Not so practical multicore programming

A simple model for sequential consistency, extended...

Luc Maranget
Luc.Maranget@inria.fr

Breaking news
Exercises, Feb. 20 8h45–11h45
Exam, March 6, 8h45–11h45
Bring your computer for both events.

Part 1.
Axiomatic Sequential Consistency

Shared memory computer

Thread_1
\[W \rightarrow R\]
\[\cdots\]
Thread_n
\[W \rightarrow R\]
\[\cdots\]

Shared Memory
Sequential consistency

**Original definition:** (Leslie Lamport)

[... ] 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.

(And stores take effect immediately).

**Interleaving semantics:** This is “interleaving semantics” as “some sequential order” results from interleaving “the order specified by the program of all individual processors”.

A first, one expect shared multiprocessors to behave that way, which of course they don’t.

---

Events

The effect of “operations executed by the processors” are represented by events. More precisely, as we interleave memory accesses, we define *memory events* $(a) \cdot d(\ell)v$ which consist in:

- Unique label typically $(a), (b), \text{etc.}$
- Direction $d$, that is read (R) or write (W)
- Memory location $\ell$, typically $x, y, \text{etc.}$
- Value $v$, typically 0, 1 etc.
- Originating thread: $T_0, T_1$ (omitted)

The program order $\rightarrow po$ is a linear order amongst the events originating from the same processor.

Relation $\rightarrow po$ represents the sequential execution of events by one processor that follows the *uniprocessor model*: the usual processor execution model, where instruction are executed by following the order given in program.

---

Example of program-order

```c
/* x, t and y are (shared) memory locations, t = { 2, 3, } */
int r1,r2=0 ; // non-shared locations (e.g. registers)
x = 1 ;
for (int k = 0 ; k < 2 ; k++) { r1 = t[k] ; r2 += r1 ; }
y = r2 ;
```

Events and program order:

$(a): W[x]1 \rightarrow po (b): R[t + 0]2 \rightarrow po (c): R[t + 4]3 \rightarrow po (d): W[y]5$

---

A definition of SC

A transcription of L. Lamport’s definition.

**Definition (SC 1)**

An execution is SC when there exists a total order on events $<$, such that:

- **Order** $<$ is compatible with program order:
  
  \[ e_1 \rightarrow po e_2 \Rightarrow e_1 < e_2. \]

- **Reads** read from the closest write upwards:

  \[
  (w, r) \rightarrow \text{Def } \{ (w', r) \mid w = \max(w', \text{loc}(w')) = \text{loc}(r) \land w' < r \}.
  \]
Example of a question on SC

Program:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>(a) $x \leftarrow 1$</td>
<td>(c) $y \leftarrow 2$</td>
</tr>
<tr>
<td>(b) $y \leftarrow 1$</td>
<td>(d) $r_0 \leftarrow x$</td>
</tr>
</tbody>
</table>

Observed? $y=2$, $r_0=0$

How do we know? Let us enumerate all interleavings and observe if $b < c$ then $y=2$, if $a < d$ then $r_0=0$.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, b, c, d$</td>
<td>$y=2$, $r_0=1$</td>
</tr>
<tr>
<td>$a, c, b, d$</td>
<td>$y=1$, $r_0=1$</td>
</tr>
<tr>
<td>$a, c, d, b$</td>
<td>$y=1$, $r_0=1$</td>
</tr>
<tr>
<td>$c, d, a, b$</td>
<td>$y=1$, $r_0=0$</td>
</tr>
<tr>
<td>$c, a, b, d$</td>
<td>$y=1$, $r_0=1$</td>
</tr>
<tr>
<td>$c, a, d, b$</td>
<td>$y=1$, $r_0=1$</td>
</tr>
</tbody>
</table>

Let us be a bit more clever

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>(a) $x \leftarrow 1$</td>
<td>(c) $y \leftarrow 2$</td>
</tr>
<tr>
<td>(b) $y \leftarrow 1$</td>
<td>(d) $r_0 \leftarrow x$</td>
</tr>
</tbody>
</table>

Observed? $y=2$, $r_0=0$

Collecting constraints on the scheduling order $<$:

- We respect program order, thus $a < b$, $c < d$.
- We observe $r_0=0$, thus $d < a$.
- We observe $y=2$, thus $b < c$.

Hence we have a cycle in $<$, which prevents it from being an order!

$$a < b < c < d < a \cdots$$

Conclusion: No SC execution would ever yield the output "$y=2$, $r_0=0$;".

Systematic approach

At the moment, an “execution” (candidate) consists in assuming some events and a program order relation.

We assume two additional relations:

- **Read-from** ($\rightarrow^r$): Relates write events to read events that read the stored value (initial writes left implicit in diagrams).

  $$\forall r, \exists! w, w \rightarrow^r r$$

  *(Notice: $w$ and $r$ have identical location and value.)*

- **Coherence** ($\rightarrow^c$): Relates write events to the same location.

  For any location $\ell$, the restriction of $\rightarrow^c$ to write events to location $\ell$ ($W_\ell$) is a total order.

Coherence as a characteristics of shared memory

The very existence of $\rightarrow^c$ is implied by the existence of a shared, coherent, memory — Given location $x$, there is exactly one memory cell whose location is $x$.

$$W_0 \rightarrow^c W_1 \rightarrow^c x = 2 \rightarrow^c W_3 \rightarrow^c \cdots$$

Of course, in reality, there caches, buffers etc. But the system will behave “as if”.
Example of \( rf \leftrightarrow \) LB

<table>
<thead>
<tr>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( r_0 \leftarrow x )</td>
<td>(c) ( r_1 \leftarrow y )</td>
</tr>
<tr>
<td>(b) ( y \leftarrow 1 )</td>
<td>(d) ( x \leftarrow 1 )</td>
</tr>
</tbody>
</table>

Observe: \( r_0; r_1; \)

There are 4 possible \( rf \leftrightarrow \) relations (initial value is 0).

\[
\begin{align*}
r_0=1; & \quad r_1=1; \\
r_0=0; & \quad r_1=0;
\end{align*}
\]

\[
\begin{align*}
a: Rx=1 & \quad c: Ry=1 \\
b: Wy=1 & \quad d: Wx=1
\end{align*}
\]

\[
\begin{align*}
a: Rx=0 & \quad c: Ry=0 \\
b: Wy=1 & \quad d: Wx=1
\end{align*}
\]

Notice: In this simple case of two stores, the value finally observed in locations determines \( co \leftrightarrow \) for them.

