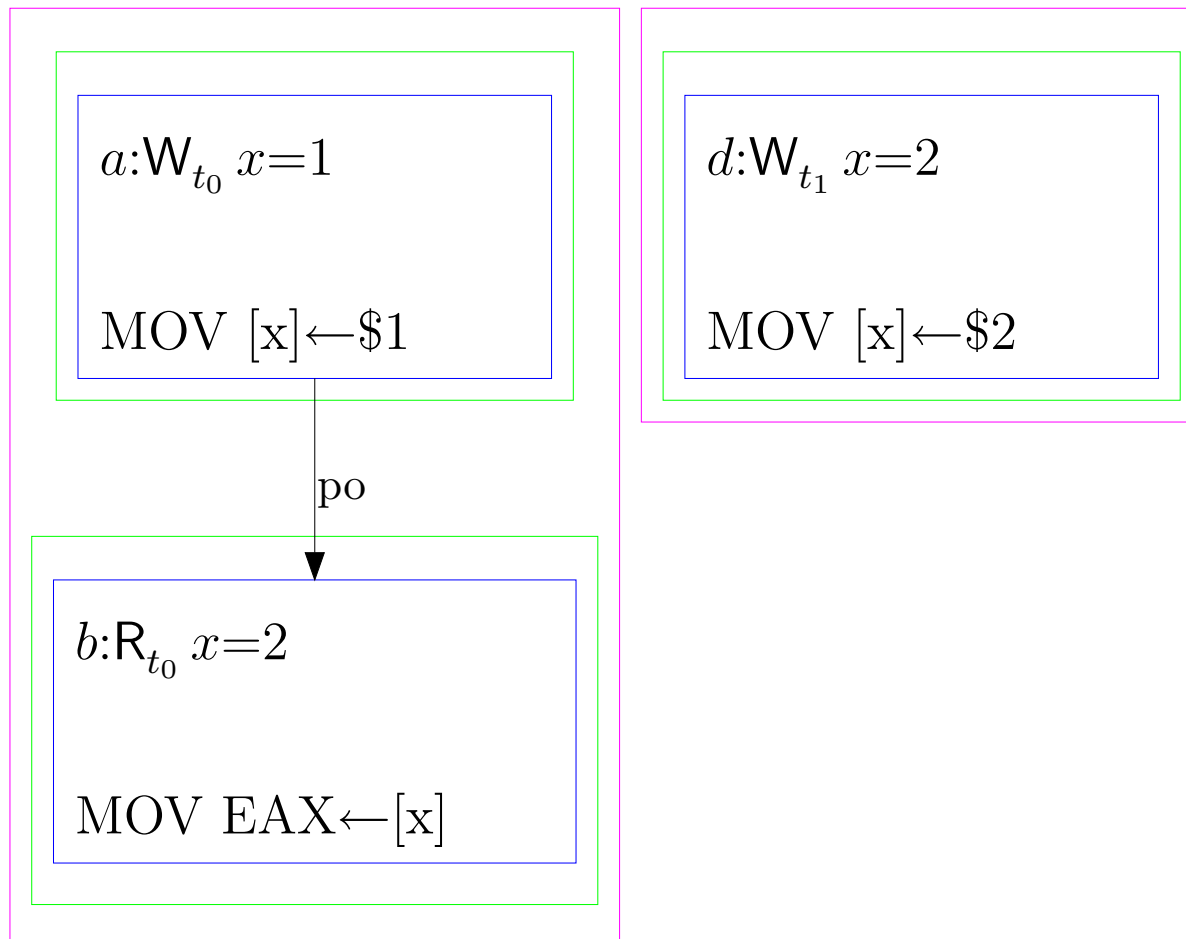
- a set of events
- program order (po) and intra-instruction causality order (iico) over them (strict partial orders)
- an atomicity relation over them (a partial equivalence relation)

Execution witness X :

execution_witness =

$\langle \langle$ *memory_order* : event reln;
rfmap : event reln;
initial_state : addr \rightarrow value $\rangle \rangle$

tso1	Thread t_0	Thread t_1
	MOV [x]←\$1	MOV [x]←\$2
	MOV EAX←[x]	



Axioms: Memory Order

$X.memory_order$ is a partial order over memory read and write events of E

$X.memory_order$, when restricted to the write events of E , is a linear order.

Axioms: Reads-from map

The r fmap only relates such pairs with the same address and value:

reads_from_map_candidates E $rfmap =$

$$\forall (ew, er) \in rfmap. (er \in \text{mem_reads } E) \wedge (ew \in \text{mem_writes } E) \wedge \\ (\text{loc } ew = \text{loc } er) \wedge (\text{value_of } ew = \text{value_of } er)$$

Auxiliary functions over events: `loc`, `value_of`

Auxiliary functions over event structures:

`mem_reads`, `mem_writes`, `mem_accesses`, `mfences`

Axioms: *check_rfmap_written*

Let po_iico E be the union of (strict) program order and intra-instruction causality.

