Not so practical multicore programming

A simple model for sequential consistency, extended...

Luc Maranget
Luc.Maranget@inria.fr

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.
Events

The effect of "operations executed by the processors" are represented by events. More precisely, as we interleave memory accesses, we define memory events \(d[ℓ]v\) which consist in:

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

The program order \(\preceq\) is a linear order amongst the events originating from the same processor.

Relation \(\preceq\) 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 ;
for (int k = 0 ; k < 2 ; k++) { r1 = t[k] ; r2 += r1 ; }      
y = r2 ;
```

Events and program order:

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

A definition of SC

Definition (SC 1)

An execution is SC when there exists a total order on events \(\prec\), such that:

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

2. Reads read from the closest write upwards:
   \[ \text{Def } \quad (w, r) | w = \max\{w', \text{loc}(w') = \text{loc}(r) \land w' < r\}. \]

Example of a question on SC

Program:

<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)</td>
</tr>
<tr>
<td>(b)</td>
<td>(y \leftarrow 1)</td>
<td>(d)</td>
</tr>
</tbody>
</table>

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

How do we know? Let us enumerate all interleaveings and observe if \(b < c\) then \(y=2\), if \(a < d\) then \(r0=1\).

<table>
<thead>
<tr>
<th></th>
<th>(y=2; r0=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, b, c, d)</td>
<td>(y=2; r0=1)</td>
</tr>
<tr>
<td>(a, c, b, d)</td>
<td>(y=1; r0=1)</td>
</tr>
<tr>
<td>(a, c, d, b)</td>
<td>(y=1; r0=1)</td>
</tr>
<tr>
<td>(c, d, a, b)</td>
<td>(y=1; r0=0)</td>
</tr>
<tr>
<td>(c, a, b, d)</td>
<td>(y=1; r0=1)</td>
</tr>
<tr>
<td>(c, a, d, b)</td>
<td>(y=1; r0=1)</td>
</tr>
</tbody>
</table>
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) ( x \leftarrow 1 )</td>
<td>(c) ( y \leftarrow 2 )</td>
<td></td>
</tr>
<tr>
<td>(b) ( y \leftarrow 1 )</td>
<td>(d) ( r_0 \leftarrow x )</td>
<td></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 “\( y=2; r_0=0; \)”.

Example of \( \xrightarrow{rt} \)

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>(b) ( y \leftarrow 1 )</td>
<td>(d) ( x \leftarrow 1 )</td>
<td></td>
</tr>
</tbody>
</table>

Observe: \( r_0; r_1; \)

There are 4 possible \( \xrightarrow{rt} \) relations (initial value is 0).

<table>
<thead>
<tr>
<th></th>
<th>( r_0=1; r_1=1; )</th>
<th>( r_0=1; r_1=0; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( Rx=1 )</td>
<td>( \xrightarrow{rf} ) ( Ry=1 )</td>
<td>( \xrightarrow{po} ) ( Ry=1 )</td>
</tr>
<tr>
<td>b: ( Wy=1 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
</tr>
<tr>
<td>c: ( Ry=0 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
</tr>
<tr>
<td>d: ( Wx=1 )</td>
<td>( \xrightarrow{rf} ) ( Wx=1 )</td>
<td>( \xrightarrow{rf} ) ( Wx=1 )</td>
</tr>
</tbody>
</table>

Example of \( \xrightarrow{co} \)

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

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

<table>
<thead>
<tr>
<th></th>
<th>( x=1; y=2; )</th>
<th>( x=2; y=2; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( Wx=2 )</td>
<td>( \xrightarrow{co} ) ( Wx=2 )</td>
<td>( \xrightarrow{co} ) ( Wx=2 )</td>
</tr>
<tr>
<td>b: ( Wy=1 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
</tr>
<tr>
<td>c: ( Wy=2 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
