Multicore Semantics and Programming

Peter Sewell Tim Harris

University of Cambridge Oracle

October – November, 2015

These Lectures

Part 1: 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

Part 2: Multicore Programming: Concurrent algorithms (Tim Harris, Oracle)

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

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*

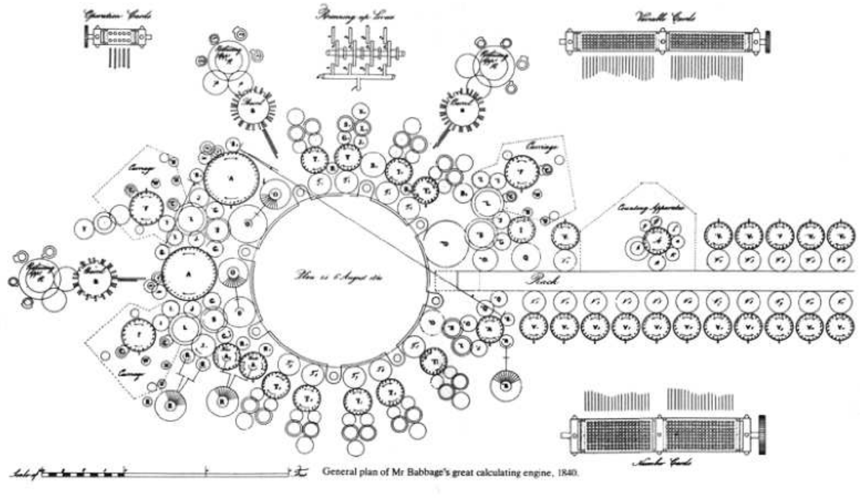
The abstraction of a *memory* goes back some time...

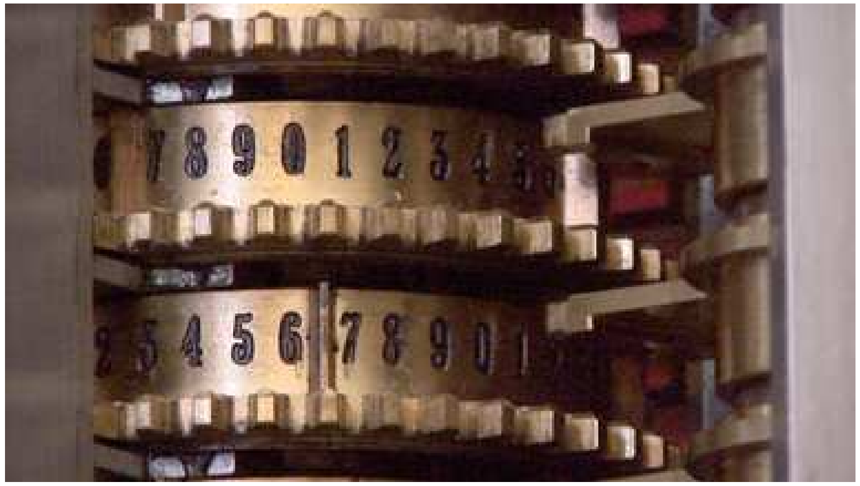
The calculating part of the engine may be divided into two portions