Check that the *rfmap* relates a read to the most recent preceding write.

$previous_writes$ E er $<_{order}$ =

$$\{ew' \mid ew' \in mem_writes\ E \wedge ew' <_{order} er \wedge (loc\ ew' = loc\ er)\}$$

$check_rfmap_written$ E X =

$$\forall(ew, er) \in (X.rfmap).$$

$$ew \in maximal_elements (previous_writes\ E\ er\ (<_{X.memory_order}) \cup \\ previous_writes\ E\ er\ (<_{(po_iico\ E)})) \\ (<_{X.memory_order})$$

Axioms: *check_rfmap_initial*

And similarly for the initial state:

$\text{check_rfmap_initial } E \ X =$

$\forall er \in (\text{mem_reads } E \setminus \text{range } X.\text{rfmap}).$

$(\exists l. (\text{loc } er = l) \wedge (\text{value_of } er = X.\text{initial_state } l)) \wedge$

$(\text{previous_writes } E \ er (<_{X.\text{memory_order}}) \cup$

$\text{previous_writes } E \ er (<_{(\text{po_iico } E)}) = \{\})$

Axioms: R/A Program Order

Program order is included in memory order, for a memory read before a memory access (mo_po_read_access) (SPARCV8's **LoadOp**):

$\forall er \in (\text{mem_reads } E). \forall e \in (\text{mem_accesses } E).$

$$er <_{(\text{po_iico } E)} e \implies er <_{X.\text{memory_order}} e$$

Axioms: W/W Program Order

Program order is included in memory order, for a memory write before a memory write (mo_po_write_write) (the SPARCv8 **StoreStore**):

$\forall ew_1 ew_2 \in (\text{mem_writes } E).$

$$ew_1 <_{(\text{po_iico } E)} ew_2 \implies ew_1 <_{X.\text{memory_order}} ew_2$$

Axioms: Fencing

Program order is included in memory order, for a memory write before a memory read, *if* there is an MFENCE between (mo_po_mfence).

$$\forall ew \in (\text{mem_writes } E). \forall er \in (\text{mem_reads } E). \forall ef \in (\text{mfences } E). \\ (ew <_{(\text{po_iico } E)} ef \wedge ef <_{(\text{po_iico } E)} er) \implies ew <_{X.\text{memory_order}} er$$

Axioms: Locked Instructions

Program order is included in memory order, for any two memory accesses where at least one is from a LOCK'd instruction (mo_po_access/lock):

$$\forall e_1 e_2 \in (\text{mem_accesses } E). \forall es \in (E.\text{atomicity}). \\ ((e_1 \in es \vee e_2 \in es) \wedge e_1 <_{(\text{po_iico } E)} e_2) \implies e_1 <_{X.\text{memory_order}} e_2$$

Axioms: Atomicity

The memory accesses of a LOCK'd instruction occur atomically in memory order (*mo_atomicity*), i.e., there must be no intervening memory events.

Further, all program order relationships between the locked memory accesses and other memory accesses are included in the memory order (this is a generalization of the SPARCV8 **Atomicity** axiom):

$$\forall es \in (E.\textit{atomicity}). \forall e \in (\textit{mem_accesses } E \setminus es).$$
$$(\forall e' \in (es \cap \textit{mem_accesses } E). e <_{X.\textit{memory_order}} e') \vee$$
$$(\forall e' \in (es \cap \textit{mem_accesses } E). e' <_{X.\textit{memory_order}} e)$$

