Part 1.

Axiomatic Sequential Consistency

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.
Formalism: events

The effect of "operations executed by the processors" are represented by events.

Operations we consider are the memory accesses. Hence, we define memory events \((a,d[\ell]v)\), where:

- 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)

Formalism: program order

The program order \(\overset{\text{po}}{\rightarrow}\) is a linear order amongst the events originating from the same processor.

Relation \(\overset{\text{po}}{\rightarrow}\) represents the sequential execution of events by one processor that follows the usual processor execution model, where instructions are executed by following the order given in program.

Example

/* 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:

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

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:

1. Order \(<\) is compatible with program order:
   \[e_1 \overset{\text{po}}{\rightarrow} e_2 \implies e_1 < e_2.\]

2. Reads read from the closest write upwards (a.k.a. “most recent”):
   \[\overset{\text{rf}}{\prec} \overset{\text{Def}}{=} \{(w, r) | w = \max(w', \text{loc}(w')) = \text{loc}(r) \land w' < r\}.\]

Example of a question on SC

<table>
<thead>
<tr>
<th>(R)</th>
<th>(T_0)</th>
<th>(T_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\overset{\text{rf}}{\prec})</td>
<td>(a)</td>
<td>(x \leftarrow 1)</td>
</tr>
<tr>
<td></td>
<td>(b)</td>
<td>(y \leftarrow 1)</td>
</tr>
<tr>
<td></td>
<td>(c)</td>
<td>(y \leftarrow 2)</td>
</tr>
<tr>
<td></td>
<td>(d)</td>
<td>(x \leftarrow 0)</td>
</tr>
</tbody>
</table>

Observed? \(y=2, r0=0\)

How do we know? Let us enumerate all interleavings:

\[
\begin{align*}
\text{a, b, c, d} & \quad y=2, r0=1; \\
\text{a, c, b, d} & \quad y=1, r0=1; \\
\text{a, c, d, b} & \quad y=1, r0=1; \\
\text{c, d, a, b} & \quad y=1, r0=0; \\
\text{c, a, b, d} & \quad y=1, r0=1; \\
\text{c, a, d, b} & \quad y=1, r0=1;
\end{align*}
\]

Remark: if \(b < c\) then \(y=2\), if \(a < d\) then \(r0=1\).
Let us be a bit more clever

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$x \leftarrow 1$</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>$y \leftarrow 1$</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>$y \leftarrow 2$</td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td>$r_0 \leftarrow x$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed?</td>
<td>$y=2$; $r_0=0$</td>
<td></td>
</tr>
</tbody>
</table>

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!

\[ a < b < c < d < a \ldots \]

**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 ($\overset{rf}{\longrightarrow}$):** Relates write events to read events that read the stored value (initial writes left implicit in diagrams).
  \[ \forall r, \exists! w, w \overset{rf}{\longrightarrow} r \]
  *(Notice: $w$ and $r$ have identical location and value.)*

- **Coherence ($\overset{co}{\longrightarrow}$):** Relates write events to the same location.
  For any location $\ell$, the restriction of $\overset{co}{\longrightarrow}$ to write events to location $\ell$ ($W_\ell$) is a total order.

---

**Coherence as a characteristics of shared memory**

The very existence of $\overset{co}{\longrightarrow}$ is implied by the existence of a shared, coherent, memory — Given location $x$, there is exactly one memory cell whose location is $x$.

\[
W_x^0 \overset{co}{\longrightarrow} W_x^1 \overset{co}{\longrightarrow} x = 2 \overset{co}{\longrightarrow} W_x^3 \overset{co}{\longrightarrow} \ldots
\]

Of course, in reality, there caches, buffers etc. But the system will behave "as if".

---

**Example of $\overset{rf}{\longrightarrow}$**

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$r_0 \leftarrow x$</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>$y \leftarrow 1$</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>$r_1 \leftarrow y$</td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td>$x \leftarrow 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observe: $r_0; r_1$;</td>
<td>$r_0=1; r_1=1$;</td>
<td>$r_0=1; r_1=0$;</td>
</tr>
</tbody>
</table>

There are 4 possible $\overset{rf}{\longrightarrow}$ relations (initial value is 0).

\[
\begin{array}{c|cc|cc}
\hline
& \text{a: Rx=1} & \text{c: Ry=1} & \text{a: Rx=1} & \text{c: Ry=0} \\
\hline
\text{po} & \text{po} & \text{po} & \text{po} & \text{po} \\
\text{b: Wy=1} & \text{d: Wx=1} & \text{b: Wy=1} & \text{d: Wx=1} \\
\hline
\end{array}
\]

\[
\begin{array}{c|cc|cc}
\hline
& \text{a: Rx=0} & \text{c: Ry=1} & \text{a: Rx=0} & \text{c: Ry=0} \\
\hline
\text{po} & \text{po} & \text{po} & \text{po} & \text{po} \\
\text{b: Wy=1} & \text{d: Wx=1} & \text{b: Wy=1} & \text{d: Wx=1} \\
\hline
\end{array}
\]
Example of $\rightarrow^\text{co}$