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

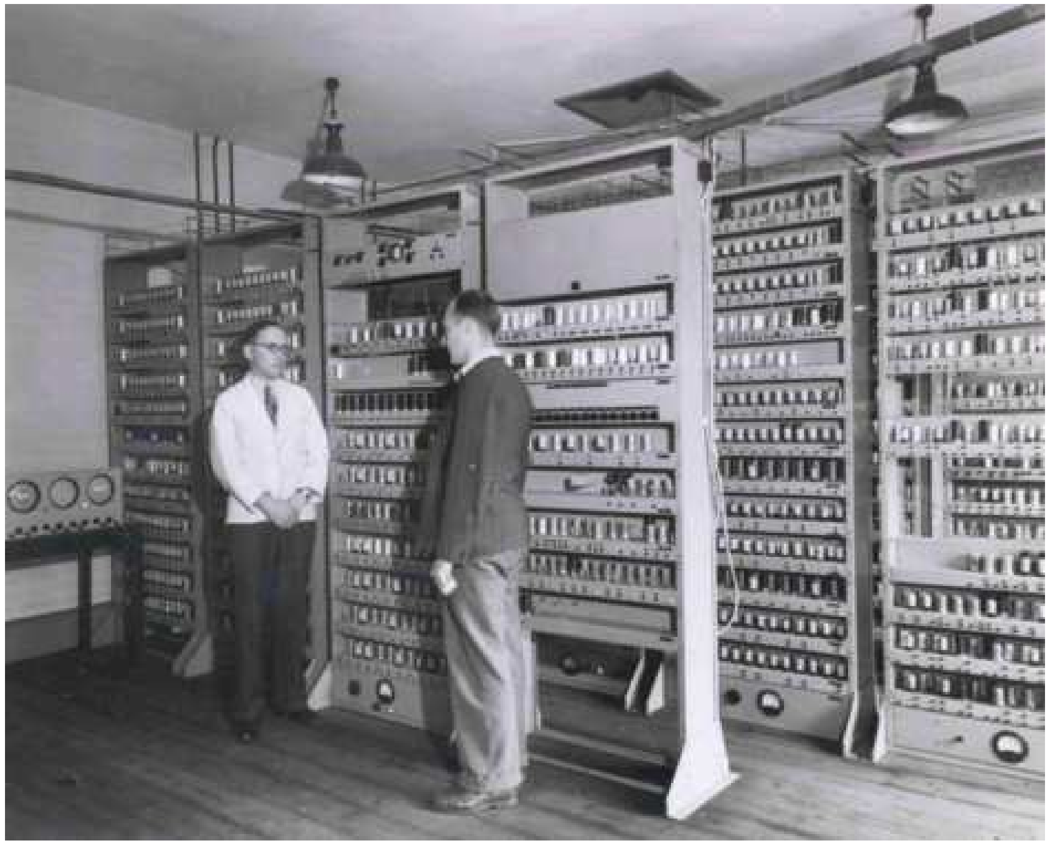
2nd *The* Store *in which all the numbers are originally placed and to which the numbers computed by the engine are returned.*

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


General plan of Mr Babbage's great calculating engine, 1840.