<td>( \xrightarrow{po} ) ( Wy=1 )</td>
</tr>
<tr>
<td>d: ( Wx=1 )</td>
<td>( \xrightarrow{co} ) ( Wx=1 )</td>
<td>( \xrightarrow{co} ) ( Wx=1 )</td>
</tr>
</tbody>
</table>

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

Systematic approach

For a particular execution we assume two relations:

- **Read-from** (\( \xrightarrow{rt} \)): Relates write events to read events that read the stored value (initial writes left implicit in diagrams).

  \[ \forall r, \exists! w, w \xrightarrow{rt} r \]

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

- **Coherence** (\( \xrightarrow{co} \)): Relates write events to the same location.

  For any location \( \ell \), the restriction of \( \xrightarrow{co} \) to write events to location \( \ell \) (\( W_\ell \)) is a total order.

  *(Notice: To me the very existence of \( \xrightarrow{co} \) is implied by the existence of a shared, coherent, memory — Given location \( \ell \), there is exactly one memory cell whose location is \( \ell \).)*
One more relation: $\rightarrow_{fr}$

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

$\rightarrow_{fr}$

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

<table>
<thead>
<tr>
<th>MP</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$(a) x \leftarrow 1$</td>
<td>$(a) x \leftarrow 1$</td>
</tr>
<tr>
<td>$(b) y \leftarrow 1$</td>
<td>$(b) y \leftarrow 1$</td>
</tr>
<tr>
<td>$(c) r0 \leftarrow y$</td>
<td>$(c) r0 \leftarrow y$</td>
</tr>
<tr>
<td>$(d) r1 \leftarrow x$</td>
<td>$(d) r1 \leftarrow x$</td>
</tr>
<tr>
<td>Observed? $r0=1; r1=0$</td>
<td>Observed? $r0=0; r1=0$</td>
</tr>
</tbody>
</table>

Playing with $\rightarrow_{fr}$

Particular, easy, case: a read from the inital state is in $\rightarrow_{fr}$ with writes by the program.

Second definition of SC

Definition (SC 2)

An execution is SC when:

$$\text{Acyclic} \left( \rightarrow_{fr} \cup \rightarrow_{co} \cup \rightarrow_{fr} \cup \rightarrow_{po} \right)$$

And of course:

Theorem

The two definitions of SC are equivalent.

SC 1 $\implies$ SC 2

Assume the existence of the total order “<”.

Define:

$$\rightarrow_{co} = \{ (w_1, w_2) | \text{loc}(w_1) = \text{loc}(w_2) \land w_1 < w_2 \}$$

Notice that $\rightarrow_{fr}$ is already defined: $\rightarrow_{fr} = \rightarrow_{fr}$. Also notice $\rightarrow_{po} \subseteq <$, $\rightarrow_{co} \subseteq <$ and $\rightarrow_{fr} \subseteq <$.

Proof:

Define $\rightarrow_{fr} = \rightarrow_{fr}^{-1}$; $\rightarrow_{co}$, and prove $\rightarrow_{fr} \subseteq <$.

Let $r \rightarrow_{fr} w$. Let further $w_0 \rightarrow_{fr} r$, then, by definition of $\rightarrow_{fr}$, we have $w_0 \rightarrow_{co} w$ 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 $\rightarrow_{fr} \cup \rightarrow_{co} \cup \rightarrow_{fr} \cup \rightarrow_{po}$ would be a cycle in order “<”
SC 2 $\rightarrow$ SC 1

Since $\rightarrow\cup co\rightarrow\cup fr\rightarrow\cup po\rightarrow$ is a partial order, there exists a total order $<$ that "extends" it (no question on mathematical foundations, ...).

From $<$ define $\rightarrow<_c$:

$$\rightarrow<_c \overset{\text{Def}}{=} \left\{ (w, r) | w = \max(w', \text{loc}(w')) = \text{loc}(r) \wedge w' < r \right\}.$$  

and show $\rightarrow = \rightarrow<_c$.

- Let $w_0 \rightarrow r$ and let $w \in W$, $w \neq w_0$ then ($co$ 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 $w_0 \rightarrow<_c r$.

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

Introducing herd, a memory model simulator

A model sc.cat:

```
% cat sc.cat
"Sequential consistency"
let com = rf | co | fr
acyclic (po | com) as hb
```

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.

Simulating SC

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

SC 1 Enumerate interweavings.

SC 2 Enumerate axiomatic execution candidates (i.e. $po\rightarrow, rf\rightarrow, co\rightarrow$);
check the acyclicity of $\rightarrow\cup co\rightarrow\cup fr\rightarrow\cup po\rightarrow$.

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

Herd structure

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

  a: Wx=1  c: Wy=2  a: Wx=1  c: Wy=2
  po \rightarrow rf  po \rightarrow po
  b: Wy=1  d: Rx=1  b: Wy=1  d: Rx=1
  Ok  Ok

  No  Ok

- Apply model checks to each candidate execution.
Part 2.

Studying Non-Sequentially Consistent Executions.

Violations of SC

A cycle of $\xrightarrow{po}$, $\xrightarrow{rf}$, $\xrightarrow{co}$, $\xrightarrow{fr}$ describes a violation of SC.

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

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: $\xrightarrow{po}$ and $\xrightarrow{com}$ steps alternate

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

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

  $\xrightarrow{com} = \xrightarrow{rf} \cup \xrightarrow{co} \cup \xrightarrow{fr} \cup (\xrightarrow{co}; \xrightarrow{rf}) \cup (\xrightarrow{fr}; \xrightarrow{rf})$.

Because $(\xrightarrow{co}; \xrightarrow{co}) \subseteq \xrightarrow{co}$, $(\xrightarrow{fr}; \xrightarrow{co}) \subseteq \xrightarrow{fr}$, and $(\xrightarrow{fr}; \xrightarrow{fr}) \subseteq \xrightarrow{co}$.

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

Simplifying cycles: all $\xrightarrow{com}$ steps are external

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

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

- Either $e_1 \xrightarrow{po} e_2$ and we consider this $\xrightarrow{po}$ step in the cycle, in place of the $\xrightarrow{com}$ step (further merging $\xrightarrow{po}$ steps to get a smaller cycle).
- Or $e_2 \xrightarrow{po} e_1$, then we have a very simple cycle $e_2 \xrightarrow{po} e_1 \xrightarrow{com} e_2$. Such cycles are "violations of coherence" (more on them later).
- Case $e_1 = e_2$ is impossible ($\xrightarrow{com}$ is acyclic, see later)

**Notice:** A similar reasoning applies to individual $\xrightarrow{com}$ steps in composite $\xrightarrow{com}$. 
Simplifying cycles – Threads

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

\[ e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}; \text{po}} e_3 \xrightarrow{\text{po}} e_4 \xrightarrow{\text{com}; \text{po}} \ast \xrightarrow{\text{com}} e_1 \]

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

\[ e_1 \xrightarrow{\text{po}} e_3 \xrightarrow{\text{com}; \text{po}} e_4 \xrightarrow{\text{po}} e_1 \]

(Conclude with $\xrightarrow{\text{po}}$ being transitive)
If $e_1 = e_3$, we also have a simpler cycle:

\[ e_1 \xrightarrow{\text{po}} e_3 \xrightarrow{\text{com}; \text{po}} e_3 = e_1 \]

Conclusion: Pass through each thread only once.

Test from cycles — Threads

Cycle: $R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}}$

Consider a test execution on two threads:

The test execution features a smaller cycle:

\[ a: \text{Rx=1} \]
\[ b: \text{Wy=1} \]
\[ c: \text{Rz=1} \]
\[ d: \text{Wa=1} \]
\[ e: \text{Ry=1} \]
\[ f: \text{Wz=1} \]
\[ g: \text{Ra=1} \]
\[ h: \text{Wx=1} \]

Generally: one passage per thread

Test from cycles — Locations