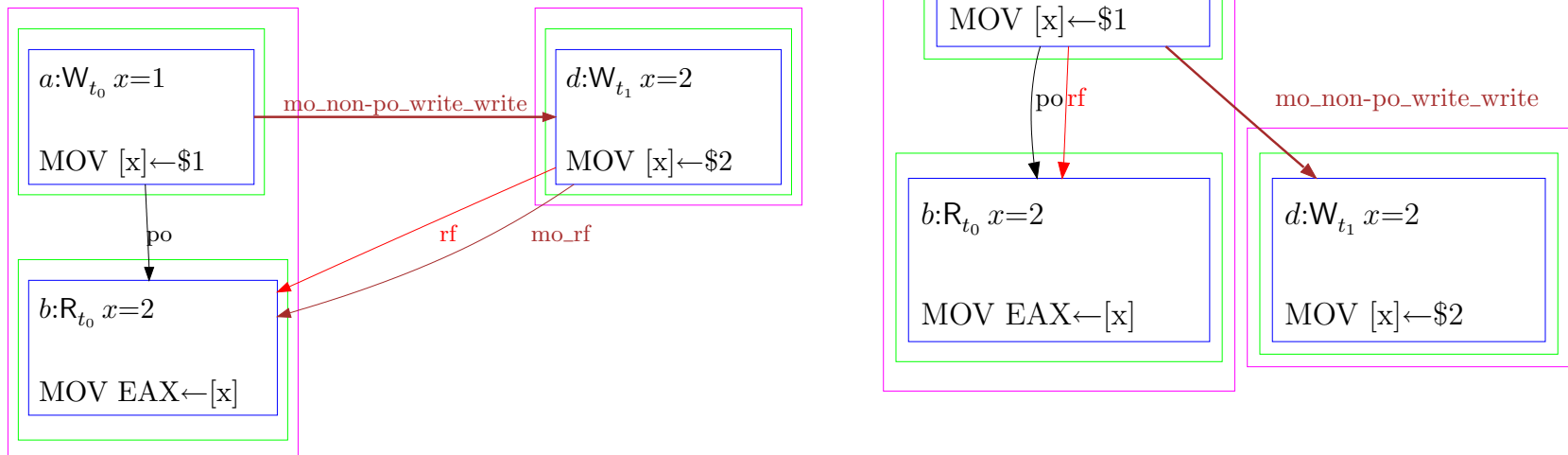
Axioms: Infinite Executions

For this course, consider only finite executions (E with finite sets of events).

(In general, we require that the prefixes of the memory order are all finite, ensuring that there are no limit points, and, to ensure that each write eventually takes effect globally, there must not be an infinite set of reads unrelated to any particular write, all on the same memory location (this formalizes the SPARCv8 **Termination** axiom).)

Say $\text{valid_execution } E \ X$ iff all the above hold.

Example



Equivalence of the two models

Loosely:

Theorem 1 *For any abstract-machine execution with*

- *atomic sets properly bracketed by lock/unlock pairs*
- *non- τ /L/U events E*
- *ordered according to po_iico*

there is an X such that $\text{valid_execution } E X$, with the $X.\text{memory_order}$ the order in which machine memory reads and buffer flushes occurred.

Theorem 2 *For any axiomatic $\text{valid_execution } E X$, there is some abstract-machine path which when τ /L/U-erased has the same events (ordered according to po_iico and with atomic sets properly bracketed by lock/unlock pairs) in which memory reads and buffer flushes respect $X.\text{memory_order}$.*

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

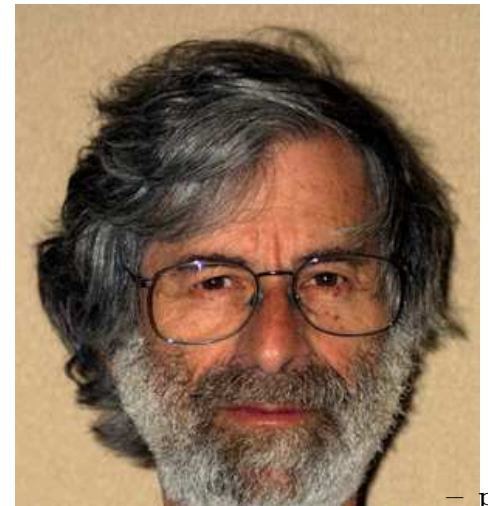
Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.

Leslie Lamport, 1979



Data Race Freedom (DRF)

Basic Principle (you'd hope):

If a program has no data races in any sequentially consistent (SC) execution, then any relaxed-memory execution is equivalent to some sequentially consistent execution.

NB: premise only involves SC execution.

Data Race Freedom (DRF)

Basic Principle (you'd hope):

If a program has no data races in any sequentially consistent (SC) execution, then any relaxed-memory execution is equivalent to some sequentially consistent execution.

NB: premise only involves SC execution.

But what *is* a data race?

what does *equivalent* mean?

What is a data race — first attempt

Suppose SC executions are traces of events

- $R_t x=v$ for thread t reading value v from address x
- $W_t x=v$ for thread t writing value v to address x

(erase τ 's, and ignore lock/unlock/mfence for a moment)

Then say an SC execution has a data race if it contains a pair of adjacent accesses, by different threads, to the same location, that are not both reads:

- $\dots, R_{t_1} x=u, W_{t_2} x=v, \dots$
- $\dots, W_{t_1} x=u, R_{t_2} x=v, \dots$
- $\dots, W_{t_1} x=u, W_{t_2} x=v, \dots$

What is a data race — for x86

1. Need not consider write/write pairs to be races
2. Have to consider SC semantics for LOCK'd instructions (and MFENCE), with events:
 - L_t at the start of a LOCK'd instruction by t
 - U_t at the end of a LOCK'd instruction by t
 - B_t for an MFENCE by thread t
3. Need not consider a LOCK'd read/any write pair to be a race

Say an *x86 data race* is an execution of one of these shapes:

- $\dots, R_{t_1} x=u, W_{t_2} x=v, \dots$
- $\dots, R_{t_1} x=u, L_{t_2}, \dots, W_{t_2} x=v, \dots$

(or v.v. No unlocks between the L_{t_2} and $W_{t_2} x=v$)

DRF Principle for x86-TSO

Say a program is *data race free* (DRF) if no SC execution contains an x86 data race.