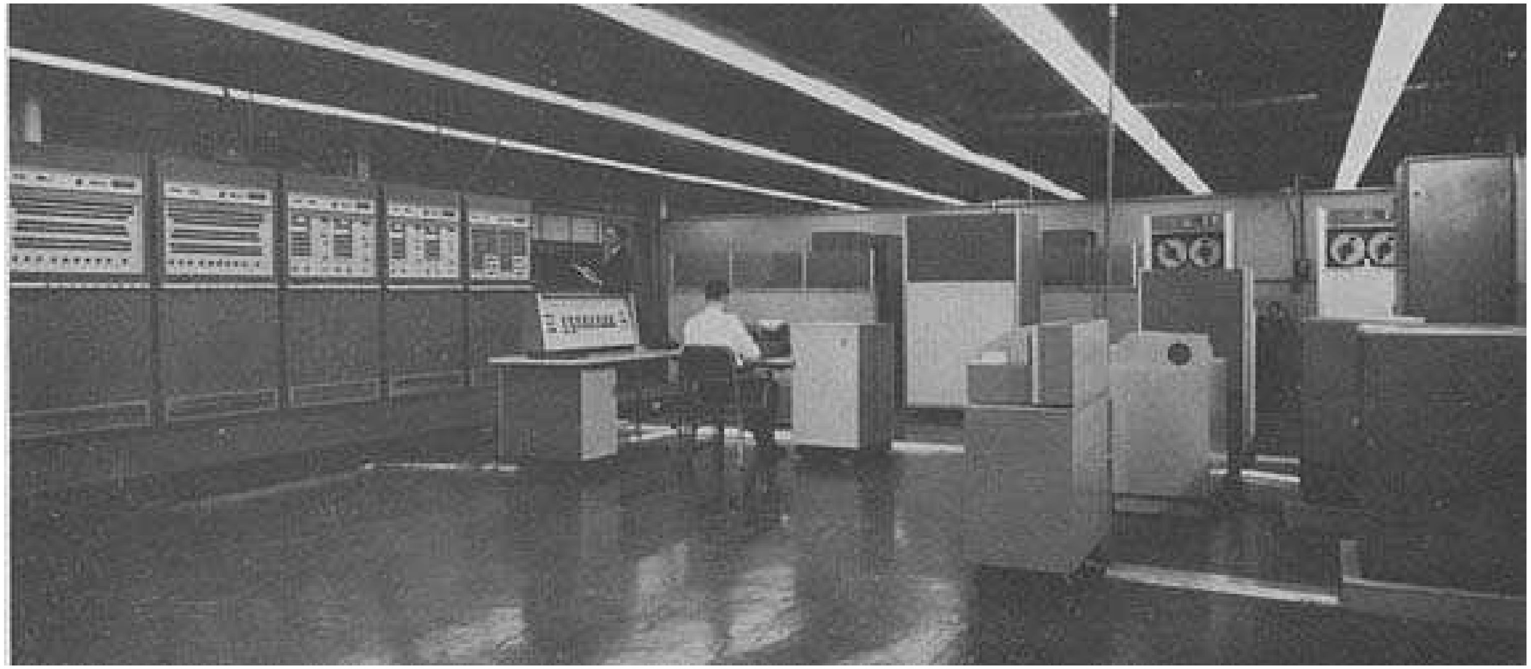
The Golden Age, (1837–) 1945–1962



$$\begin{tikzcd}\boxed{\text{Processor}} \arrow[d, bend right] \\ \boxed{\text{Memory}} \arrow[u, bend right]\end{tikzcd}$$