Example of \( co \leftrightarrow \)

<table>
<thead>
<tr>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( x \leftarrow 2 )</td>
<td>(c) ( y \leftarrow 2 )</td>
</tr>
<tr>
<td>(b) ( y \leftarrow 1 )</td>
<td>(d) ( x \leftarrow 1 )</td>
</tr>
</tbody>
</table>

Observe? \( x=1; \quad y=1; \)

\[
\begin{align*}
x=1; & \quad y=2; \\
x=2; & \quad y=1;
\end{align*}
\]

One more relation: \( fr \leftrightarrow \)

The new relation \( fr \leftrightarrow \) (from read) relates reads to “younger writes” (younger w.r.t. \( co \leftrightarrow \)).

\[
r \xrightarrow{fr} w \quad \text{Def} = w' \xrightarrow{fr} r \wedge w' \xrightarrow{co} w
\]

This amounts to place a read into the coherence order of its location:

Given

\[
\begin{align*}
w_0 \xrightarrow{co} & \quad w_1 \xrightarrow{co} \ldots \xrightarrow{co} w_n \\
r \xrightarrow{fr} & \\
\end{align*}
\]

We have

\[
\begin{align*}
w_0 \xrightarrow{rf} & \quad w_1 \xrightarrow{rf} \ldots \xrightarrow{rf} w_n
\end{align*}
\]

(Or: \( fr \leftrightarrow \) Def = \( (fr \leftrightarrow)^{-1} ; \) \( co \leftrightarrow \))

Playing with \( fr \leftrightarrow \)

Particular, easy case: a read from the initial state is in \( fr \leftrightarrow \) with writes by the program.

<table>
<thead>
<tr>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( x \leftarrow 1 )</td>
<td>(c) ( r_0 \leftarrow y )</td>
</tr>
<tr>
<td>(b) ( r_0 \leftarrow y )</td>
<td>(d) ( r_1 \leftarrow x )</td>
</tr>
</tbody>
</table>

Observe? \( r_0=0; \quad r_1=0; \)

\[
\begin{align*}
a: Wx=1 & \quad c: Ry=1 \\
b: Wy=1 & \quad d: Rx=0
\end{align*}
\]

<table>
<thead>
<tr>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( x \leftarrow 1 )</td>
<td>(c) ( y \leftarrow 1 )</td>
</tr>
<tr>
<td>(b) ( r_0 \leftarrow y )</td>
<td>(d) ( r_1 \leftarrow x )</td>
</tr>
</tbody>
</table>

Observe? \( r_0=0; \quad r_1=0; \)

\[
\begin{align*}
a: Wx=1 & \quad c: Wy=1 \\
b: Ry=0 & \quad d: Rx=0
\end{align*}
\]
Second definition of SC

Definition (SC 2)
An execution is SC when:

\[
\text{Acyclic} \left( \mathcal{R} \to \cup \mathcal{C} \to \cup \mathcal{F} \to \cup \mathcal{P} \to \right)
\]

And of course:

Theorem
The two definitions of SC are equivalent.

SC 2 \implies SC 1
Since \( \mathcal{R} \to \cup \mathcal{C} \to \cup \mathcal{F} \to \cup \mathcal{P} \to \) is a partial order, there exists a total order
\( < \) that “extends” it (no question on mathematical foundations,...).

From \( < \) define \( \mathcal{R} < \):

\[
\mathcal{R} < \overset{\text{Def}}{=} \left\{ (w, r) : w = \max(<, \loc(w')) = \loc(r) \land w' < r \right\}
\]

and show \( \mathcal{R} = \mathcal{R} < \).

1. Let \( w_0 \mathcal{R} r \) and let \( w \in W_r, w \neq w_0 \) then \( \mathcal{C} \) total order on \( W_r \):
   - Either \( w \mathcal{C} w_0 \) and \( w < w_0 < r \).
   - Or, \( w_0 \mathcal{C} w \), and then \( \mathcal{F} r \to w \), and \( r < w \).
   - Finally \( w_0 \mathcal{R} r \).

2. Let \( w \mathcal{F} r \) (i.e. \( w \in W_r, w \neq w_0 \)), then
   - Either \( w \mathcal{C} w_0 \) and \( \mathcal{C} \subseteq < \) \( w \mathcal{F} r \).
   - Or \( w_0 \mathcal{C} w \), thus \( \mathcal{F} r \to w \), and \( \mathcal{F} r \subseteq < \) \( w \mathcal{F} r \).

Simulating SC
Which model, SC 1 or SC 2 is the most convenient/efficient?

SC 1 Enumerate interleavings.
SC 2 Enumerate axiomatic execution candidates \( \{i.e. \mathcal{P}, \mathcal{R}, \mathcal{C}\} \);
check the acyclicity of \( \mathcal{R} \cup \mathcal{C} \cup \mathcal{F} \cup \mathcal{P} \).

Answer: we view SC 2 as being more convenient, since the generated
objects usually are smaller.
Introducing herd, a memory model simulator

A model sc.cat:

```cat
% cat sc.cat
include "cos.cat" #define co (and fr)
let com = rf | co | fr #communication
acyclic po | com as hb #validity condition
```

Running R on SC (demo in demo/02):

Test R Allowed
States 3
1:EAX=0; y=1;
1:EAX=1; y=1;
1:EAX=1; y=2;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (y=2 \ 1:EAX=0)
Observation R Never 0 3

Notice: Outcome 1:EAX=0; y=2; is forbidden by SC.

Herd structure

- Generate all candidate executions, i.e. all possible \( \text{po} \rightarrow \text{rf} \) and \( \text{co} \rightarrow \) (\( \text{fr} \) deduced):

  a: Wx=1  c: Wy=2  
  b: Wy=1  d: Rx=1  
  Ok

  a: Wx=1  c: Wy=2  
  b: Wy=1  d: Rx=0  
  No

- Apply model checks to each candidate execution.

Violations of SC

A cycle of \( \text{po} \rightarrow \text{rf} \), \( \text{co} \rightarrow \) describes a violation of SC.

From such a cycle, one may easily generate programs that potentially violate SC, and run them on actual machines.

However, the cycle does not describe:
- How many threads are involved.
- How many memory locations are involved.

We now aim at:
- Extract a subset of significant cycles.
- Generate one program out of one cycle.

Part 2.

Studying Non-Sequentially Consistently Executions.
Simplifying cycles: $\rightarrow^\text{po}$ and $\rightarrow^\text{com}$ steps alternate

A cycle in $\rightarrow^\text{com} \cup \rightarrow^\text{po}$ is a cycle in $(\rightarrow^\text{po} +, \rightarrow^\text{com} +)$ (group $\rightarrow^\text{po}$ and $\rightarrow^\text{com}$ steps together). Then:

- $\rightarrow^\text{po}$ is transitive $\rightarrow^\text{po} + \subseteq \rightarrow^\text{po}$.
- $\rightarrow^\text{com}$ is the union of the five following relations:
  - $\rightarrow^\text{com} = \rightarrow^\text{rf} \cup \rightarrow^\text{co} \cup \rightarrow^\text{fr}$
  - $\rightarrow^\text{com} = \rightarrow^\text{fr} \cup \rightarrow^\text{co} \cup \rightarrow^\text{rf}$

Because $(\rightarrow^\text{co}; \rightarrow^\text{co}) \subseteq \rightarrow^\text{co}$, $(\rightarrow^\text{fr}; \rightarrow^\text{co}) \subseteq \rightarrow^\text{fr}$, and $(\rightarrow^\text{rf}; \rightarrow^\text{fr}) \subseteq \rightarrow^\text{co}$.