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

Observe: $x=1; y=1$

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$x \leftarrow 1$</td>
<td>$c: Wx=1$</td>
</tr>
<tr>
<td>(c)</td>
<td>$y \leftarrow 1$</td>
<td>$c: Wy=1$</td>
</tr>
<tr>
<td></td>
<td>($d$) $x \leftarrow 1$</td>
<td></td>
</tr>
</tbody>
</table>

Observe: $x=1; y=1$

**Notice:** In this simple case of two stores, the value finally observed in locations determines $\rightarrow^\text{co}$ for them.

One more relation: $\rightarrow^\text{fr}$

The new relation $\rightarrow^\text{fr}$ (from read) relates reads to “younger writes” (younger w.r.t. $\rightarrow^\text{co}$).

$$r \rightarrow^\text{fr} w \overset{\text{Def}}{=} w' \rightarrow^\text{fr} r \wedge w' \rightarrow^\text{co} w$$

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

$$w_0 \rightarrow^\text{co} w_1 \rightarrow^\text{co} \ldots \rightarrow^\text{co} w_n$$

We have

$$w_0 \rightarrow^\text{fr} r \rightarrow^\text{fr} w_1 \rightarrow^\text{fr} \ldots \rightarrow^\text{fr} w_n$$

(Or: $\rightarrow^\text{fr} \overset{\text{Def}}{=} (\rightarrow^\text{fr})^{\rightarrow^\text{co}}$)

Second definition of SC

**Definition (SC 2)**

An execution is SC when:

$$\text{Acyclic}(\rightarrow^\text{fr} \cup \rightarrow^\text{co} \cup \rightarrow^\text{fr} \cup \rightarrow^\text{po})$$

And of course:

**Theorem**

*The two definitions of SC are equivalent.*
SC 1 $\implies$ SC 2

Assume the existence of the total order “<”.
Define:
\[
\text{co} \overset{\text{Def}}{=} \{ (w_1, w_2) \mid \text{loc}(w_1) = \text{loc}(w_2) \land w_1 < w_1 \}.
\]

Notice that $\rightarrow$ is already defined: $\rightarrow = \overset{\text{Def}}{=} \rightarrow <$. Also notice $\rightarrow < \subseteq <$, $\rightarrow < \subseteq <$ and $\rightarrow \subseteq <$.

Proof:
Define $\rightarrow_{\text{fr}} = \rightarrow^{-1} \cup \rightarrow$; and prove $\rightarrow_{\text{fr}} \subseteq <$.

Let $r \rightarrow_{\text{fr}} w$. Let further $w_0 \rightarrow_{\text{fr}} r$, then, by definition of $\rightarrow_{\text{fr}}$, we have $w_0 \rightarrow_{\text{co}} w$ and thus $w_0 < w$.

But, $w_0$ is maximal amongst all $w' < r$. That is: “$w' < r \implies w' \leq w_0$” or, “$w_0 < w \implies r < w$” QED.

Hence, a cycle in $\rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}}$ would be a cycle in order “<”

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. $\rightarrow_{\text{po}}, \rightarrow_{\text{fr}}, \rightarrow_{\text{co}}$); check the acyclicity of $\rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}}$.

Answer: we view SC 2 as being more convenient, since the generated objects usually are smaller.

SC 2 $\implies$ SC 1

Since $\rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}} \cup \rightarrow_{\text{fr}} \cup \rightarrow_{\text{co}}$ is a partial order, there exists a total order $<$ that “extends” it (no question on mathematical foundations, . . . ).

From < define $\rightarrow_{<}$:
\[
\rightarrow_{<} = \{ (w, r) \mid w = \max < (w', \text{loc}(w') = \text{loc}(r) \land w' < r) \}.
\]

and show $\rightarrow_{fr} = \rightarrow_{fr} <$.

Let $w_0 \rightarrow_{fr} r$ and let $w \in W$, $w \neq w_0$ then ($\rightarrow_{co}$ total order on $W$):
- Either $w \rightarrow_{co} w_0$ and $w < w_0 < r$.
- Or, $w_0 \rightarrow_{co} w$, and $r \rightarrow_{fr} w$, and thus $r < w$.

Finally $w_0 \rightarrow_{fr} r$.

Let $w \rightarrow_{fr} r$ (i.e. $w \in W$, $w \neq w_0$), then
- Either $w \rightarrow_{co} w_0$, and thus ($\rightarrow_{co} \subseteq <$) $w \rightarrow_{fr} r$.
- Or $w_0 \rightarrow_{co} w$, thus $r \rightarrow_{fr} w$, and thus ($\rightarrow_{fr} \subseteq <$) $w \rightarrow_{fr} r$.