1962: First(?) Multiprocessor

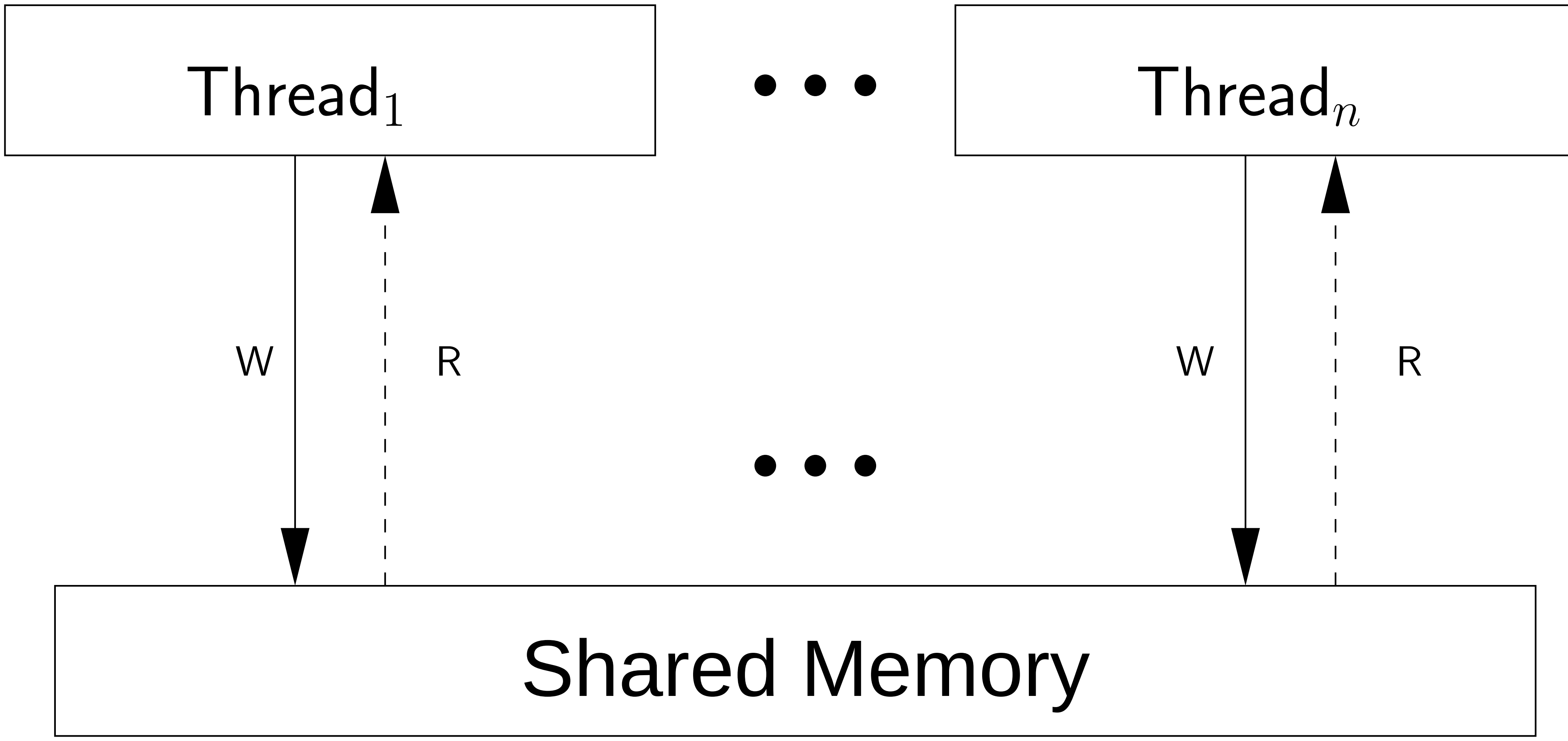
BURROUGHS D825, 1962



‘‘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, 2015

Intel Xeon E7-8895 v3
36 hardware threads

Commonly 4 or 8 hardware threads.

IBM Power 8 server
(up to 1536 hardware threads)

Oracle Sparc, Intel Itanium

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

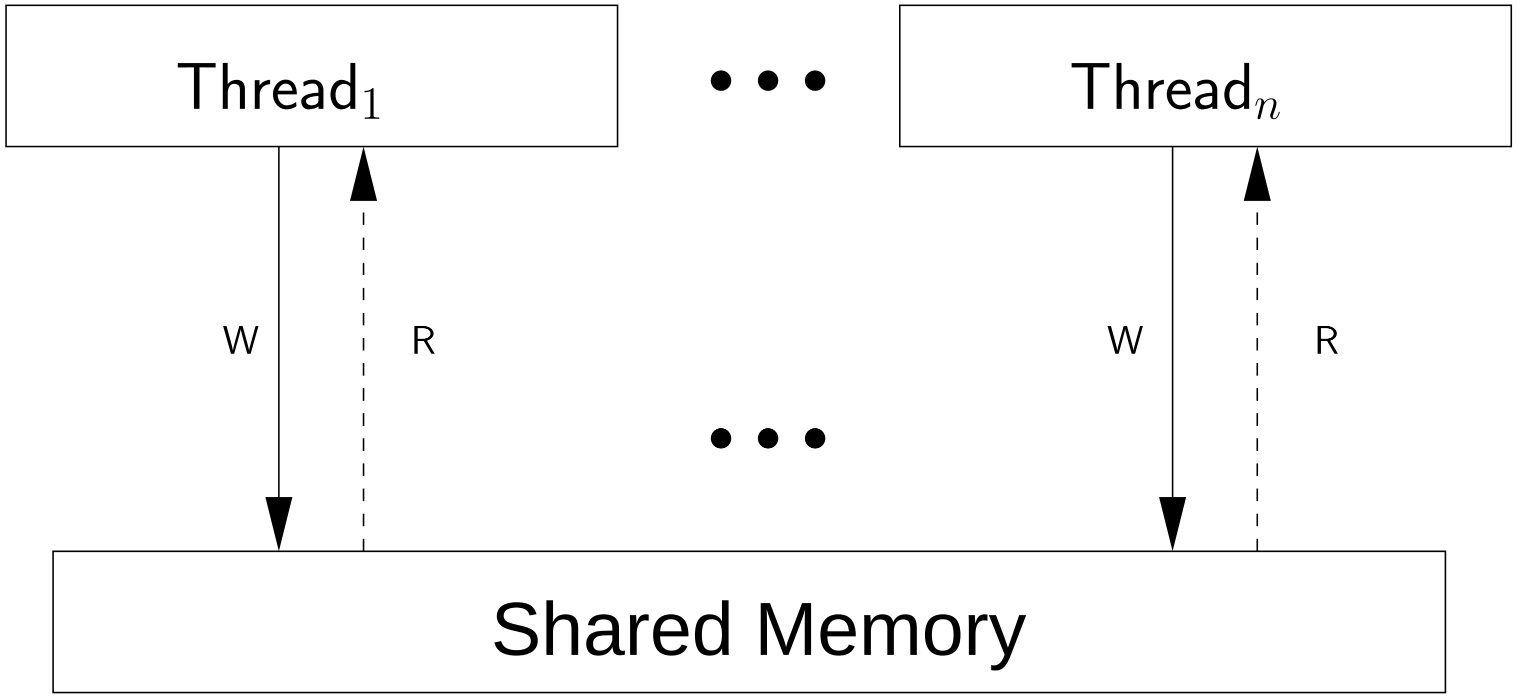
At many scales:

- intra-core
- multicore processors $\leftarrow$ our focus
- ...and programming languages $\leftarrow$ 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
	$\mid$	t:W $x{=}n$	write
	$\mid$	t:R $x{=}n$	read
	$\mid$	$t{:}\tau$	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'

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

$$\frac{}{M \xrightarrow{t:\mathsf{W}\ x=n} M \oplus (x \mapsto n)} \quad \text{M\_WRITE}$$

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:

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

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

$$\frac{\mathit{lookup}\ x\ L = n}{L \xrightarrow{t:\mathsf{R}\ x=n} (t{:}\mathsf{R}\ x{=}n){::}L}\ \textsc{LRead}$$

$$\frac{}{L \xrightarrow{t:\mathsf{W}\ x=n} (t{:}\mathsf{W}\ x{=}n){::}L}\ \textsc{LWrite}$$

Theorem 1 (?) *M_0 and $\mathtt{nil}$ 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 can combine the memory model with a threadwise semantics for a tiny concurrent language....

A Tiny Language: Design Choices

- A concurrent imperative language.
- Distinguish syntactically between (thread-local) “registers” and “memory”.
- Include explicit parallel threads, with thread ids.
- Define an operational semantics that exposes the potential memory events (reads and writes of a value at a memory address) of a thread or process as labelled transitions; this lets us glue it on to an SC or TSO memory.
- Keep the register behaviour internal. Use an explicit register state rather than substitution to highlight the relationship to the memory semantics.
- Otherwise, as simple as possible: just enough computational power to write litmus tests (no loops, no thread creation,

A Tiny Language: Example

Thread 0: $x = 1\,;\ r_0 = y$
Thread 1: $y = 1\,;\ r_1 = x$

and, with the initial register state R_0 for each thread, and an initial SC memory state:

$$\langle t_0 : \langle x = 1\,;\ r_0 = y,\ R_0\rangle \mid t_1 : \langle y = 1\,;\ r_1 = x,\ R_0\rangle,\ \{x \mapsto 0,\ y \mapsto 0\}\rangle$$

A Tiny Language: Syntax

$register,\ r$	register name
$location,\ x,\ y,\ m$	address
$integer,\ n$	integer
$thread\_id,\ t$	thread id
$event\_id,\ a$	event id
k	index variable

Let R range over *register states*, functions from register names r to integers n. Write R_0 for the initial register state in which all registers hold 0.

$expression,\ e$	$::=$		register expression
	$\mid$	n	integer literal
	$\mid$	r	register value
	$\mid$	$e + e'$	plus

A Tiny Language: Syntax

$\mathit{statement},\ s ::=$		statement
$\mid$	$r = e$	compute register value
$\mid$	$r = x$	read from memory
$\mid$	$x = e$	write to memory
$\mid$	$\texttt{if (}e \texttt{==} n\texttt{)}\ s_1\ \texttt{else}\ s_2$	conditional
$\mid$	$s_1\,\texttt{;}\ s_2$	sequential composition
$\mid$	$\texttt{skip}$	empty statement
$\mathit{thread},\ T ::=$		thread
$\mid$	$t : \langle s,\ R \rangle$	id, statement, reg state
$\mathit{process},\ P ::=$		process
$\mid$	T	thread
$\mid$	$P \mid P'$	parallel composition

That was just the syntax — now we’ll be precise about the permitted behaviours of programs

Defining the Semantics: expressions

$\boxed{\langle e, R\rangle \rightarrow n}$ in register state R, e evalutes to n

$$\frac{}{\langle n, R\rangle \rightarrow n} \quad \text{E\_INT}$$

$$\frac{R(r) = n}{\langle r, R\rangle \rightarrow n} \quad \text{E\_REG}$$

$$\frac{\begin{array}{c}\langle e, R\rangle \rightarrow n \\ \langle e', R\rangle \rightarrow n' \\ n'' = n + n'\end{array}}{\langle e + e', R\rangle \rightarrow n''} \quad \text{E\_PLUS}$$

These expressions read the register state, but do not mutate registers or memory (as you can see just from the form of the judgement).