**Conclusion:** Any cyclic $\rightarrow^\text{com} \cup \rightarrow^\text{po}$ includes a cycle in $(\rightarrow^\text{po}; \rightarrow^\text{com})$ — i.e. that alternates $\rightarrow^\text{po}$ steps and $\rightarrow^\text{com}$ steps.

Simplifying cycles – Threads

Assume a cycle with two $\rightarrow^\text{po}$ steps on the same thread:

$$e_1 \rightarrow^\text{po} e_2 (\rightarrow^\text{com}; \rightarrow^\text{po})^* ; \rightarrow^\text{com} e_3 \rightarrow^\text{po} e_4 (\rightarrow^\text{com}; \rightarrow^\text{po})^* ; \rightarrow^\text{com} e_1$$

Assuming for instance, $e_1 \rightarrow^\text{po} e_3$ then we have a "simpler" cycle:

$$e_1 \rightarrow^\text{po} e_2 \rightarrow^\text{po} e_3 (\rightarrow^\text{com}; \rightarrow^\text{po})^* ; \rightarrow^\text{com} e_3 = e_1$$

(Conclude with $\rightarrow^\text{po}$ being transitive)

If $e_1 = e_3$, we also have a simpler cycle:

$$e_1 \rightarrow^\text{po} e_2 (\rightarrow^\text{com}; \rightarrow^\text{po})^* ; \rightarrow^\text{com} e_3 = e_1$$

**Conclusion:** Pass through each thread only once.

Simplifying cycles: all $\rightarrow^\text{com}$ steps are external

Given a cycle, we consider that all $\rightarrow^\text{com}$ and $\rightarrow^\text{po}$ steps are external, (i.e. source and target events are from pairwise distinct thread).

Given $e_1 \rightarrow^\text{com} e_2$, s.t. $e_1$ and $e_2$ are from the same thread:

- Either $e_1 \rightarrow^\text{po} e_2$ and we consider this $\rightarrow^\text{po}$ step in the cycle, in place of the $\rightarrow^\text{com}$ step (further merging $\rightarrow^\text{po}$ steps to get a smaller cycle).
- Or $e_2 \rightarrow^\text{po} e_1$, then we have a very simple cycle $e_2 \rightarrow^\text{po} e_1 \rightarrow^\text{com} e_2$.

Such cycles are "violations of coherence" (more on them later).

**Case** $e_1 = e_2$ is impossible ($\rightarrow^\text{com}$ is acyclic, see later)

**Notice:** A similar reasoning applies to individual $\rightarrow^\text{com}$ steps in composite $\rightarrow^\text{com}$.

**Conclusion:** Pass through each thread only once.

Test from cycles — Threads

Consider a test execution on two threads:

**Cycle:** $R \rightarrow^\text{po} W \rightarrow^\text{rf} R \rightarrow^\text{po} W \rightarrow^\text{rf} R \rightarrow^\text{po} W \rightarrow^\text{rf}$

The test execution features a smaller cycle

- $a$: Rx=1
- $b$: Wy=1
- $c$: Rz=1
- $d$: Wa=1
- $e$: Ry=1
- $f$: Wz=1
- $g$: Ra=1
- $h$: Wx=1

Generally: one passage per thread
Simplifying cycles, a lemma

**Lemma (Identical locations)**

Let $e_1$, $e_2$ two different events with the same location.

- either $e_1 \overset{\text{com}}{\rightarrow} e_2$.
- or $e_2 \overset{\text{com}}{\rightarrow} e_1$.
- or $w \overset{\text{rf}}{\rightarrow} e_1$ and $w \overset{\text{rf}}{\rightarrow} e_2$.

Case analysis:

- $w_1, w_2$, then either $w_1 \overset{\text{co}}{\rightarrow} w_2$ or $w_2 \overset{\text{co}}{\rightarrow} w_1$ (total order).
- $r_1, r_2$, let $w_1 \overset{\text{rf}}{\rightarrow} r_1$ and $w_2 \overset{\text{rf}}{\rightarrow} r_2$. Then, either $w_1 = w_2$ and we are in case 3; or (for instance) $w_1 \overset{\text{co}}{\rightarrow} w_2$ and we have $r_1 \overset{\text{rf}}{\rightarrow} w_2 \overset{\text{rf}}{\rightarrow} r_2$.
- $r_1, w_2$, let $w_1 \overset{\text{rf}}{\rightarrow} r_1$. Then, either $w_1 = w_2$ and $w_2 \overset{\text{rf}}{\rightarrow} r_1$; or $w_2 \overset{\text{co}}{\rightarrow} w_2$ and $r_1 \overset{\text{rf}}{\rightarrow} w_2$; or $w_2 \overset{\text{co}}{\rightarrow} w_1$ and $w_2 \overset{\text{co}}{\rightarrow} r_1$.

**Corollary:** $\overset{\text{com}}{\rightarrow}$ is acyclic.

Test from cycles — Locations

Cycle: $R \overset{\text{po}}{\rightarrow} W \overset{\text{rf}}{\rightarrow} R \overset{\text{po}}{\rightarrow} W \overset{\text{rf}}{\rightarrow} R \overset{\text{po}}{\rightarrow} W \overset{\text{rf}}{\rightarrow}$

- One interpretation (four locations):
  - a: $Rx=1$
  - c: $Ry=1$
  - e: $Rz=1$
  - g: $Ra=1$
  - b: $Wy=1$
  - d: $Wz=1$
  - f: $Wa=1$
  - h: $Wx=1$

- Another interpretation (two locations):
  - a: $Rx=2$
  - c: $Ry=1$
  - e: $Rx=1$
  - g: $Ry=2$
  - b: $Wy=1$
  - d: $Wx=1$
  - f: $Wy=2$
  - h: $Wx=2$

The second interpretation is not “minimal”

Reminding the interpretation with two locations:

Let us choose: $Wx_1 \overset{\text{co}}{\rightarrow} Wx_2$:

- a: $Rx=2$
- c: $Ry=1$
- e: $Rx=1$
- g: $Ry=2$
- b: $Wy=1$
- d: $Wx=1$
- f: $Wy=2$
- h: $Wx=2$

But, coherence $\overset{\text{co}}{\rightarrow}$ totally orders write events to a given location.

We have a smaller cycle: $d \overset{\text{co}}{\rightarrow} h \overset{\text{rf}}{\rightarrow} a \overset{\text{po}}{\rightarrow} b \overset{\text{rf}}{\rightarrow} c \overset{\text{po}}{\rightarrow} d$.

Choosing $Wx_2 \overset{\text{co}}{\rightarrow} Wx_1$ would yield another smaller cycle.

Generally: do not repeat locations in cycles.

Simplifying cycles — Identical Locations

We show that we can restrict cycles to those where events with identical locations are related by $\overset{\text{com}}{\rightarrow}$ steps.

Assume a cycle including $e_1$ and $e_2$ with the same location.