Introducing herd, a memory model simulator

A model sc.cat:

```plaintext
% 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.
Part 2.

Studying Non-Sequentially Consistent Executions.

Violations of SC

A cycle of $\rightarrow po, \rightarrow rf, \rightarrow co, \rightarrow fr$ 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.

Simplifying cycles: $\rightarrow po$ and $\rightarrow com$ steps alternate

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

- $\rightarrow po$ is transitive $\rightarrow po^+ \subseteq \rightarrow po$.
- $\rightarrow com^+$ is the union of the five following relations:

$$
\rightarrow com = \rightarrow rf \cup \rightarrow co \cup \rightarrow fr \cup \left( \rightarrow co ; \rightarrow rf \right) \cup \left( \rightarrow fr ; \rightarrow rf \right).
$$

Because $(\rightarrow co ; \rightarrow co) \subseteq \rightarrow co, (\rightarrow fr ; \rightarrow co) \subseteq \rightarrow fr,$ and $(\rightarrow rf ; \rightarrow fr) \subseteq \rightarrow co$.

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

Herd structure

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Wx=1</td>
<td>c</td>
<td>Wy=2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Wy=1</td>
<td>d</td>
<td>Rx=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ok</td>
<td></td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

- Apply model checks to each candidate execution.
Simplifying cycles: all \( \text{com} \rightarrow \) steps are external.

**Simplifying cycles - Threads**

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

\[
\begin{align*}
e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}} e_3 \xrightarrow{\text{po}} e_4 \xrightarrow{\text{com}} e_5 \xrightarrow{\text{po}} e_1
\end{align*}
\]

Assuming for instance, \( e_1 \xrightarrow{\text{po}} e_3 \) then we have a “simpler” cycle:

\[
\begin{align*}
e_1 \xrightarrow{\text{po}} e_3 \xrightarrow{\text{po}} e_4 \xrightarrow{\text{com}} e_5 \xrightarrow{\text{po}} e_1
\end{align*}
\]

(Conclude with \( \xrightarrow{\text{po}} \) being transitive)

If \( e_1 = e_3 \), we also have a simpler cycle:

\[
\begin{align*}
e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}} e_3 = e_1
\end{align*}
\]

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

**Test from cycles — Locations**

Cycle: \( R \xrightarrow{\text{po}} W \xrightarrow{R} W \xrightarrow{\text{po}} R \xrightarrow{\text{po}} W \xrightarrow{\text{po}} R \xrightarrow{\text{po}} W \xrightarrow{\text{po}} R \xrightarrow{\text{po}} W \xrightarrow{\text{po}} W \xrightarrow{\text{po}}
\]

Consider a test execution on two threads:

The test execution features a smaller cycle

- One interpretation (four locations):

a: \( Rx=1 \)  e: \( Ry=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

b: \( Wy=1 \)  f: \( Wz=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

c: \( Rz=1 \)  g: \( Ra=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

d: \( Wa=1 \)  h: \( Wx=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

- Another interpretation (two locations):

a: \( Rx=2 \)  e: \( Rx=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

d: \( Wy=1 \)  f: \( Wx=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

c: \( Ry=2 \)  g: \( Ry=1 \)

\[
\begin{align*}
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po} \\
&\text{po}
\end{align*}
\]

Generally: one passage per thread
The second interpretation is not “minimal”

Reminding the interpretation with two locations:

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Event</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Rx=2</td>
<td>po</td>
<td>e: Rx=1</td>
</tr>
<tr>
<td>c: Ry=1</td>
<td>po</td>
<td>g: Ry=2</td>
</tr>
<tr>
<td>b: Wy=1</td>
<td>rf</td>
<td>f: Wy=2</td>
</tr>
<tr>
<td>d: Wx=1</td>
<td></td>
<td>h: Wx=2</td>
</tr>
</tbody>
</table>
```

But, coherence \( \rightarrow \co \) totally orders write events to a given location.

Let us choose: \( \text{Wx1} \rightarrow \co \rightarrow \text{Wx2} \):
```
<table>
<thead>
<tr>
<th>Location</th>
<th>Event</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Rx=2</td>
<td>po</td>
<td>e: Rx=1</td>
</tr>
<tr>
<td>c: Ry=1</td>
<td>po</td>
<td>g: Ry=2</td>
</tr>
<tr>
<td>b: Wy=1</td>
<td>rf</td>
<td>f: Wy=2</td>
</tr>
<tr>
<td>d: Wx=1</td>
<td></td>
<td>h: Wx=2</td>
</tr>
</tbody>
</table>
```

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

Choosing \( \text{Wx2} \rightarrow \co \rightarrow \text{Wx1} \) 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 \( \com \) 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 \( \po \) and one \( \com \) in both directions. By the identical locations lemma:
  - Either, \( e_1 \rightarrow \com \rightarrow e_2 \) or \( e_2 \rightarrow \com \rightarrow e_1 \), and we have a simpler cycle.
  - or, \( \po \rightarrow r \rightarrow e_1 \) and \( \po \rightarrow r \rightarrow e_2 \), see next page!