Theorem 3 (DRF) *If a program is DRF then any x86-TSO execution is equivalent to some SC execution.*

(where *equivalent* means that there is an SC execution with the same subsequence of writes and in which each read reads from the corresponding write)

Proof: via the x86-TSO axiomatic model

Scott Owens, ECOOP 2010



Happens-Before Version

Here:

An SC race is two adjacent conflicting actions.

In the setting of an axiomatic model:

Often have a *happens before* partial order over events

...and a race is two conflicting actions that aren't ordered by happens-before

What is a data race, again?

acquire_mutex(l)

write $x \leftarrow 1$

release_mutex(l)

acquire_mutex(l)

read x

release_mutex(l)

Simple Spinlock

acquire_mutex(x)

critical section

release_mutex(x)

Simple Spinlock

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

critical section

```
release_mutex(x)
```

Invariant:

lock taken if $x \leq 0$

lock free if $x=1$

Simple Spinlock

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

critical section

```
release_mutex(x)
```

Simple Spinlock

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

critical section

$x \leftarrow 1$ *OR* atomic_write(x, 1)

Simple Spinlock

```
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 LOCK'd instructions in implementations of Linux spinlocks.

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

x = 0

acquire

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	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

⋮		$y \leftarrow v_2$
⋮		⋮
$x \leftarrow v_1$		x
⋮		⋮

Triangular Races

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

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	X
⋮	⋮

Not triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	$X \leftarrow w$
⋮	⋮

Triangular Races

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

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	X
⋮	⋮

Not triangular race

⋮	$y \leftarrow v_2$
⋮	mfence
$X \leftarrow v_1$	X
⋮	⋮

Triangular Races

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

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$x \leftarrow v_1$	x
⋮	⋮

Not triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$x \leftarrow v_1$	lock x
⋮	⋮

Triangular Races

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

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	X
⋮	⋮

Not triangular race

⋮	lock $y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	X
⋮	⋮

Triangular Races

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

Triangular race

⋮		$y \leftarrow v_2$
⋮		⋮
$X \leftarrow v_1$		X
⋮		⋮

Triangular race

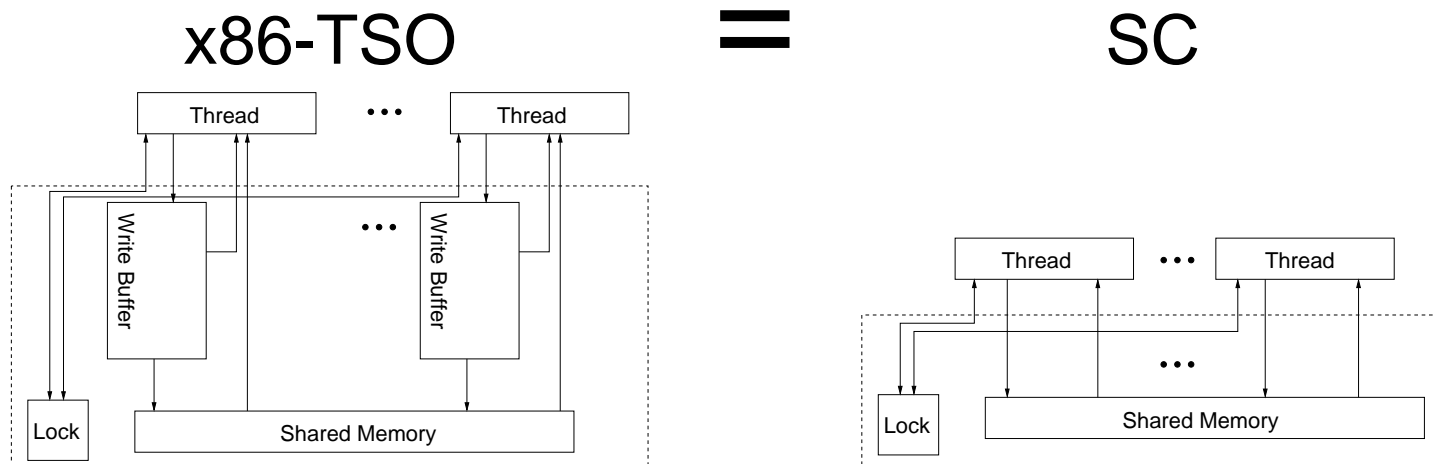
⋮		$y \leftarrow v_2$
⋮		⋮
lock $X \leftarrow v_1$		X
⋮		⋮

TRF Principle for x86-TSO

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

Theorem 4 (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 5 *Any well-synchronized program that uses the spinlock correctly is TRF.*

Theorem 6 *Spinlock-enforced critical sections provide mutual exclusion.*

Other Applications

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

Reflections

We've introduced a plausible model, x86-TSO.

Usable:

- as spec to test h/w against
- to give a solid intuition for systems programmers
- to develop reasoning tools above
- to develop code testing tools above (daemonized emulator)

In terms of that model, we can clearly see why (and indeed *prove*) that that Linux spinlock optimisation is correct.



x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors by P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. CACM, to appear.

A Better x86 Memory Model: x86-TSO by S. Owens, S. Sarkar, and P. Sewell. TPHOLs 2009.

Reasoning about the Implementation of Concurrency Abstractions on x86-TSO by S. Owens. ECOOP 2010.