- If $e_1$ and $e_2$ are from different threads. By hypothesis, $e_1$ and $e_2$ are related by complex steps (i.e. at least one $\overset{\text{po}}{\rightarrow}$ and one $\overset{\text{co}}{\rightarrow}$) in both directions. By the identical locations lemma:
  - Either, $e_1 \overset{\text{com}}{\rightarrow} e_2$ or $e_2 \overset{\text{com}}{\rightarrow} e_1$, and we have a simpler cycle.
  - or, $w \overset{\text{rf}}{\rightarrow} e_1$ and $w \overset{\text{rf}}{\rightarrow} e_2$, — see next page!

- If $e_1$ and $e_2$ are from the same thread, i.e. for instance $e_1 \overset{\text{po}}{\rightarrow} e_2$, while $e_2$ relates to $e_1$ by complex steps:
  - either $e_1 \overset{\text{com}}{\rightarrow} e_2$ and we replace the $\overset{\text{po}}{\rightarrow}$ step in cycle, yielding a simpler cycle (one $\overset{\text{po}}{\rightarrow}$ step less)
  - or $e_2 \overset{\text{com}}{\rightarrow} e_1$ and we have a very simple cycle $e_1 \overset{\text{po}}{\rightarrow} e_2 \overset{\text{com}}{\rightarrow} e_1$.
  - Or $w \overset{\text{rf}}{\rightarrow} e_1$ and $w \overset{\text{rf}}{\rightarrow} e_2$, we short-circuit the cycle — as the cycle must be $\cdots w \overset{\text{rf}}{\rightarrow} e_1 \overset{\text{po}}{\rightarrow} e_2 \overset{\text{co}}{\rightarrow} \cdots$, which we reduce into $\cdots w \overset{\text{rf}}{\rightarrow} \cdots e_2 \cdots$. 

...
Next page

So let us assume a cycle that includes \( r_1 \) and \( r_2 \), related in both directions by complex steps and such that \( w \xrightarrow{rf} r_1 \) and \( w \xrightarrow{rf} r_2 \). We consider:

- If \( w \xrightarrow{rf} r_1 \) is in cycle, then there is an obvious short-circuit: replace \( \xrightarrow{rf} \) followed by the complex steps from \( r_1 \) to \( r_2 \) by a single \( w \xrightarrow{rf} r_2 \) step.
- If \( w \xrightarrow{rf} r_2 \) is in cycle, symmetrical case.
- Otherwise, it must be that both \( r_1 \) and \( r_2 \) are the target of \( \xrightarrow{po} \) steps and the source of \( \xrightarrow{fr} \) steps: let \( w_1 \) and \( w_2 \) be the targets of those steps.

Then, in all possible three situations: \( w_1 = w_2 \), \( w_1 \xrightarrow{co} w_2 \) and \( w_2 \xrightarrow{co} w_1 \) we construct a simpler cycle that does not contain \( r_1 \) or \( r_2 \).

... Simplifying cycles — Conclusion

In a non SC execution we find:

- A violation of coherence, that is a cycle \( e_1 \xrightarrow{po} e_2 \xrightarrow{com} e_1 \).
- Or a critical cycle that is:
  - The cycle alternates \( \xrightarrow{po} \) steps and external \( \xrightarrow{com} \) steps.
  - The cycle passes through a given thread at most once.
  - All \( \xrightarrow{com} \) steps have pairwise different locations.
  - The source and target of one given \( \xrightarrow{po} \) steps have different locations.

Notice: By the last condition, such cycles have four steps or more and pass through two threads or more.

For a more formal presentation see D. Shasha and M. Snir Toplas 88 article, which introduced critical cycles.

Violations of coherence

A violation of coherence is a cycle \( e_1 \xrightarrow{po} e_2 \xrightarrow{com} e_1 \).

Given the definition of \( \xrightarrow{com} \), there are five such cycles, which can occur as the following executions: \( \xrightarrow{co} \) contradicts \( \xrightarrow{co} \), \( \xrightarrow{rf} \), \( \xrightarrow{fr} \), \( \xrightarrow{co} \); \( \xrightarrow{rf} \), \( \xrightarrow{fr} \).

Application, all possible SC violations on two threads

Simply list all (critical) cycles for 2 threads, we have six cycles:

- 2+2W \( \xrightarrow{po} \xrightarrow{co} \xrightarrow{po} \xrightarrow{co} \)
- LB \( \xrightarrow{po} \xrightarrow{rf} \xrightarrow{po} \xrightarrow{rf} \)
- MP \( \xrightarrow{po} \xrightarrow{fr} \xrightarrow{po} \xrightarrow{fr} \)
- R \( \xrightarrow{po} \xrightarrow{co} \xrightarrow{po} \xrightarrow{fr} \)
- S \( \xrightarrow{po} \xrightarrow{rf} \xrightarrow{po} \xrightarrow{co} \)
- SB \( \xrightarrow{po} \xrightarrow{fr} \xrightarrow{po} \xrightarrow{fr} \)

Any non-SC execution on two threads includes one of the above six cycles.

Notice: up to coherence violations (previous slide).
Generating two-threads SC violations

The tool diy generates cycles (and tests) from a vocabulary of “edges”. It can be configured for the two threads case as follows:

-arch X86
-safe Pod**,Rfe,Fre,Wse
-nprocs 2
-size 4
-num false

Demo in demo/diy.

% diy7 -conf 2.conf
Generator produced 6 tests
% ls
2+2W.litmus 2.conf @all LB.litmus
MP.litmus R.litmus SB.litmus S.litmus
% diy7 -conf 4.conf
Generator produced 68 tests...

Three violations of SC

<table>
<thead>
<tr>
<th>2+2W</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>(a) x ← 2</td>
<td>(c) y ← 2</td>
<td></td>
</tr>
<tr>
<td>(b) y ← 1</td>
<td>(d) x ← 1</td>
<td></td>
</tr>
</tbody>
</table>

Observed? x=1; y=1;

<table>
<thead>
<tr>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>(a) r0 ← x</td>
</tr>
<tr>
<td>(b) y ← 1</td>
</tr>
<tr>
<td>(c) r1 ← y</td>
</tr>
<tr>
<td>(d) x ← 1</td>
</tr>
</tbody>
</table>

Observed: r0=1; r1=1;

<table>
<thead>
<tr>
<th>MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>(a) x ← 1</td>
</tr>
<tr>
<td>(b) y ← 1</td>
</tr>
<tr>
<td>(c) r0 ← y</td>
</tr>
<tr>
<td>(d) r1 ← x</td>
</tr>
</tbody>
</table>

Observed? r0=1; r1=0

Three more violations of SC

<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>(a) x ← 1</td>
</tr>
<tr>
<td>(b) y ← 1</td>
</tr>
</tbody>
</table>

Observed? y=2; r0=0

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>(a) x ← 2</td>
</tr>
<tr>
<td>(b) y ← 1</td>
</tr>
</tbody>
</table>

Observed? x=2; r0=1

<table>
<thead>
<tr>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>(a) x ← 1</td>
</tr>
<tr>
<td>(b) r0 ← y</td>
</tr>
</tbody>
</table>

Observed? r0=0; r1=0

Application

We assume the following on modern shared memory architectures:
- No valid execution includes a violation of coherence.
- No valid execution includes a cycle whose steps include the adequate fence instruction between source and target instructions.
- The full memory barrier is always adequate.