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

Let \( e_1, e_2 \) two different events with the same location,

- either \( e_1 \rightarrow \com \rightarrow e_2 \),
- or \( e_2 \rightarrow \com \rightarrow e_1 \),
- or \( \po \rightarrow r \rightarrow e_1 \) and \( \po \rightarrow r \rightarrow e_2 \).

Case analysis:

- \( \text{w}_1, \text{w}_2 \), then either \( \text{w}_1 \rightarrow \com \rightarrow \text{w}_2 \) or \( \text{w}_2 \rightarrow \com \rightarrow \text{w}_1 \) (total order).
- \( \text{r}_1, \text{r}_2 \), let \( \text{w}_1 \rightarrow \rf \rightarrow \text{r}_1 \) and \( \text{w}_2 \rightarrow \rf \rightarrow \text{r}_2 \). Then, either \( \text{w}_1 = \text{w}_2 \) and we are in case 3; or (for instance) \( \text{w}_1 \rightarrow \com \rightarrow \text{w}_2 \) and we have \( \text{r}_1 \rightarrow \rf \rightarrow \text{w}_2 \rightarrow \rf \rightarrow \text{r}_2 \).
- \( \text{r}_1, \text{w}_2 \), let \( \text{w}_1 \rightarrow \rf \rightarrow \text{r}_1 \). Then, either \( \text{w}_1 = \text{w}_2 \) and \( \text{w}_2 \rightarrow \rf \rightarrow \text{r}_1 \); or \( \text{w}_1 \rightarrow \com \rightarrow \text{w}_2 \) and \( \text{r}_1 \rightarrow \rf \rightarrow \text{w}_2 \); or \( \text{w}_2 \rightarrow \com \rightarrow \text{w}_1 \) and \( \text{w}_2 \rightarrow \com \rightarrow \rf \rightarrow \text{r}_1 \).

Corollary: \( \com \) is acyclic.

Next page

So let us assume a cycle that includes \( \text{r}_1 \) and \( \text{r}_2 \), related in both directions by complex steps and such that \( \po \rightarrow \rf \rightarrow \text{r}_1 \) and \( \po \rightarrow \rf \rightarrow \text{r}_2 \). We consider:

- If \( \po \rightarrow \rf \rightarrow \text{r}_1 \) is in cycle, then there is an obvious short-circuit: replace \( \rf \rightarrow \text{r}_1 \) followed by the complex steps from \( \text{r}_1 \) to \( \text{r}_2 \) by a single \( \po \rightarrow \rf \rightarrow \text{r}_2 \) step.
- If \( \po \rightarrow \rf \rightarrow \text{r}_2 \) is in cycle, symmetrical case.
- Otherwise, it must be that both \( \text{r}_1 \) and \( \text{r}_2 \) are the target of \( \po \rightarrow \rf \rightarrow \text{r}_1 \) steps and the source of \( \po \rightarrow \rf \rightarrow \text{r}_2 \) steps: let \( \text{w}_1 \) and \( \text{w}_2 \) be the targets of those steps.

Then, in all possible three situations: \( \text{w}_1 = \text{w}_2 \), \( \text{w}_1 \rightarrow \com \rightarrow \text{w}_2 \) and \( \text{w}_2 \rightarrow \com \rightarrow \text{w}_1 \) we construct a simpler cycle that does not contain \( \text{r}_1 \) or \( \text{r}_2 \).
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 $po$ steps and external $com$ steps.
  - The cycle passes through a given thread at most once.
  - All $com$ steps have pairwise different locations.
  - The source and target of one given $po$ 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 $com$, there are five such cycles, which can occur as the following executions: $po$ contradicts $co$, $rf$, $fr$, $”co”, rf$, $”fr”, rf$.

Application, all possible SC violations on two threads

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

$$
\begin{align*}
2+2W & : \xrightarrow{po, co, po, co} \\
LB & : \xrightarrow{po, rf, po, rf} \\
MP & : \xrightarrow{po, rf, po, fr} \\
R & : \xrightarrow{po, co, po, fr} \\
S & : \xrightarrow{po, fr, po, fr} \\
SB & : \xrightarrow{po, fr, po, fr}
\end{align*}
$$

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 # target architecture
-safe Pod**,Rfe,Fre,Wse # vocabulary
-nprocs 2 # 2 procs
-size 4 # max size of cycle (2 X nprocs)
-num false # for naming tests

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...
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 $\rightarrow_p$ 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 $\rightarrow_p$ step of those.