A Rely-Guarantee proof system for x86-TSO assembly code programs by Tom Ridge. VSTTE 2010.

The Semantics of x86-CC Multiprocessor Machine Code by S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. POPL 2009.

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

Hardware Models

SC

(x86-)TSO

PSO

RMO

Power/ARM

IA-64 (Itanium)

Alpha

Coherence

Coherence

Minimal property of a 'cache coherent' system:

for each location independently, each thread sees the same order of writes to that location.

(Ambiguity: does a thread 'see' its own writes?)

Abstract Machine: bi-FIFO from each processor to each memory cell?

Then need to add some additional facilities for programming synchronisation

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

Power ISA 2.06 and ARM v7

Visible behaviour is much weaker & more subtle than x86

Key spec concept: actions being *performed* or *observed*.

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 define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).

Power ISA 2.06 and ARM v7

Visible behaviour is much weaker & more subtle than x86

Key spec concept: actions being *performed* or *observed*.

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 define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).

But it's *subjunctive*: it refers to a hypothetical store by P2.

Power ISA Version 2.06 Revision B (July 23, 2010):

http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf

see Book II Chapters 1 and 4

By Example

Work in progress: developing usable Power/ARM models.

For now: introduce behaviour by example, with some experimental data.

Store Buffering (SB)

SB

x86

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 \wedge Thread 1:EBX=0	

SB

PPC

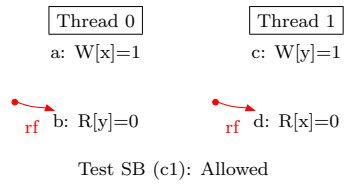
Thread 0	Thread 1
stw r6,0(r4)	stw r6,0(r5)
lwz r2,0(r5)	lwz r2,0(r4)
Initial state: 0:r4=x \wedge 0:r5=y \wedge 0:r6=1 \wedge 1:r4=x \wedge 1:r5=y \wedge 1:r6=1	
Allowed: 0:r2=0 \wedge 1:r2=0	

SB

ARM

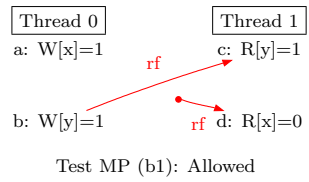
Thread 0	Thread 1
STR R6, [R4]	STR R6, [R5]
LDR R2, [R5]	LDR R2, [R4]
Initial state: 0:R4=x \wedge 0:R5=y \wedge 0:R6=1 \wedge 1:R4=x \wedge 1:R5=y \wedge 1:R6=1	
Allowed: 0:R2=0 \wedge 1:R2=0	

Store Buffering (SB)



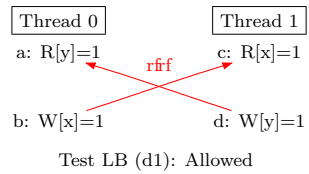
Power 6: Observed, 4e7/2e9

Message Passing (MP)



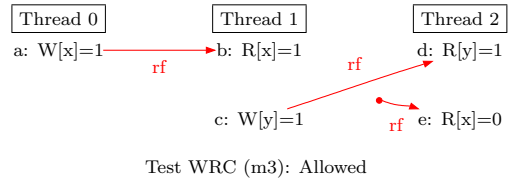
Power 6: Observed, 9e6/2e9

Load Buffering (LB)



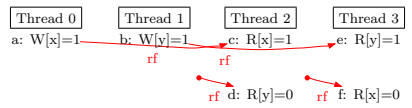
Power 6: Not observed, but architecturally permitted

Write-to-Read Causality (WRC)



Power 6: Observed 1e4/1e9

Independent Reads of Independent Writes (IRIW)



Test IRIW (ppc-cookbook6.5-amd6-cpp.iriw.nofence): Allowed

Power 6: Observed 259/8e8

So how can we ever write concurrent code?

- Dependencies (various kinds)
- Memory Barriers (various kinds)
- Load-reserve/Store-conditional exclusive pairs

Dependencies

From a load to a load: address dependencies

(data-flow path through registers and arith/logical operations from the value of the first load to the address of the second)

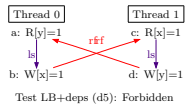
(intensional — even if the address doesn't numerically depend...)

(not via memory)

From a load to a store: address, data & control dependencies

(as above, or to the value stored, or data flow to the test of an intermediate conditional branch)

LB+deps

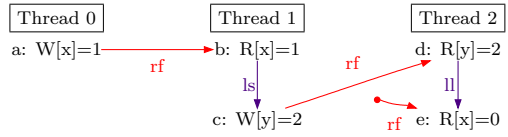


LB+deps

ARM

Thread 0	Thread 1
LDR R2, [R5] AND R3, R2, #0 STR R1, [R3,R4]	LDR R2, [R4] AND R3, R2, #0 STR R1, [R3,R5]
Initial state: $0:R1=1 \wedge 0:R4=x \wedge 0:R5=y$ $\wedge 1:R1=1 \wedge 1:R4=x \wedge 1:R5=y$	
Forbidden: $0:R2=1 \wedge 1:R2=1$	

WRC+deps



Test WRC+deps (isa1v2): Allowed

Power 6: observed 1e4/1e9

Memory Barriers

Power: ptesync, hwsync, lwsync, eieio, mbar, isync

ARM: DSB, DMB

Memory Barriers