To guarantee SC:
- Find all possible critical cycles of all possible executions on the architecture.
- Insert a fence in every step of those.

Simplification:
Insert fences between all pairs of racy accesses with different locations (notice that always includes a write).

Optimisation
Forbid specific (critical) cycles by specific means (lightweight barriers, dependencies).
A semi realistic example

```c
for (int k = N ; k >= 0 ; k--) {
    a: x = k ;
    b: go = 1 ;
    c: while (go == 1) ;
}
```

To insert fence, consider separating accesses to `go` and `x`.

```c
for (int k = N ; k >= 0 ; k--) {
    a: x = k ;
    sync();
    b: go = 1 ;
    c: while (go == 1) ;
    sync();
}
```

The resulting static $\rightarrow_{\text{po}}$ relation is as follows.

Analysis based upon Sekar et al. Power model (PLDI'11). Test MP

\[ a \xrightarrow{\text{lw}} b, d \xrightarrow{\text{ct}} e \]

X86: no fence needed.

A semi realistic example, more precise fencing

```c
for (int k = N ; k >= 0 ; k--) {
    int sum = 0 ;
    for (int k = N ; k >= 0 ; k--) {
        d: while (go == 0) ;
        e: sum += x ;
        f: go = 0 ;
    }
}
```

```c
int sum = 0 ;
for (int k = N ; k >= 0 ; k--) {
    d: while (go == 0) ;
    e: sum += x ;
    f: go = 0 ;
}
```

Analysis based upon Sekar et al. Power model (PLDI'11). Test R

\[ a \xrightarrow{\text{sync}} b, d \xrightarrow{\text{sync}} e \]

X86: \( f \xrightarrow{\text{mfence}} e \)
Cycle 3

Analysis based upon Sekar et al. Power model (PLDI'11). Test SB

\[ a \xrightarrow{\text{syn}} c, f \xrightarrow{\text{syn}} e \]

X86: \( a \xrightarrow{\text{mfence}} c, f \xrightarrow{\text{mfence}} e \)

Cycle 4

Analysis based upon Sekar et al. Power model (PLDI'11). Test MP

\[ b \xrightarrow{\text{blwync}} a, e \xrightarrow{\text{ctrlisync}} d \]

X86: no fence needed.

Cycle 5

Analysis based upon Sekar et al. Power model (PLDI'11). Test S

\[ b \xrightarrow{\text{blwync}} a, e \xrightarrow{\text{ctrl}} f \]

X86: no fence needed.

Cycle 6

Analysis based upon Sekar et al. Power model (PLDI'11). Test LB

\[ c \xrightarrow{\text{ctrl}} a, e \xrightarrow{\text{ctrl}} f \]

X86: no fence needed.
**Sufficient fencing, X86**

\[
\begin{align*}
&\text{for (int } k = N; k >= 0; k--) \{ \\
&\quad a: x = k; \\
&\quad \text{mfence}(); \\
&\quad b: \text{go} = 1; \quad \text{\textbf{if}} \; \text{go} == 1; \\
&\quad c: \text{while (go == 1)}; \\
&\} \\
\end{align*}
\]

**Notice:** Inserting full memory fence between racy writes gives the same result.

**Inline assembler for fences and ctrlisync**

```c
inline static void sync() {
    asm __volatile__ ("sync" :: : "memory");
}

inline static void lwsync() {
    asm __volatile__ ("lwsync" :: : "memory");
}

inline static void ctrlisync(int t) {
    asm __volatile__ ("cmpwi %[t],0
        beq 0f
        isync
        ::[t] :r(t) :memory
    ");
}
```

**Notice:** Inserting full memory fence between racy accesses is much more simple.

**Sufficient fencing, Power**

\[
\begin{align*}
&\text{for (int } k = N; k >= 0; k--) \{ \\
&\quad a: x = k; \\
&\quad \text{sync}(); \\
&\quad b: \text{go} = 1; \\
&\quad c: \text{while (go == 1)}; \\
&\quad \text{lwsync}(); \\
&\} \\
\end{align*}
\]

**Part 3.**

**Axiomatic TSO**
The write buffer explains how "reads can pass over writes".
Restoring SC with \textit{mfence}

Replace “relaxed” (not in \textit{hib}) $\text{WR}(\text{po} \rightarrow c)$ by $\text{mfence} \rightarrow (\text{in \textit{hib}})$.

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
& $T_0$ & $T_1$ \\
\hline
$R+p+mfence$ & & \\
\hline
(a) $x \leftarrow 1$ & (c) $y \leftarrow 2$ & No \\
(b) $y \leftarrow 1$ & $\text{mfence}$ & \\
(c) $r_0 \leftarrow x$ & (d) $r_1 \leftarrow x$ & \\
\hline
\end{tabular}
\end{table}

SB+mfences

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
& $T_0$ & $T_1$ \\
\hline
SB+rfi-pos & & \\
\hline
(a) $x \leftarrow 1$ & (c) $y \leftarrow 1$ & No \\
(b) $r_0 \leftarrow y$ & (d) $r_1 \leftarrow x$ & \\
\hline
\end{tabular}
\end{table}

Observation of \textbf{SB+rfi-pos}

Demo in demo/TSO2.

- Create test from cycle:
  \begin{verbatim}
  \% diyone7 -norm -arch X86 Rfi PodRR Fre Rfi PodRR Fre
  \% is
  SB+rfi-pos.litmus
  \end{verbatim}

- Run test:
  \begin{verbatim}
  \% litmus7 -mach x86.cfg src/SB+rfi-pos.litmus
  \end{verbatim}

Our TSO 1 model is wrong!

Consider:

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
& $T_0$ & $T_1$ \\
\hline
SB+rfi-pos & & \\
\hline
(a) $x \leftarrow 1$ & (d) $y \leftarrow 1$ & \\
(b) $r_0 \leftarrow x$ & (e) $r_2 \leftarrow y$ & \\
(c) $r_1 \leftarrow y$ & (f) $r_3 \leftarrow x$ & \\
\hline
\end{tabular}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Diagram of TSO 1 model with mfence.
\label{fig:tso1}
\end{figure}

According to model? No. As we have the \textit{hib} cycle:

$\text{RF} \rightarrow \text{PO} \rightarrow \text{RR} \rightarrow \text{RF} \rightarrow \text{PO} \rightarrow \text{RR}$

According to experiments? Ok. Hence TSO 1 is invalidated by hardware.

The effect originates from “\textit{store forwarding}”: A thread can read its own writes from its store buffer, \textit{i.e.} before they reach memory.

