# Tests and Testing

# ‘Empirical Science of the Artificial’

Treating these human-made artifacts as objects of empirical science

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

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

# Litmus Testing

<table>
<tr><td colspan="2">Initial state: x=0 and y=0</td></tr>
<tr><td>Thread 0</td><td>Thread 1</td></tr>
<tr><td>$x$ = $1$ ;<br>$r_0$ = $y$</td><td>$y$ = $1$ ;<br>$r_1$ = $x$</td></tr>
<tr><td colspan="2">Allowed? Thread 0's $r_0 = 0 \wedge$ Thread 1's $r_1 = 0$</td></tr>
</table>

# Litmus Testing

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

Step 1: Get the compiler out of the way, writing tests in assembly: `SB.litmus`:

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

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

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

# Litmus Testing

Step 2: Want to run that test

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

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

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

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

# Litmus Testing

Download `litmus`:
`http://diy.inria.fr/sources/litmus.tar.gz`

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

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

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

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

More tests on course web page.

# Litmus Output (1/2)

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

{x=0; y=0;}

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

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

# Litmus Output (2/2)

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

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

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

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

# What's a Test?

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

# What's a Test?

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

In the operational model, is there a trace

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

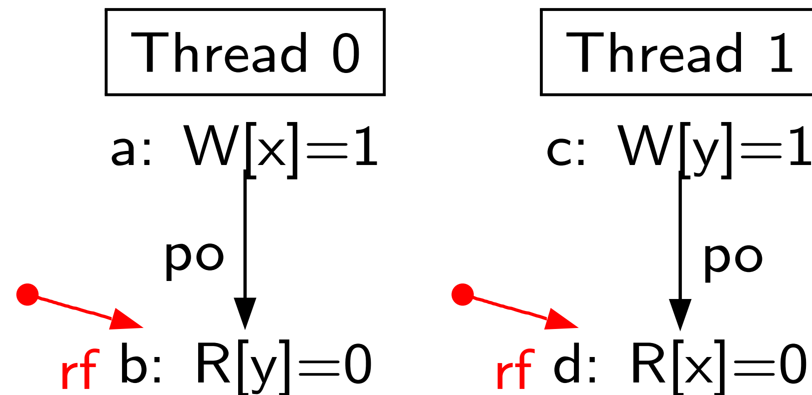
$$\xrightarrow{l_1} \ldots \xrightarrow{ln}$$

$$\langle t_0 : \langle \texttt{skip},\ R_0'\rangle \mid t_1 : \langle \texttt{skip},\ R_1'\rangle,\ M'\rangle$$

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

# Candidate Execution Diagrams

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




Test SB

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

# Coherence

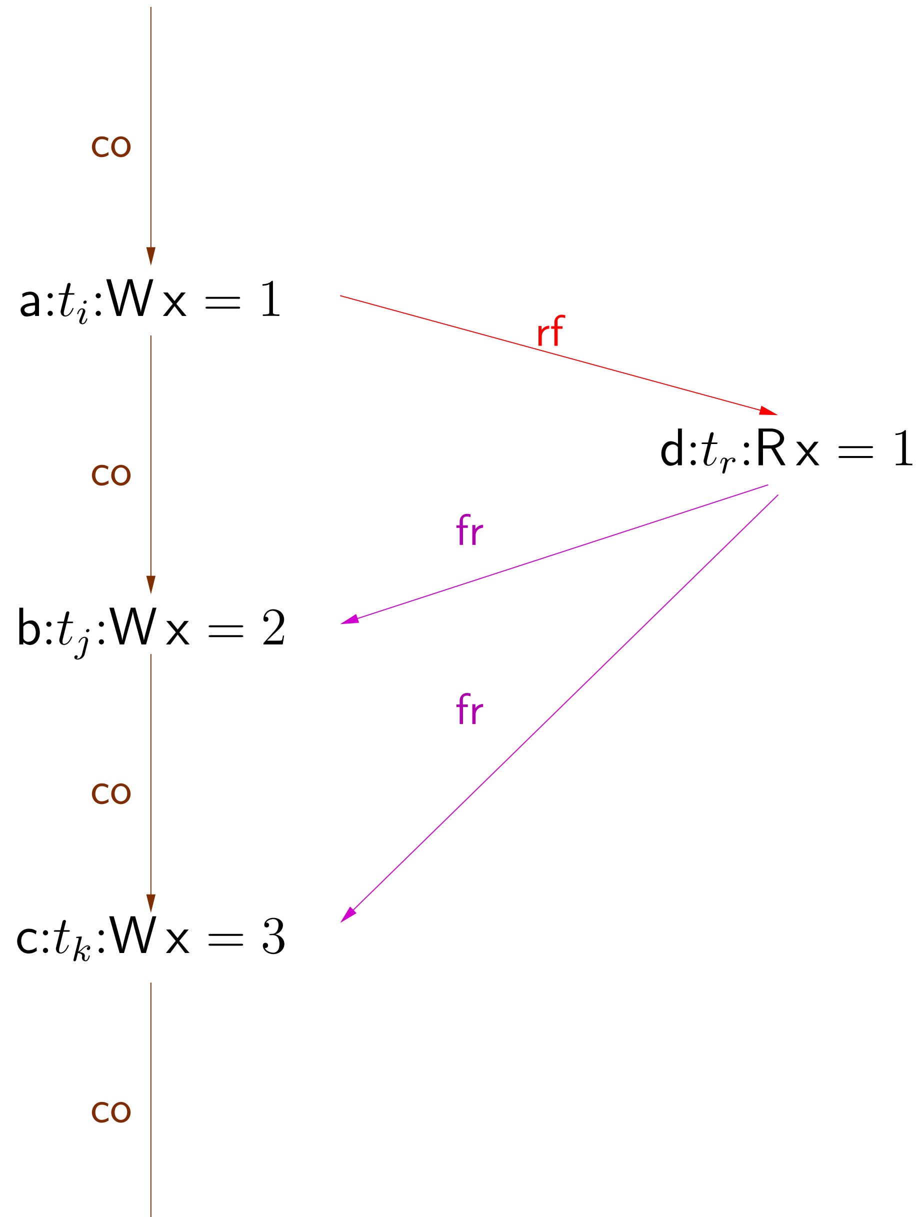
Conventional hardware architectures guarantee *coherence*:

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

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

# From-reads

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



# From-reads

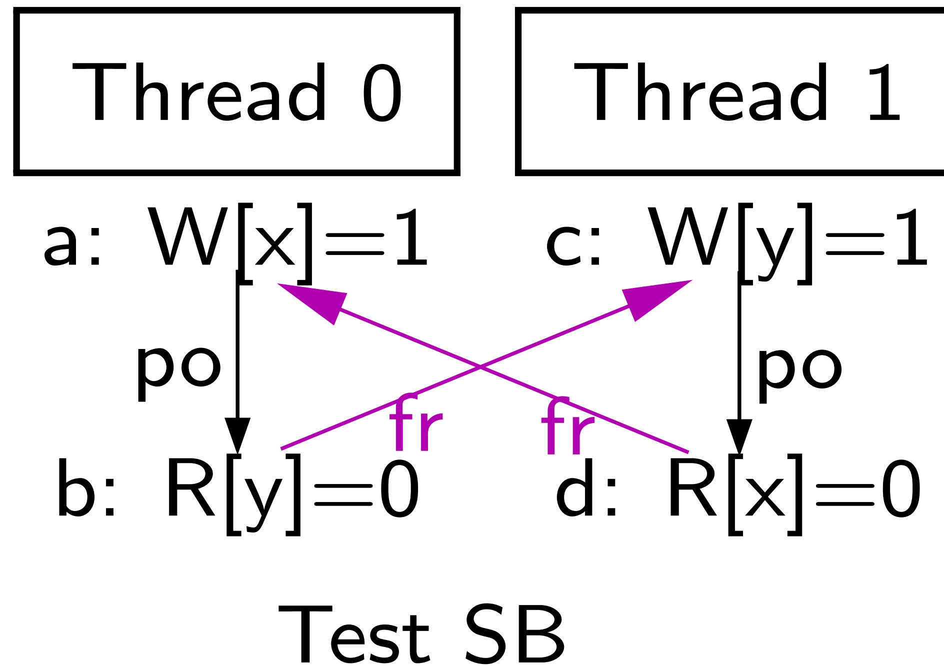
Given that, we can think of a read event as "before" the coherence-successors of the write it reads from.

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

$$
\begin{aligned}
r \xrightarrow{\mathsf{fr}} w \quad \textsf{iff} \quad & (\exists w_0.\ w_0 \xrightarrow{\mathsf{co}} w \quad \wedge \quad w_0 \xrightarrow{\mathsf{rf}} r) \quad \vee \\
& (\neg \exists w_0.\ w_0 \xrightarrow{\mathsf{rf}} r)
\end{aligned}
$$

(co is an irreflexive transitive relation)

# The SB cycle




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

# Candidate Executions, more precisely

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

Say a *candidate pre-execution* $E$ consists of

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

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

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

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

  [note this is intensional: it identifies *which write*, not just the value]

- *coherence* ($co$), an irreflexve transitive relation over $E$ relating only writes that are to the same address; total when restricted to the writes of each address separately

  [intuitively, the hardware coherence order for each address]

# SC, said differently again: pre-executions

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

**Theorem 1 (?)** *$E$ is SC-L iff there exists a trace $\vec{l} \in \mathrm{traces}(M_0)$ of $M_0$ such that the events of $E$ are the labels of $\vec{l}$ (with a choice of unique id for each) and po is the union of the order of $\vec{l}$ restricted to each thread.*

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

# SC, said differently again: “Axiomatically”

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

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

**Theorem 2 (?)** *$E$ is SC-L iff there exists an execution witness $X$ (satisfying the well-formedness conditions of the last-but-one slide) such that $E, X$ is SC-A.*

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

# How to generate good tests?

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

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

  [`diy` tool of Alglave and Maranget, Alglave, Maranget, Sarkar, Sewell, CAV2010 (`http://diy.inria.fr/doc/gen.html`); Shasha and Snir, TOPLAS 1988]
- systematic families of those (see periodic table, later)

Accumulated library of 1000's of litmus tests.

# How to compare test results and models?

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

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

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

# The SC semantics as executable test oracles

Given $P$, either:

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

   (maybe with some partial-order reduction), or

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

(actually for (2a), use an inductive-on-syntax characterisation of the set of all pre-executions of a process)

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

# References


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