Power: ptesync, hwsync, lwsync, eieio, mbar, isync

ARM: DSB, DMB

For each applicable pair a_i, b_j the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

- A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- B includes all applicable storage accesses by any such processor or mechanism that are performed after a *Load* instruction executed by that processor or mechanism has returned the value stored by a store that is in B .

Memory Barriers

Power: ptesync, hwsync, lwsync, eieio, mbar, isync

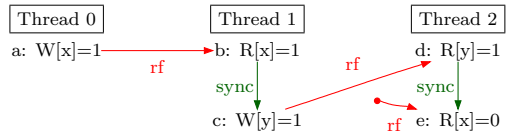
ARM: DSB, DMB

For each applicable pair a_i, b_j the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the Memory Coherence Requirements, before b_j is performed with respect to that processor or mechanism.

- A includes all instructions that are performed by any such processor or mechanism before the barrier is created.
- B includes all instructions that are performed by any such processor or mechanism after a Load or Store instruction execution has returned the value stored by a store that is in B .



WRC+syncs (or +DMBs)



Test WRC+syncs (m3s): Forbidden

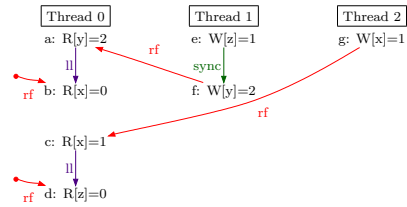
Power 6: not observed (0/8e8)

Load-reserve/Store-conditional

lwarx/LDREX atomically (a) loads, and (b) creates a reservation for this “storage granule”

stwcx/STREX atomically (a) stores and (b) sets a flag, *if* the storage granule hasn't been written to by any thread in the meantime

Reads from Different Writes (RDW)

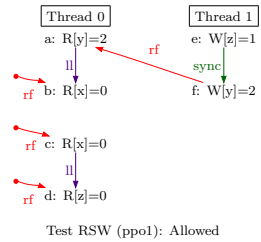


Test RDW (pp03): Forbidden

Power 6: Not observed (0/1e9)

Power 5: Not observed (0/1e10)

Reads from Same Write (RSW)



Power 6: Not observed (0/2e9)

Power 5: Observed (5e5/2e10)

Sources of H/W Relaxed Memory

Microarchitecture:

- store buffering (hierarchical, split)
- cache protocol (e.g. not waiting for invalidates)
- speculative execution in the pipeline

Changes radically (e.g. Power 5 \mapsto 6 \mapsto 7)

Unclear what it is ... so no pictures

Other Issues

Not touched on:

- precise specification for any of this
- when barriers and/or dependencies suffice
- lwsync, isync
- visibility of register shadowing
- ARM conditional instructions
- progress properties
- other memory types
- self-modifying code
- page table management

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

SPARC

TSO: introduced as SPARC model, fairly clearly defined.

Also two weaker models, PSO and RMO.

But... Solaris always uses TSO. Linux 'uses' RMO, but most processors actually implement TSO

Alpha

Very weak model.

No longer produced — but the Linux memory barrier macros are based on the Alpha memory model

Itanium (IA-64 \neq Intel 64=AMD64)

Axiomatic definition

- unordered load and store
- store-release (become remotely visible to all procs in the same order)
- load-acquire

MP-rel-acq

Thread 0			Thread 1		
st	[x]=1	(write x=1)	ld.acq	r1=[y]	(read.acq y=1)
st.rel	[y]=1	(write.rel y=1)	ld	r2=[x]	(read x=0)
Forbidden					

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

High-level languages

High-level languages are not immune to these problems.

Actually, the situation is even worse:

- the compiler might reorder/remove/add memory accesses;
- and *then* the hardware will do some relaxed execution.

Constant Propagation

$$x = 3287$$

$$y = 7 - x / 2$$



$$x = 3287$$

$$y = 7 - 3287 / 2$$

Constant Propagation

$x = 3287$ \longrightarrow $x = 3287$
 $y = 7 - x / 2$ $y = 7 - 3287 / 2$

Initially $x = y = 0$	
$x = 1$	$\text{if } (x==1) \{$
$\text{if } (y==1)$	$\quad x = 0$
$\text{print } x$	$\quad y=1 \quad \}$

SC: can never print 1

Sun HotSpot JVM or GCJ: always prints 1

Non-atomic Accesses

Consider misaligned 4-byte accesses

Initially `int32_t a = 0`

`a = 0x44332211`

`if a = 0x00002211`

`print "oops"`

Non-atomic Accesses

Consider misaligned 4-byte accesses