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
& $T_0$ & $T_1$ \\
\hline
SB+rfi-pos & & \\
\hline
(a) $x \leftarrow 1$ & (d) $y \leftarrow 1$ & \\
(b) $r_0 \leftarrow x$ & (e) $r_1 \leftarrow y$ & \\
(c) $r_0 \leftarrow x$ & (f) $r_1 \leftarrow y$ & \\
\hline
\end{tabular}
\end{table}

Corrected model: TSO 2

Internal $\rightarrow \text{RF} \rightarrow \text{RF}$ does not create order, external $\rightarrow \text{RF}$ does:

\begin{verbatim}
let com-hb = rfe | fr | co #rfi not in hb
acyclic ppo | com-hb | mfence
\end{verbatim}

The new hb is no longer cyclic:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Diagram of TSO 2 model.
\label{fig:tso2}
\end{figure}

(Also consider that $a \rightarrow \text{WR} \rightarrow c$ and $d \rightarrow \text{WR} \rightarrow f$ are non-global.)
This is not over yet...

Our TSO 2 model:

\[
\begin{align*}
\text{let } \text{ppo} &= (R*M \mid W*W) \quad \# (W*R) & \text{po omitted} \\
\text{let } \text{com-hb} &= rfe \mid fr \mid co \quad \# \text{rfi omitted} \\
\text{acyclic (ppo \mid com-hb \mid mfence) as } \text{hb}
\end{align*}
\]

Allows two violations of coherence:

- CoRW1
  - a: Rx=1
  - b: Wx=1

- CoWR
  - a: Wx=1
  - b: Rx=0

Although TSO2 is not invalidated by hardware. Those “surprising” behaviours must be rejected by our TSO model.

A new check: uniproc

We add a specific uniproc check to rule out coherence violations:

\[
\text{Irreflexive } (\text{po-loc} \rightarrow; \text{com})
\]

Where \(\text{po-loc}\) is \(\rightarrow\) between accesses to the same memory location.

\[
\begin{align*}
\text{let } \text{complus} &= rf \mid fr \mid co \mid (co;rf) \mid (fr;rf) \\
\text{irreflexive (po-loc; complus) as uniproc}
\end{align*}
\]

In the TSO case we can “optimise”:

- irreflexive rf;RW(po-loc)
- irreflexive fr;WR(po-loc)

because the other coherence violations are rejected by the hb check.

Our final TSO model

TSO3

\[
\begin{align*}
\text{let } \text{comhat} &= rf \mid fr \mid co \mid (co;rf) \mid (fr;rf) \\
\text{irreflexive (po-loc; comhat) as uniproc}
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{ppo} &= (R*M \mid W*W) \& \text{po} \# (W*R) \& \text{po omitted} \\
\text{let } \text{com-hb} &= rfe \mid fr \mid co \# \text{rfi omitted} \\
\text{acyclic ppo \mid mfence \mid com-hb as } \text{hb}
\end{align*}
\]

Notice: There are two checks... The axiomatic frameworks defines principles that the operational model/hardware implement.

For instead, we do not explain how uniproc is implemented. Instead, we specify admissible behaviours.

A word on uniproc

An alternative definitions of “coherence” amounts to “SC per location”.


**Definition (Uniproc 1)**

\[
\text{Acyclic } (\text{po-loc} \leftrightarrow \text{com})
\]

with \(\text{com} = \rightarrow \cup \text{co} \cup \text{fr} \).

From cycle analysis, we have the more attractive definition (since relying on local action of the core and on the existence of coherence orders):

**Definition (Uniproc 2)**

\[
\text{Irreflexive } (\text{po-loc} \rightarrow; \text{com})
\]

Definitions are equivalent.
Equivalence of uniproc definitions

Uniproc 1 \(\implies\) Uniproc 2 is obvious, as \(\text{po-loc} \cup \text{com} \) is included in \(\text{po-loc} \cup \text{com}^+\) (since \(\text{com} = (\text{com})^+\)).

Conversely, we use the “Identical locations” lemma.

Consider a cycle in \(\text{po-loc} \cup \text{com}\), s.t. for all \(e_1 \xrightarrow{\text{po}} e_2\) steps we do not have \(e_2 \xrightarrow{\text{com}} e_1\). Then, for a given \(e_1 \xrightarrow{\text{po}} e_2\) step:
- Either, \(r_1 \xrightarrow{\text{fr}} r_2\), with \(w \xrightarrow{\text{rf}} r_1\) and \(w \xrightarrow{\text{rf}} r_2\). We short-circuit the \(\text{po}\) step, replacing \(w \xrightarrow{\text{rf}} r_1 \xrightarrow{\text{po}} r_2\) by \(w \xrightarrow{\text{rf}} r_2\).
- Or, \(e_1 \xrightarrow{\text{co}} e_2\). We replace the \(\text{po}\) step by \(\text{com}\) steps.

As a result we have a cycle in \(\text{com}\), which is impossible.

From TSO to x86-TSO: locked instructions

Those instructions perform a load then a store to the same location: they generate an atomic pair \(r \xrightarrow{\text{rmw}} w\). Additionally, \(r\) and \(w\) are tagged “atomic”.

**Example:** \text{xchg}1 \(r,x\).

We further enforce:

- Writes \(w'\) to the location are either before the pair or after it:
  \[
  \left( r \xrightarrow{\text{rmw}} w \right) \implies \left( w' \xrightarrow{\text{rf}} r \vee w' \xrightarrow{\text{co}} r \vee w \xrightarrow{\text{co}} w' \right)
  \]

  Or more concisely, we forbid \(r \xrightarrow{\text{fr}} w' \xrightarrow{\text{co}} w\), that is no \(w'\) in-between.

  \[
  \text{rmw} \cap \left( \text{fr}, \text{co} \right) = \emptyset
  \]

- “Fence semantics”: locked instructions act as fences.

ATOM check

The \text{ATOM} check forbids this execution:

<table>
<thead>
<tr>
<th>EXCH</th>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)x ← 1</td>
<td>r ← 2</td>
<td>(b/c)r0 ← x</td>
</tr>
</tbody>
</table>

Observed? \(r=0; y=2\)

<table>
<thead>
<tr>
<th>SB+EXCH</th>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>r ← 1</td>
<td>r ← 1</td>
<td></td>
</tr>
<tr>
<td>(a/b)r ← x</td>
<td>(d/e)r ← y</td>
<td></td>
</tr>
<tr>
<td>(c)r0 ← y</td>
<td>(f)r1 ← x</td>
<td></td>
</tr>
</tbody>
</table>

Observed? \(r0=0; r1=0\)

Implied fences

Implied fences forbid this execution

<table>
<thead>
<tr>
<th>SB+EXCH</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Rx*=0</td>
<td>d: Ry*=0</td>
<td></td>
</tr>
<tr>
<td>b: Wx*=1</td>
<td>e: Wy*=1</td>
<td></td>
</tr>
<tr>
<td>c: Ry=0</td>
<td>f: Rx=0</td>
<td></td>
</tr>
</tbody>
</table>