Defining the Semantics: threads (1/2)

$\boxed{T \xrightarrow{l} T'}$ thread T does l to reach T'

$$\frac{}{t : \langle r = x,\ R\rangle \xrightarrow{t:\mathsf{R}\,x=n} t : \langle \mathtt{skip},\ R \oplus (r \mapsto n)\rangle}\ \text{T\_READ}$$

$$\frac{\langle e, R\rangle \to n}{t : \langle x = e,\ R\rangle \xrightarrow{t:\mathsf{W}\,x=n} t : \langle \mathtt{skip},\ R\rangle}\ \text{T\_WRITE}$$

$$\frac{\langle e, R\rangle \to n}{t : \langle r = e,\ R\rangle \xrightarrow{t:\tau} t : \langle \mathtt{skip},\ R \oplus (r \mapsto n)\rangle}\ \text{T\_COMPUTE}$$

Register writes mutate the thread's register state R; memory reads and writes are exposed as labelled transitions, with read values unconstrained.

Defining the Semantics: threads (2/2)

$\boxed{T \xrightarrow{l} T'}$ thread T does l to reach T'

$$\frac{\langle e, R\rangle \to n' \qquad n = n'}{t : \langle \texttt{if}\ (e \texttt{==} n)\ s_1\ \texttt{else}\ s_2,\ R\rangle \xrightarrow{t:\tau} t : \langle s_1,\ R\rangle}\ \text{T\_COND1}$$

$$\frac{\langle e, R\rangle \to n' \qquad n \neq n'}{t : \langle \texttt{if}\ (e \texttt{==} n)\ s_1\ \texttt{else}\ s_2,\ R\rangle \xrightarrow{t:\tau} t : \langle s_2,\ R\rangle}\ \text{T\_COND2}$$

$$\frac{}{t : \langle \texttt{skip};\ s_2,\ R\rangle \xrightarrow{t:\tau} t : \langle s_2,\ R\rangle}\ \text{T\_SEQ\_SKIP}$$

$$\frac{t : \langle s_1,\ R\rangle \xrightarrow{l} t : \langle s_1',\ R'\rangle}{t : \langle s_1;\ s_2,\ R\rangle \xrightarrow{l} t : \langle s_1';\ s_2,\ R'\rangle}\ \text{T\_SEQ\_CONTEXT}$$

Example thread transitions

$$t_0 : \langle r_1 \mathtt{=} r_0 \mathtt{+} 1,\ R_0 \rangle \xrightarrow{t_0:\tau} t_0 : \langle \mathtt{skip},\ R_0 \oplus (r_1 \mapsto 1) \rangle$$

Example thread transitions

$$t_0 : \langle r_1 = r_0 + 1,\ R_0 \rangle \xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_1 \mapsto 1) \rangle$$

$$\begin{aligned} t_0 : \langle r_0 = 3 ;\ r_1 = r_0,\ R_0 \rangle &\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip} ;\ r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\ &\xrightarrow{t_0:\tau} t_0 : \langle r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\ &\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 3, r_1 \mapsto 3) \rangle \end{aligned}$$

Example thread transitions

$$t_0 : \langle r_1 = r_0 + 1,\ R_0\rangle \xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_1 \mapsto 1)\rangle$$

$$\begin{aligned}
t_0 : \langle r_0 = 3;\ r_1 = r_0,\ R_0\rangle &\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip};\ r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3)\rangle\\
&\xrightarrow{t_0:\tau} t_0 : \langle r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3)\rangle\\
&\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 3, r_1 \mapsto 3)\rangle
\end{aligned}$$

The transitions are those derivable by trees of instantiations of the rules, e.g.

$$\dfrac{\dfrac{\dfrac{}{\langle 3, R_0\rangle \to 3}\ \textsf{E\_INT}}{t_0 : \langle r_0 = 3,\ R_0\rangle \xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 3)\rangle}\ \textsf{T\_COMPUTE}}{t_0 : \langle r_0 = 3;\ r_1 = r_0,\ R_0\rangle \xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip};\ r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3)\rangle}\ \textsf{T\_SEQ\_CONTEXT}$$

where that instance of T_SEQ_CONTEXT has instantiation:

t	$\mapsto$	t_0	R	$\mapsto$	R_0	l	$\mapsto$	$t_0{:}\tau$
s_1	$\mapsto$	$r_0 = 3$	s_1'	$\mapsto$	$\texttt{skip}$			
s_2	$\mapsto$	$r_1 = r_0$	R'	$\mapsto$	$R_0 \oplus (r_0 \mapsto 3)$			