Simplification:
- Insert fences between all pairs of racy accesses with different locations (notice that $\rightarrow_c$ 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 ;
    sync() ;
    b: go = 1 ;
    c: while (go == 1) ;
    sync() ;
    d: while (go == 0) ;
    e: int t = x; sum += t;
    sync() ;
    f: go = 0 ;
}
```

To insert fence, consider separating accesses to go and x.
A semi realistic example, more precise fencing

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

The resulting static $\rightarrow$ relation is as follows.

```
a: W[x]=v  d: R[go]=1
b: W[go]=1  e: R[x]=w
f: W[go]=0
```

Cycle 1

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

```
a $\xrightarrow{\text{loop}}$ b, d $\xrightarrow{\text{ctrlsync}}$ e
```

X86: no fence needed.

Cycle 2

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

```
a $\xrightarrow{\text{sync}}$ b, f $\xrightarrow{\text{sync}}$ e
```

X86: $f \xrightarrow{\text{mfence}} e$

Cycle 3

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

```
a $\xrightarrow{\text{sync}}$ c, f $\xrightarrow{\text{sync}}$ e
```

X86: $a \xrightarrow{\text{sync}} c, f \xrightarrow{\text{mfence}} e$
Cycle 4

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

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

X86: no fence needed.

Cycle 5

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

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

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

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

\[
\text{for (int } k = N ; k >= 0 ; k--) { \\
\quad a: x = k ; \\
\quad mfence() ; \\
\quad b: go = 1 ; \\
\quad c: while (go == 1) ; \\
\quad } \\
\]

\[
\text{int } sum = 0 ; \\
\text{for (int } k = N ; k >= 0 ; k--) { \\
\quad d: while (go == 0) ; \\
\quad e: int } t = x; sum += t; \\
\quad f: go = 0 ; \\
\quad mfence() ; \\
\quad } \\
\]

Notice: Inserting full memory fence between racy writes gives the same result.
Sufficient fencing, Power

\[
\begin{align*}
& a \xrightarrow{\text{lwsync}} b, d \xrightarrow{\text{ctrlisync}} e, \\
& a \xrightarrow{\text{sync}} b, f \xrightarrow{\text{sync}} e, \\
& a \xrightarrow{\text{sync}} c, f \xrightarrow{\text{sync}} e, \\
& b \xrightarrow{\text{lwsync}} a, e \xrightarrow{\text{ctrlisync}} d, \\
& b \xrightarrow{\text{lwsync}} a, e \xrightarrow{\text{ctrl}} f, \\
& c \xrightarrow{\text{ctrl}} a, e \xrightarrow{\text{ctrl}} f.
\end{align*}
\]

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

Part 3.

Axiomatic TSO

Inline assembler for fences and ctrlisync

\[
\begin{align*}
\text{inline static void sync()} \\
& \quad \text{asm } \_\_\text{volatile}\_\_ ("sync" ::: "memory") ;
\end{align*}
\]

\[
\begin{align*}
\text{inline static void lwsync()} \\
& \quad \text{asm } \_\_\text{volatile}\_\_ ("lwsync" ::: "memory") ;
\end{align*}
\]

\[
\begin{align*}
\text{inline static void ctrlisync(int t)} \\
& \quad \text{asm } \_\_\text{volatile}\_\_ ( \\
& \quad \quad \text{"cmpwi } \%[t],0\text{\t}\text{\n\t"} \\
& \quad \quad \text{"beq.0f\text{\t}\text{\n\t"} } \\
& \quad \quad \text{"0:\text{\t}\text{\n\t"} } \\
& \quad \quad \text{"isync}\text{\t}\text{\n\t"} \\
& \quad \quad :: [t] "r" (t) : "memory") ;
\end{align*}
\]

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

TSO — The Model of X86 machines

The write buffer explains how "reads can pass over writes".
An experimental study of x86

Demo: (in demo/TSO1) Compiling:
% litmus7 -mach ./x86 ../diy/src2/@all -o run
% make -C run -j 4

Running:
% cd run
% sh run.sh > X.00

Analysis:
% grep Observation X.00
Observation R Sometimes 79 1999921
Observation MP Never 0 2000000
Observation 2+2W Never 0 2000000
Observation S Never 0 2000000
Observation SB Sometimes 1194 1998806
Observation LB Never 0 2000000

Results for running the six test on this machine

Axiomatic TSO, model TSO 1

- Remember SC:
  \[
  \text{Acyclic} \left( \text{r} \cup \text{fr} \cup \text{co} \cup \text{rf} \cup \text{fr} \cup \text{po} \right)
  \]

  A model for herd, our generic simulator:
  let ppo = po # ppo stands for ‘preserved program-order’
  let com-hb = fr | rf | co # All communications create order
  acyclic (ppo | com-hb)