Cycle: \(b \xrightarrow{\text{implied}} c \xrightarrow{\text{fr}} e \xrightarrow{\text{implied}} f \xrightarrow{\text{fr}} \).
**x86-TSO model for herd**

"X86 TSO"

(* Uniproc *)
let comhat = rf | fr | co | (co;rf) | (fr;rf)
irreflexive (po-loc; comhat) as uniproc

(* Atomic pairs *)
empty rmw & (fre;coe) as atom

(* Implied fences (restricted to WR pairs) *)
let poWR = (W*R) & po
let implied = (M*A | A*M) & poWR

(* Happens-before *)
let ppo = (R*M | W*W) & po # W*R pairs omitted
let com-hb = rfe | fr | co # rfi omitted
acyclic ppo | mfence | implied | com-hb as hb

---

**Part 4.**

**Axiomatic ARM/Power**

---

**Situation of (our) ARM/Power models**

- **Architecture public reference**: Informal, cannot clearly explain how fences restore SC for instance.
- **Simple, global-time model**: (CAV’10) too relaxed. It remains useful as it supports simple reasoning on SC-violations (CAV’11).
- **Operational model**: (PLDI’11) more precise, developed with IBM experts. It is quite complex, and the simulator is very slow.
- **Multi-event axiomatic model**: (CAV’12) more precise (equivalent to PLDI’11), uses several events per access.
- **Single-event axiomatic model**: (...) more precise (proved to be more relaxed than PLDI’11, experimentally equivalent). A more simple axiomatic model.

Joint work with (in order of appearance) Jade Alglave, Susmit Sarkar, Peter Sewell, Derek Williams, Kayvan Memarian, Scott Owens, Mark Batty, Sela Mador-Haim, Rajeev Alur, Milo M. K. Martin and Michael Tautschnig.

---

**A relaxed shared memory computer**

More or less visible to user code:

- **Cores**: Out of order execution, Branch speculation
- **Memory**: Write buffers
- **Memory**: Physically distributed, Caches
Some issues for ARM/Power

- No simple preserved-program-order. More precisely, $\text{pppo}\rightarrow$ will now account for core constraints, such as dependencies.
- Communication relations alone do not define happen-before steps.
- A variety of memory fences: lightweight (Power $\text{lwsync}$) and full (Power $\text{sync}$).

Two-threads SC violation for ARM

Generating tests is as simple as:

```
% diy -conf 2.conf -arch ARM
```

With the same configuration file 2.conf as for X86. Then, compile (in two steps, generate C locally, compile it on target machine), run and...

Observation R Sometimes 5722 1994278
Observation MP Sometimes 17439 1982561
Observation S Sometimes 7270 1992730
Observation SB Sometimes 9788 1990212
Observation LB Sometimes 4782 1995218
All Non-SC behaviours observed!

No hope to define $\text{pppo}\rightarrow$ as simply as for TSO.

An experiment on ARM/Power

Consider test MP:

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$x \gets 1$</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>$y \gets 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observed?</td>
<td>$r_0 = 1; r_1 = 0$</td>
</tr>
</tbody>
</table>

We know that the test is Ok (observed, valid) on ARM/Power, what does it take (amongst fences, dependencies,) to make the test No (unobserved, invalid)?

- Fences: $\text{dsb}, \text{dmb}, \text{isb}$ (ARM); $\text{sync}, \text{lwsync}, \text{isync}$ (Power).
- Dependencies: address, data, control, control+$\text{isb/isync}$.

Dependencies (Power)

Address dependency:

```
(lwz r1,0(r8)) # r8 contains the address of 'x'
slw r7,r1,2 # sizeof(int) = 4
lwzx r2,r7,r9 # r9 contains the address of 't'
```

Data dependency:

```
(r1 ← x)
(y ← r1+1)
stw r2,0(r9) # r9 contains the address of 'y'
```

Control dependency: (+$\text{isync}$)

```
lwz r1,0(r8)
cmpwi r1,0
if $r_1 = 0$ then
  (isync)
  li r2,1
  stw r2,0(r9)
L1:
```
Generating tests (ARM), yet another tool: diycross

Generating tests with diycross (demo in demo/diycross):

```bash
% diycross -arch ARM
PodWW,DMBdWW,DSBdWW,ISBdWW
Rfe
PodRR,DpCtrl1dR,DpCtrl1IsbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR
Fre
```

Generator produced 28 tests

- One generates
  - `MP` as `diyone PodWW Rfe PodRR Fre`
  - `diycross r1, . . . , r1N, . . . , rM` generates the \( N_1 \times \cdots \times N_M \) cycles \( r_1 \times \cdots \times r_M \) by cross-producting the given edge list arguments.

This generates some variations in the `MP` family.

We then compile and run, and...

---

Optimal fencing/dependencies for MP

---

Some observations

In the previous slide we considered increasing power (and cost):

\[
\text{addr} < \text{lwsync} < \text{sync}
\]

Then:
- Dependencies (address) are sufficient to restore order from reads to writes and reads in two-threads examples (but...)
- Fences restore order from writes to write and reads.
- Full fence (\text{sync}) is required from write to read.
- When to use the lightweight fence between writes is complex: \( 2+2W+lwsyncs \) vs. \( R+lwsync+sync \).

---

Optimal fencing for the 6 two-threads tests (Power)
Dependencies are enough

\[
\begin{array}{c|c}
\text{CAUSAL} & \\
\hline
T_0 & T_1 \\
\hline
(a) r_0 \leftarrow x & (c) r_1 \leftarrow y \\
(b) y \leftarrow r_0 & (d) x \leftarrow r_1 \\
\end{array}
\]

Observe? \( r_0=42; r_1=42 \):
\[
\begin{array}{c}
a: R_x=42 \quad \text{data} \\
b: W_y=42 \quad \text{data} \\
c: R_y=42 \\
d: W_x=42 \\
\end{array}
\]

Of course we never observe this behaviour (values out of thin air) and any (hardware) model should forbid it.

**Happens-before** If we order: (1) stores: the point in time when the value is made available to other threads (2) loads: the point when the value is read by core.

Dependencies from reads not always enough!

Consider test \( \text{WRC+data+addr} \):

\[
\begin{array}{c|c|c}
\text{WRC} & T_0 & T_1 \\
\hline
(a) x \leftarrow 1 & (b) r_0 \leftarrow x & (d) r_1 \leftarrow y \\
(c) y \leftarrow 1 & (e) r_1 \leftarrow x \\
\end{array}
\]

Observe? \( r_0=1; r_1=0; \):
\[
\begin{array}{c}
a: W_x=1 \quad \text{rf} \\
b: R_x=1 \quad \text{data} \\
c: W_y=1 \quad \text{rf} \\
d: R_y=1 \quad \text{addr} \\
e: R_x=0 \\
\end{array}
\]

Behaviour observed on Power 6 and 7 (not on ARM, but documentation allows it).