Example thread transitions

$$t_0 : \langle r_1 = r_0 + 1,\ R_0 \rangle \xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_1 \mapsto 1) \rangle$$

$$\begin{aligned} t_0 : \langle r_0 = 3 ;\ r_1 = r_0,\ R_0 \rangle &\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip} ;\ r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\ &\xrightarrow{t_0:\tau} t_0 : \langle r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\ &\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 3, r_1 \mapsto 3) \rangle \end{aligned}$$

$$t_0 : \langle x = 3,\ R_0 \rangle \xrightarrow{t_0:\mathsf{W}\ x=3} t_0 : \langle \texttt{skip},\ R_0 \rangle$$

Example thread transitions

$$t_0 : \langle r_1 = r_0 + 1,\ R_0 \rangle \xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_1 \mapsto 1) \rangle$$

$$\begin{aligned}
t_0 : \langle r_0 = 3;\ r_1 = r_0,\ R_0 \rangle &\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip};\ r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\
&\xrightarrow{t_0:\tau} t_0 : \langle r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\
&\xrightarrow{t_0:\tau} t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 3, r_1 \mapsto 3) \rangle
\end{aligned}$$

$$t_0 : \langle x = 3,\ R_0 \rangle \xrightarrow{t_0:\mathsf{W}\ x=3} t_0 : \langle \texttt{skip},\ R_0 \rangle$$

$$\begin{tikzcd}
 & t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 7) \rangle \\
t_0 : \langle r_0 = x,\ R_0 \rangle \arrow[ur, "t_0:\mathsf{R}\ x=7"] \arrow[dr, "t_0:\mathsf{R}\ x=23"] & \\
 & t_0 : \langle \texttt{skip},\ R_0 \oplus (r_0 \mapsto 23) \rangle
\end{tikzcd}$$

Example thread transitions

$$t_0 : \langle r_1 = r_0 + 1,\ R_0 \rangle \xrightarrow{t_0:\tau} t_0 : \langle \mathtt{skip},\ R_0 \oplus (r_1 \mapsto 1) \rangle$$

$$\begin{aligned}
t_0 : \langle r_0 = 3;\ r_1 = r_0,\ R_0 \rangle &\xrightarrow{t_0:\tau} t_0 : \langle \mathtt{skip};\ r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\
&\xrightarrow{t_0:\tau} t_0 : \langle r_1 = r_0,\ R_0 \oplus (r_0 \mapsto 3) \rangle \\
&\xrightarrow{t_0:\tau} t_0 : \langle \mathtt{skip},\ R_0 \oplus (r_0 \mapsto 3, r_1 \mapsto 3) \rangle
\end{aligned}$$

$$t_0 : \langle x = 3,\ R_0 \rangle \xrightarrow{t_0:\mathsf{W}\,x=3} t_0 : \langle \mathtt{skip},\ R_0 \rangle$$

$$\begin{tikzcd}
 & t_0 : \langle \mathtt{skip},\ R_0 \oplus (r_0 \mapsto 7) \rangle \\
t_0 : \langle r_0 = x,\ R_0 \rangle \arrow[ur, "t_0:\mathsf{R}\,x=7"] \arrow[dr, "t_0:\mathsf{R}\,x=23"'] & \\
 & t_0 : \langle \mathtt{skip},\ R_0 \oplus (r_0 \mapsto 23) \rangle
\end{tikzcd}$$

$$\begin{aligned}
t_0 : \langle x = 3;\ r_0 = x,\ R_0 \rangle &\xrightarrow{t_0:\mathsf{W}\,x=3} t_0 : \langle \mathtt{skip};\ r_0 = x,\ R_0 \rangle \\
&\xrightarrow{t_0:\tau} t_0 : \langle r_0 = x,\ R_0 \rangle \\
&\xrightarrow{t_0:\mathsf{R}\,x=7} t_0 : \langle \mathtt{skip},\ R_0 \oplus (r_0 \mapsto 7) \rangle
\end{aligned}$$

Defining the Semantics: lifting to processes

Remember the process syntax:

$process, \ P ::=$		process
$\mid$	T	thread
$\mid$	$P \,\vert\, P'$	parallel composition