- In TSO:
  - Write-to-read does not create order:
    let ppo = (R*M | W*W) # ppo # W*R pairs omitted
  - Communication create order
    let com-hb = rf | co | fr
  - TSO “happens-before” (HIB) check:
    acyclic (ppo | com-hb | mfence) as hb

Notice: Relations can be interpreted as being between the points in time where a load binds its value and where a written value reaches memory.

Restoring SC with mfence

Replace “relaxed” (not in HIB) WR(po) by mfence (in HIB).

<table>
<thead>
<tr>
<th>T_0</th>
<th>T_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) x ← 1</td>
<td>(c) y ← 2</td>
</tr>
<tr>
<td>(b) y ← 1</td>
<td>(d) r0 ← x</td>
</tr>
</tbody>
</table>

Observed? \( y=2; r0=0 \)

\( \text{SB+mfences} \)

<table>
<thead>
<tr>
<th>T_0</th>
<th>T_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) x ← 1</td>
<td>(c) y ← 1</td>
</tr>
<tr>
<td>(b) r0 ← y</td>
<td>(d) r1 ← x</td>
</tr>
</tbody>
</table>

Observed? \( r0=0; r1=0 \)
Our TSO 1 model is wrong!

Consider:

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$x \leftarrow 1$</td>
<td>(d)</td>
</tr>
<tr>
<td>(b)</td>
<td>$r0 \leftarrow x$</td>
<td>(e)</td>
</tr>
<tr>
<td>(c)</td>
<td>$r1 \leftarrow y$</td>
<td>(f)</td>
</tr>
</tbody>
</table>

Observed? $r0=1; r1=0; r2=1; r3=0$;

According to model? No. As we have the 
HB cycle:

$$a \xrightarrow{rf} b \xrightarrow{po} c \xrightarrow{fr} d \xrightarrow{rf} e \xrightarrow{po} f \xrightarrow{fr} a$$

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

The effect originates from "store forwarding": A thread can read its own writes from its store buffer, i.e. before they reach memory.

Corrected model: TSO 2

Internal $\xrightarrow{rf}$ (result of $\xrightarrow{rfi}$) does not create order, external $\xrightarrow{rf}$ ($\xrightarrow{rfe}$) does:

let com-hb = rfe | fr | co #rfi not in hb
acyclic ppo | com-hb | mfence

The new hb is no longer cyclic:

(Also consider that $a \xrightarrow{po} \text{WR} c$ and $d \xrightarrow{po} \text{WR} f$ are non-global.)

This is not over yet...

Our TSO 2 model:

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

Allows two violations of coherence:

(a) $Rx=1$
(b) $Wx=1$
(c) $Ry=0$
(d) $Wy=1$
(e) $Ry=1$
(f) $Rx=0$

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:

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

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

let complus = rf | fr | co | (co;rf) | (fr;rf)

irreflexive (po-loc; complus) as uniproc

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**

let comhat = rf | fr | co | (co;rf) | (fr;rf)

irreflexive (po-loc; comhat) as uniproc

let ppo = (R*M | W*W) & po # (W*R) & po omitted

let com-hb = rfe | fr | co # rfi omitted

acyclic ppo | mfence | com-hb as hb

**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)**

Acyclic \(\left( \text{po-loc } \rightarrow \cup \text{com } \rightarrow \right)\)

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

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)**

Irreflexive \(\left( \text{po-loc } \rightarrow; \text{com } \rightarrow \right)\)

Definitions are equivalent.

---

**Equivalence of uniproc definitions**

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

Conversely, we use the “Identical locations” lemma.

Consider a cycle in \(\text{po-loc } \rightarrow \cup \text{com } \rightarrow\), s.t. for all \(e_1 \text{po-loc } \rightarrow e_2\) steps we do not have \(e_2 \text{com } \rightarrow e_1\). Then, for a given \(e_1 \text{po } \rightarrow e_2\) step:

- Either, \(r_1 \text{po } \rightarrow r_2\), with \(w \text{rf } \rightarrow r_1\) and \(w \text{rf } \rightarrow r_2\). We short-circuit the \(\text{po } \rightarrow\) step, replacing \(w \text{rf } \rightarrow r_1 \text{po } \rightarrow r_2\) by \(w \text{rf } \rightarrow r_2\).
- Or, \(e_1 \text{com } \rightarrow e_2\). We replace the \(\text{po } \rightarrow\) step by \(\text{com } \rightarrow\) steps.

As a result we have a cycle in \(\text{com } \rightarrow\), 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 \( r \rightarrow w \). Additionally, \( r \) and \( w \) are tagged “atomic”.

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

We further enforce:
- Writes \( w' \) to the location are either before the pair or after it:
  \[
  (r \xrightarrow{\text{rmw}} w) \implies \left( \begin{array}{c}
  w' \xrightarrow{\text{fr}} r \lor w' \xrightarrow{\text{co}} r \lor w \xrightarrow{\text{co}} w'
  \end{array} \right)
  \]

  Or more concisely, we forbid \( r \xrightarrow{\text{fr}} w' \xrightarrow{\text{co}} w \), that is no \( w' \) in-between.
  \[
  \xrightarrow{\text{rmw}} \cap \left( \begin{array}{c}
  \left( \begin{array}{c}
  \xrightarrow{\text{fr}} ; \\
  \xrightarrow{\text{co}} 
  \end{array} \right)
  \end{array} \right) = \emptyset
  \]

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

**Implied fences**

Implied fences forbid this execution

\[
\begin{array}{c|c|c}
\text{SB+EXCH} & T_0 & T_1 \\
\hline
r & \xleftarrow{1} & \xleftarrow{1} \\
(a/b) r & \xleftarrow{x} & (d/e) r & \xleftarrow{y} \\
(c) r & 0 & y & (f) r & 1 & \xleftarrow{x} \\
\hline
\text{Observed?} & r=0; & y=2
\end{array}
\]

\begin{tabular}{c}
\text{Cycle: } b & \xrightarrow{\text{implied}} & c & \xrightarrow{\text{fr}} & e & \xrightarrow{\text{implied}} & f & \xrightarrow{\text{fr}}.
\end{tabular}

**ATOM check**

The ATOM check forbids this execution:

\[
\begin{array}{c|c|c}
\text{EXCH} & T_0 & T_1 \\
\hline
(a) x & \xleftarrow{1} & r & \xleftarrow{2} \\
\hline
\text{Observed?} & r=0; & y=2
\end{array}
\]

**x86-TSO model for herd**

"X86 TSO"

\[
\begin{array}{c}
(* \text{Unproc} *)
\end{array}
\]

let \( \text{comhat} = \text{rf} \mid \text{fr} \mid \text{co} \mid \text{(co;rf)} \mid \text{(fr;rf)} \)

irreflexive \((\text{po-loc}; \text{comhat})\) as uniproc

\[
\begin{array}{c}
(* \text{Atomic pairs} *)
\end{array}
\]

let \( \text{poWR} = (\text{W*R}) \land \text{po} \)

let \( \text{implied} = (\text{M*A} \mid \text{A*M}) \land \text{poWR} \)

\[
\begin{array}{c}
(* \text{Happens-before} *)
\end{array}
\]

let \( \text{ppo} = (\text{R*M} \mid \text{W*R}) \land \text{po} \)

let \( \text{com-hb} = \text{rfe} \mid \text{fr} \mid \text{co} \land \text{frf} \) omitted

acyclic \( \text{ppo} \mid \text{mfence} \mid \text{implied} \mid \text{com-hb} \) 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.

Some issues for ARM/Power

- No simple preserved-program-order. More precisely, \( \text{ppp} \) 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 \( lwsync \)) and full (Power \( sync \)).
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 $\mathit{ppo} \xrightarrow{\mathit{mp}}$ as simply as for TSO.

An experiment on ARM/Power

Consider test MP:

<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) $y \leftarrow 1$</td>
<td>(d) $r_1 \leftarrow x$</td>
</tr>
</tbody>
</table>

Observed? $r_0=1; r_1=0$

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: dsb, dmb, isb (ARM); sync, lwsync, isync (Power).
- Dependencies: address, data, control, control+isb/isync.

Generating tests (ARM), yet another tool: diycross

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

% diycross --arch ARM\nPodWW,DMBDWW,DSBDWW,ISBDWW\RFef\PodRR,DpCtrl1R,DpCtrl1IsbdR,DpAddrdR,DMBDRR,DSBDRR,ISBDRR\Fref

Generator produced 28 tests

- One generates MP as diyone PodWW Rfe PodRR Fre
- diycross $r_1, \ldots, r_k, \ldots, r_M$, generates the $N_1 \times \cdots \times N_M$ cycles $r_1^1 \ldots r_k^1 \cdots r_M^1$ by cross-producting the given edge list arguments.

This generates some variations in the MP family.

We then compile and run, and...

<table>
<thead>
<tr>
<th>Dependencies (Power)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address dependency:</td>
</tr>
<tr>
<td>$r_1 \leftarrow x$</td>
</tr>
<tr>
<td>$r_2 \leftarrow t[r_1]$</td>
</tr>
</tbody>
</table>

Data dependency:

$r_1 \leftarrow x$
$y \leftarrow r_1+1$

Control dependency: (+isync)

$r_1 \leftarrow x$
if $r_1=0$ then
(isync)
$y \leftarrow 1$

$\text{lzw } r_1,0(r_8)$ # $r_8$ contains the address of ‘x’
$\text{slwi } r_7, r_1, 2$ # size of(int) = 4
$\text{lwx } r_2, r_7, r_9$ # $r_9$ contains the address of ‘t’
$\text{add } r_2, r_1, 1$
$\text{stw } r_2, 0(r_9)$ # $r_9$ contains the address of ‘y’

lwz $r_1,0(r_8)$ # $r_8$ contains the address of ‘x’
cmpwi $r_1, 0$
bne L1
(isync)
li $r_2, 1$
stw $r_2, 0(r_9)$
L1:
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 (sync) is required from write to read.
- When to use the lightweight fence between writes is complex: \(2+2W+lwsyncs\) vs. \(R+lwsync+sync\).

\[
\begin{align*}
(a) & \quad r_0 \leftarrow x \\
(b) & \quad y \leftarrow r_0 \\
(c) & \quad r_1 \leftarrow y \\
(d) & \quad x \leftarrow r_1
\end{align*}
\]
Observed? \( r_0=42; r_1=42 \);

\[
\begin{align*}
(a) & \quad Rx=42 \\
(b) & \quad Wy=42 \\
(c) & \quad Ry=42 \\
(d) & \quad Wx=42
\end{align*}
\]

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 **WRC+data+addr**:

<table>
<thead>
<tr>
<th></th>
<th>( T_0 )</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a)) ( x \leftarrow 1 )</td>
<td>((b) r_0 \leftarrow x )</td>
<td>((d) r_1 \leftarrow y )</td>
<td>( )</td>
</tr>
<tr>
<td>((c) y \leftarrow 1 )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

Observed? \( r_0=1; r_1=0 \);

- \( a: Wx=1 \)
- \( b: Rx=1 \)
- \( d: Ry=1 \)
- \( c: Wy=1 \)
- \( e: Rx=0 \)

**WRC+data+addr**

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 WRC**

Use a lightweight fence on \( T_1 \):

- \( T_0 \)
- \( T_1 \)
- \( T_2 \)

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

**WRC+lwsync+addr**

**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 **IRIW+addrs**:

<table>
<thead>
<tr>
<th></th>
<th>( T_0 )</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( T_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a)) ( x \leftarrow 1 )</td>
<td>((b) r_0 \leftarrow x )</td>
<td>((d) r_2 \leftarrow y )</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>((c) r_1 \leftarrow y )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

Observed? \( r_0=1; r_1=0; r_2=1; r_3=0 \);

- \( a: Wx=1 \)
- \( b: Rx=1 \)
- \( d: Wy=1 \)
- \( e: Ry=1 \)
- \( c: Rx=0 \)

**IRIW+addrs**

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 \( T_2 \):

- \( T_0 \)
- \( T_1 \)
- \( T_2 \)
- \( T_3 \)

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

**IRIW+syncs**

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

Communication relations:

▶ Read-from: \( w \xrightarrow{r} r \), with \( \text{loc}(w) = \text{loc}(r) \), \( \text{val}(w) = \text{val}(r) \).

▶ Coherence: \( w \xrightarrow{c} w' \), with \( \text{loc}(w) = \text{loc}(w') = x \). Total order for given \( x \): hence “coherence orders”.

▶ We deduce from-read: \( r \xrightarrow{fr} r' \) and \( w \xrightarrow{co} w' \).

▶ We distinguish internal (same proc, \( r_{fi} \rightarrow, c_{oi} \rightarrow, f_{ri} \rightarrow \)) and external (different procs, \( r_{fe} \rightarrow, c_{oe} \rightarrow, f_{re} \rightarrow \)) 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 \( \text{ppo} \rightarrow \subseteq \text{po} \rightarrow \). 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 \( \rightarrow^{\text{strong}} \) and \( \rightarrow^{\text{light}} \). Effective means that for instance \( w \xrightarrow{\text{lwsync}} r \) does not imply \( w \xrightarrow{\text{light}} r \).

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:

let propbase = \(((\text{W} \times \text{W}) & fence) | (rfe; ((\text{R} \times \text{W}) & fence)));hb^*\)
acyclic co | prop as observation

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

let prop = \((\text{W} \times \text{W}) \& \text{propbase}|(\text{com}^*;\text{propbase}^*;\text{strong};hb^*)\)
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 i0 = dd | rfi | rdw
let cc0 = dd | po-loc | ctrl | addrpo
let ic0 = 0

(* Fixpoint from i -> c in instructions and transitivity *)

let rec ci = ci0 | (ci;i1) | (cc;ci)
and i1 = i0 | i | (ic;ci) | (ii;i1)
and cc = cc0 | i | (ci;ic) | (cc;cc)
and ic = ic0 | ii | cc | (ic;cc) | (ii ; ic)

let ppo = RW(ic) | RR(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 (×1000 w.r.t. PLDI’11) (×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. ppo, or the semantics of weak fences).
Subtle point

**Z6.1**

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

Observed? $x=2; y=2; r_0=1$

Unobserved and forbidden by model. May be allowed.

---

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. $po \rightarrow$ contradicts $fr \rightarrow$).

**Notice:** CoRR is not observed directly.