```
Initially int32_t a = 0
-----
a = 0x44332211 | if a = 0x00002211
                  print "oops"
```

(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

What about multi-word high-level language values?

Defining PL Memory Models

Defining PL Memory Models

Option 1: Don't. No Concurrency

Poor match for current trends

Defining PL Memory Models

Option 2: Don't. No Shared Memory

A good match for *some* problems

Erlang, MPI

Defining PL Memory Models

Option 3: Don't. SC Shared Memory, with Races

(What OCaml gives you — but that's not a true concurrent impl.)

(What Haskell gives you for MVars?)

In general, it's going to be expensive...

Naive impl: barriers between *every* memory access

(smarter: analysis to approximate the thread-local or non-racy accesses, but aliasing always hard)

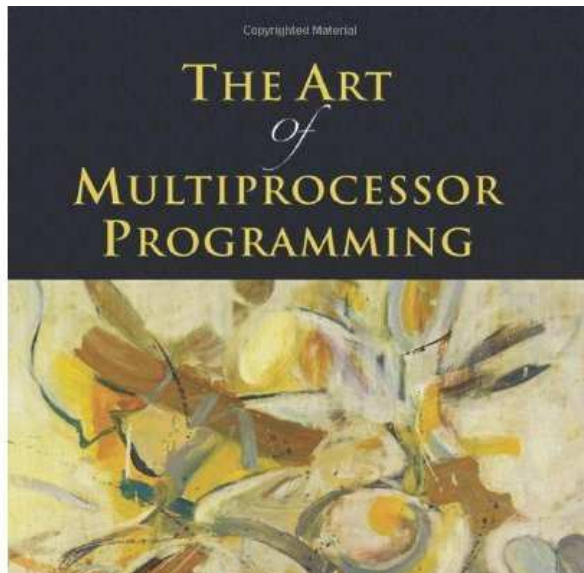
Defining PL Memory Models

Option 4: Don't. Shared Memory, but Language ensures Race-Free

e.g. by ensuring data accesses protected by associated locks

Possible — but inflexible... (pointer aliasing?)

What about all those fancy high-performance concurrent algorithms?



Defining PL Memory Models

Option 5: Don't. Shared Memory, but verify programs in concurrent separation logic and prove that implies race-freedom (and hence all executions are SC)

Appel et al.

great — but “verify”?!

Defining PL Memory Models

Option 6: Don't. Leave it (sort of) up to the hardware

Example: MLton

(high-performance ML-to-x86 compiler, with concurrency extensions)

Accesses to ML refs will exhibit the underlying x86-TSO behaviour

But, they will at least be atomic

Defining PL Memory Models

Option 7: **Do(!)** Use Data race freedom as a *definition*

- programs that are race-free in SC semantics have SC behaviour
- programs that have a race in some execution in SC semantics can behave in any way at all

Sarita Adve & Mark Hill, 1990



Option 7: DRF as a definition

Core of C++0x draft. Hans Boehm & Sarita Adve, PLDI 2008

Pro:

- Simple!
- Strong guarantees for most code
- Allows lots of freedom for compiler and hardware optimisations



‘Programmer-Centric’

Option 7: DRF as a definition

Core of C++0x draft. Hans Boehm & Sarita Adve, PLDI 2008

Con:

- programs that have a race in some execution in SC semantics *can behave in any way at all*
 - Undecidable premise.
 - Imagine debugging: either bug is X ... or there is a potential race in *some* execution
 - No guarantees for untrusted code
- restrictive. Forbids those fancy concurrent algorithms
- need to define exactly what a race is
what about races in synchronisation and concurrent datastructure libraries?

Defining PL Memory Models

Option 8: Don't. Take a concurrency-oblivious language spec (e.g. C) and bolt on a thread library (e.g. Posix or Windows threads)

Posix is sort-of DRF:

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*. Such access is restricted using functions that synchronize thread execution and *also synchronize memory with respect to other threads*

Single Unix SPEC V3 & others

Threads Cannot be Implemented as a Library, Hans Boehm,
PLDI 2005



Defining PL Memory Models

Recall DRF gives no guarantees for untrusted code

Would be a disaster for Java, which relies on unforgeable pointers for its security guarantees

Option 9: Do. DRF + some out-of-thin-air guarantee for all code

Option 9: The Java Memory Model(s)

Java has integrated multithreading, and it attempts to specify the precise behaviour of concurrent programs.

By the year 2000, the initial specification was shown:

- to allow unexpected behaviours;
- to prohibit common compiler optimisations,
- to be challenging to implement on top of a weakly-consistent multiprocessor.

Superseded around 2004 by the JSR-133 memory model.

The Java Memory Model, Jeremy Manson, Bill Pugh & Sarita Adve, POPL05



Option 9: JSR-133

- Goal 1: data-race free programs are sequentially consistent;
- Goal 2: all programs satisfy some memory safety and security requirements;
- Goal 3: common compiler optimisations are sound.

Option 9: JSR-133 — Unsoundness

The model is intricate, and *fails to meet Goal 3*: Some optimisations may generate code that exhibits more behaviours than those allowed by the un-optimised source.

As an example, JSR-133 allows $r2=1$ in the optimised code below, but forbids $r2=1$ in the source code:

$x = y = 0$	
$r1=x$	$r2=y$
$y=r1$	$x=(r2==1)?y:1$

HotSpot optimisation
→

$x = y = 0$	
$r1=x$	$x=1$
$y=r1$	$r2=y$

Jaroslav Ševčík & Dave Aspinall, ECOOP 2008



Defining PL Memory Models

Recall DRF is restrictive, forbidding racy concurrent algorithms (also costly on Power)

And note that C and C++ don't guarantee type safety in any case.

Defining PL Memory Models

Recall DRF is restrictive, forbidding racy concurrent algorithms (also costly on Power)

And note that C and C++ don't guarantee type safety in any case.

Option 10: Do. DRF + low-level atomic operations with relaxed semantics

C++0x approach.

Foundations of the C++ Memory Model, Boehm&Adve PLDI08

Working Draft, Standard for Programming Language C++, N3090, 2010-03

<http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2010/>

with Lawrence Crowl, Paul McKenney, Clark Nelson, Herb Sutter,...

Option 10: C++0x

- normal loads and stores
- lock/unlock
- atomic operations (load, store, read-modify-write, ...)
 - `seq_cst`
 - `relaxed`, `consume`, `acquire`, `release`, `acq_rel`

Idea: if you only use SC atomics, you get DRF guarantee
Non-SC atomics there for experts.

Informal-prose spec.

Formalisation in progress, in HOL — Mark Batty



Option 10: C++0x — Formalisation

Some (easy to fix) ambiguities are made manifest

Suspicion: most of the specification is (informally) axiomatic, except (§29.3:10):

The requirements do allow `r1 == r2 == 42` in the following example, with `x` and `y` initially zero:

Thread 1:

```
r1 = x.load(memory_order_relaxed);  
if (r1 == 42) y.store(r1, memory_order_  
relaxed);
```

Thread 2:

```
r2 = y.load(memory_order_relaxed);  
if (r2 == 42) x.store(42, memory_order_  
relaxed);
```

However, implementations should not allow such

Option 10: C++0x

Basic out-of-thin-air problem from Java, recurring

Solutions?

Dependency tracking

Better type systems?

Dynamic race detection (with H/W assist)? Or SC violation detection?

Wrapping up

Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

Problem: Loose Specifications

Architectures are the key interface between h/w and low-level s/w

(and language definitions between low-level s/w and applications).

They are necessarily *loose specifications*

But informal prose is a *terrible* way to express loose specifications: ambiguous, untestable, and usually wrong.

Instead, architectures should be mathematically rigorous, clarifying precisely just *how* loose one wants them to be.

(common misconception: precise = tight ?)

Problem: Untested Subtlety

For any such subtle and loose specification, how can we have any confidence that it:

- is well-defined?
must be mathematically precise
- has the desired behaviour on key examples?
exploration tools
- is internally self-consistent?
formalisation and proof of metatheory
- is what is implemented by compiler+hw?
testing tools; compiler proof
- is comprehensible to the programmer?
must be maths explained in prose
- lets us write programs above the model?
static analysis/dynamic checker/daemoniac emulator
- is implementable with good performance?
implement...

Problem/Opportunity: Legacy Complexity

Most of these talks have been dominated by complex legacy choices:

- hw: x86, Power, Alpha, Sparc, Itanium
- sw: C, C++ and Java compiler optimisations, language standards and programming idioms

We may be stuck with these - but maybe not... Can we build radically more scalable systems with a better hw/sw or lang/app interface?

The End

Thanks!