Stores are not “multi-copy atomic” \( T_0 \) and \( T_1 \) share a private buffer/cache/memory (e.g. a cache in SMT context). \( T_2 \) “does not see” the store by \( T_0 \), when \( T_1 \) does.

Restoring SC for \( \text{WRC} \)

Use a lightweight fence on \( T_1 \):

\[
\begin{array}{c|c|c}
T_0 & T_1 & T_2 \\
\hline
a: W_x=1 \quad \text{rf} \\
b: R_x=1 \quad \text{lwsync} \\
c: W_y=1 \\
ed: R_y=1 \quad \text{addr} \\
\end{array}
\]

Observation: The fence orders the writes \( a \) (by \( T_0 \)) and \( c \) (by \( T_1 \)) for any observer (here \( T_2 \)).

Another case of insufficient dependencies

Consider test \( \text{IRIW+addrs} \):

\[
\begin{array}{c|c|c|c}
\text{IRIW} & T_0 & T_1 & T_2 \\
\hline
(a) x \leftarrow 1 & (b) r_0 \leftarrow x & (d) y \leftarrow 1 & (e) r_2 \leftarrow y \\
(c) r_1 \leftarrow y & (f) r_3 \leftarrow x \\
\end{array}
\]

Observe? \( r_0=1; r_1=0; r_2=1; r_3=0; \):
\[
\begin{array}{c}
a: W_x=1 \quad \text{rf} \\
b: R_x=1 \quad \text{addr} \\
c: W_y=1 \quad \text{rf} \\
d: R_y=1 \quad \text{addr} \\
e: R_y=1 \quad \text{addr} \\
f: R_x=0 \\
\end{array}
\]

Behaviour observed on Power (not on ARM, but documentation allows it).

Stores are not “multi-copy atomic”: \( T_0 \) and \( T_1 \) have a private buffer/cache/memory, \( T_2 \) and \( T_3 \) also have one.
Restoring SC for **IRIW**

Use a full fence on \( T_1 \) and \( T_2 \):

\[
\begin{array}{c}
T_0 \quad \text{rf} \quad T_1 \quad \text{rf} \quad T_2 \quad \text{rf} \quad T_3 \\
a: Wx=1 & b: Rx=1 & d: Wy=1 & e: Ry=1 \\
c: Ry=0 & f: Rx=0 \\
\end{array}
\]

**Propagation**: Full fences order all communications.

**Relation summary**

- **Communication relations**:
  - **Read-from**: \( w \xrightarrow{rf} r \), with \( \text{loc}(w) = \text{loc}(r) \), \( \text{val}(w) = \text{val}(r) \).  
  - **Coherence**: \( w \xrightarrow{co} w' \), with \( \text{loc}(w) = \text{loc}(w') \). Total order for given \( x \): hence "coherence orders".
  - We deduce from-read: \( r \xrightarrow{fr} w \), i.e \( w' \xrightarrow{rf} r \) and \( w' \xrightarrow{co} w \).
  - We distinguish internal (same proc, \( rfe, coe, fre \)) and external (different procs, \( rfe, coils, fre \)) communications.

- **Execution** relations
  - **Program order**: \( e_1 \xrightarrow{po} e_2 \), with \( \text{proc}(e_1) = \text{proc}(e_2) \).
  - **Same location program order**: \( e_1 \xrightarrow{po-loc} e_2 \).
  - **Preserved program order**: \( e_1 \xrightarrow{ppo} e_2 \), with \( \xrightarrow{ppo} \subseteq \xrightarrow{po} \). Computed from other relations, includes (effective) dependencies (control dependency from read to read is not effective).
  - Fences: effective strong and lightweight fences in between events \( \xrightarrow{fr} \) and \( \xrightarrow{light} \). Effective means that for instance \( w \xrightarrow{light} r \) does not implies \( w \xrightarrow{light} r \).

**A model in four checks (TOPLAS'14)**

**UNIPROC**

acyclic poloc | com as uniproc

**No-Thin-Air**

let fence = strong | light
let hb = ppo | fence | rfe
acyclic hb as no-thin-air

**Observation** We now define the effect of fences (any fence) for ordering writes:

\[
\text{let propbase} = (((W*W) & fence) | (rfe; ((R*W) & fence)));hb* \]

irreflexive fre;propbase as observation

**Propagation** Strong fences wait for all communications.

\[
\text{let prop} = (W*W) & propbase | (com*;propbase*;strong;hb*) \]

acyclic co | prop as propagation

**ARM/Power preserved program order**

Rather complex, results from a two events per access analysis (cf. CAV'12).

(* Utilities *)

let dd = addr | data let rdw = po-loc & (fre;rfe)
let detour = po-loc & (coe ; rfe) let addrpo = addr;po

(* Initial value *)

let ci0 = ctrlisync | detour
let ii0 = dd | rfi | rdw
let cc0 = dd | po-loc | ctrl | addrpo
let ic0 = 0

(* Fixpoint from \( i \rightarrow c \) in instructions and transitivity *)

let rec ci = ci0 | (ci;ii) | (cc;ci)
and ii = ii0 | ci | (ic;ci) | (ii;ii)
and cc = cc0 | ci | (ci;ic) | (cc;cc)
and ic = ic0 | ii | cc | (ic;cc) | (ii ; ic)

let ppo = RW(ic) | RB(ii)
Can be limited to dependencies...
How good is our model?

Is it sound?
- A proof: any behaviour allowed is also allowed by the operational model of PLDI’11.
- Experiments
  - Soundness w.r.t. hardware (ARM being a bit problematic because of acknowledged read-after-read hazard).
  - Experimental equivalence with our previous models, saved from current debate on some subtle semantical point for lwsync.

In any case:
- Simulation is fast ($\times 1000$ w.r.t. PLDI’11) ($\times 10$ w.r.t. CAV’12).
- The existence of four checks uniproc, hb observation and propagation stand on firm bases.
- The semantics of strong fences also does.
- The model and simulator (i.e. herd) are flexible, one easily change a few relations (e.g. \( \text{ppo} \rightarrow \)), or the semantics of weak fences).

A test of coherence violation

Our setting also finds bugs...

The following execution:

is observed on all (tested) ARM machines. It features a CoRR-style coherence violation (i.e. \( \text{po} \rightarrow \text{contradicts} \frac{\text{fr}}{}; \frac{\text{fr}}{\text{po}} \)).

Notice: CoRR is not observed directly.

<table>
<thead>
<tr>
<th>( T_0 )</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( x \leftarrow 2 )</td>
<td>(c) ( y \leftarrow 2 )</td>
<td>(d) ( r_0 \leftarrow z )</td>
</tr>
<tr>
<td>(d) ( y \leftarrow 1 )</td>
<td>(e) ( z \leftarrow 1 )</td>
<td>(f) ( x \leftarrow 1 )</td>
</tr>
</tbody>
</table>

Observe? \( x=2; y=2; r_0=1 \)

\( \text{lwsync} \rightarrow \text{co} \rightarrow \text{lwsync} \rightarrow \text{addr} \)

Unobserved and forbidden by model. May be allowed...