$\boxed{P \xrightarrow{l} P'}$ process P does l to become P'

$$\frac{T \xrightarrow{l} T'}{T \xrightarrow{l} T'} \quad \text{P\_THREAD}$$

$$\frac{P_1 \xrightarrow{l} P_1'}{P_1 \,\vert\, P_2 \xrightarrow{l} P_1' \,\vert\, P_2} \quad \text{P\_PAR\_CONTEXT\_LEFT}$$

$$\frac{P_2 \xrightarrow{l} P_2'}{P_1 \,\vert\, P_2 \xrightarrow{l} P_1 \,\vert\, P_2'} \quad \text{P\_PAR\_CONTEXT\_RIGHT}$$

Free interleaving of the transitions of each thread.

Defining an SC Semantics: whole-system states

An *SC system state* $S = \langle P,\ M \rangle$ is a pair of a process and an SC memory.

$\boxed{S \xrightarrow{l} S'}$ system S does l to become S'

$$\frac{P \xrightarrow{l} P' \qquad M \xrightarrow{l} M'}{\langle P,\ M \rangle \xrightarrow{l} \langle P',\ M' \rangle} \quad \textsc{S\_access}$$

$$\frac{P \xrightarrow{t:\tau} P'}{\langle P,\ M \rangle \xrightarrow{t:\tau} \langle P',\ M \rangle} \quad \textsc{S\_internal}$$

The rules force synchronisation between the process and the memory, constraining the values of the process's read transitions to those the memory permits, and the memory's write transitions to those the process does (threads can also freely do internal transitions).

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:

$$\begin{tikzcd}
 & \langle t_1 : \langle x = 1,\ R_0\rangle \mid t_2 : \langle x = 2,\ R_0\rangle,\ \{x \mapsto 0\}\rangle \arrow[dl, "t_1{:}\mathsf{W}\,x{=}1"'] \arrow[dr, "t_2{:}\mathsf{W}\,x{=}2"] & \\
\langle t_1 : \langle \texttt{skip},\ R_0\rangle \mid t_2 : \langle x = 2,\ R_0\rangle,\ \{x \mapsto 1\}\rangle \arrow[d, "t_2{:}\mathsf{W}\,x{=}2"'] & & \langle t_1 : \langle x = 1,\ R_0\rangle \mid t_2 : \langle \texttt{skip},\ R_0\rangle,\ \{x \mapsto 2\}\rangle \arrow[d, "t_1{:}\mathsf{W}\,x{=}1"] \\
\langle t_1 : \langle \texttt{skip},\ R_0\rangle \mid t_2 : \langle \texttt{skip},\ R_0\rangle,\ \{x \mapsto 2\}\rangle & & \langle t_1 : \langle \texttt{skip},\ R_0\rangle \mid t_2 : \langle \texttt{skip},\ R_0\rangle,\ \{x \mapsto 1\}\rangle
\end{tikzcd}$$

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...

Message Passing Example

In SC, message passing should work as expected:

Thread 1	Thread 2
`data = 1`	
`ready = 1`	`if (ready == 1)`
	`   print data`

In SC, the program should only print nothing or 1, and on bare-metal x86 it does (not ARM/Power). What about Java/C?

Message Passing Example

Thread 1	Thread 2

`data = 1`
`ready = 1`
	`int r1 = data`
	`if (ready == 1)`
	`    print data`

In SC, the program should only print nothing or 1, and on bare-metal x86 it does (not ARM/Power). What about Java/C?

It should be regardless of other reads.

Message Passing Example

Thread 1	Thread 2
`data = 1`	`int r1 = data`
`ready = 1`	`if (ready == 1)`
	`   print data`

In SC, the program should only print nothing or 1, and on bare-metal x86 it does (not ARM/Power). What about Java/C?

But common subexpression elimination (e.g. in `HotSpot`) can rewrite

`print data` $\Longrightarrow$ `print r1`

Message Passing Example

Thread 1	Thread 2

`data = 1`
`ready = 1`
	`int r1 = data`
	`if (ready == 1)`
	`   print r1`

In SC, the program should only print nothing or 1, and on bare-metal x86 it does (not ARM/Power). What about Java/C?

But common subexpression elimination (e.g. in `HotSpot`) can rewrite

`print data` $\Longrightarrow$ `print r1`

So the compiled program can print 0

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,....