Cycle: $R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} W \xrightarrow{\text{rf}}$

- One interpretation (four locations):

  \[ a: \text{Rx=1} \]
  \[ c: \text{Ry=1} \]
  \[ e: \text{Rz=1} \]
  \[ g: \text{Ra=1} \]
  \[ b: \text{Wy=1} \]
  \[ d: \text{Wz=1} \]
  \[ f: \text{Wa=1} \]
  \[ h: \text{Wx=1} \]

- Another interpretation (two locations):

  \[ a: \text{Rx=2} \]
  \[ c: \text{Ry=1} \]
  \[ e: \text{Rx=1} \]
  \[ g: \text{Ry=2} \]
  \[ b: \text{Wy=1} \]
  \[ d: \text{Wx=1} \]
  \[ f: \text{Wy=2} \]
  \[ h: \text{Wx=1} \]

The second interpretation is not “minimal”

Reminding the interpretation with two locations:

\[ a: \text{Rx=2} \]
\[ c: \text{Ry=1} \]
\[ e: \text{Rx=1} \]
\[ g: \text{Ry=2} \]
\[ b: \text{Wy=1} \]
\[ d: \text{Wx=1} \]
\[ f: \text{Wy=2} \]
\[ h: \text{Wx=2} \]

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

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

\[ a: \text{Rx=2} \]
\[ c: \text{Ry=1} \]
\[ e: \text{Rx=1} \]
\[ g: \text{Ry=2} \]
\[ b: \text{Wy=1} \]
\[ d: \text{Wx=1} \]
\[ f: \text{Wy=2} \]
\[ h: \text{Wx=2} \]

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

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

Generally: do not repeat locations in cycles.
Simplifying cycles, a lemma

**Lemma (Identical locations)**

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

- either $e_1 \xrightarrow{\text{com}} e_2$,
- or $e_2 \xrightarrow{\text{com}} e_1$,
- or $w \xrightarrow{r_f} e_1$ and $w \xrightarrow{r_f} e_2$.

**Case analysis:**

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

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

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 $w \xrightarrow{r_f} r_1$ and $w \xrightarrow{r_f} r_2$. We consider:

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

Then, in all possible three situations: $w_1 = w_2$, $w_1 \xrightarrow{\text{co}} w_2$ and $w_2 \xrightarrow{\text{co}} w_1$ 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 $e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}} e_1$.
- Or a critical cycle that is:
  - The cycle alternates $\xrightarrow{\text{po}}$ steps and external $\xrightarrow{\text{com}}$ steps.
  - The cycle passes through a given thread at most once.
  - All $\xrightarrow{\text{com}}$ steps have pairwise different locations.
  - The source and target of one given $\xrightarrow{\text{po}}$ 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:

- $po \xrightarrow{co} \xrightarrow{rf} \xrightarrow{fr} \xrightarrow{co} \xrightarrow{rf}$,
- $fr \xrightarrow{co} \xrightarrow{rf}$.

Application, all possible SC violations on two threads

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

- $2+2W$
- $LB$
- $MP$
- $R$
- $S$
- $SB$

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

Notice: coherence violations neglected.

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.

% diy -conf 2.conf
Generator produced 6 tests
% ls
2+2W.litmus 2.conf @all LB.litmus
MP.litmus R.litmus SB.litmus S.litmus
% diy -conf 4.conf
Generator produced 68 tests...
Three more violations of SC

<table>
<thead>
<tr>
<th></th>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x ← 1</td>
<td>c: Wy=2</td>
</tr>
<tr>
<td>b</td>
<td>y ← 1</td>
<td>po</td>
</tr>
<tr>
<td>d</td>
<td>r₀ ← x</td>
<td>po</td>
</tr>
</tbody>
</table>

Observed? y=2; r₀=0

<table>
<thead>
<tr>
<th></th>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x ← 2</td>
<td>c: Wy=1</td>
</tr>
<tr>
<td>b</td>
<td>y ← 1</td>
<td>po</td>
</tr>
<tr>
<td>d</td>
<td>r₁ ← x</td>
<td>po</td>
</tr>
</tbody>
</table>

Observed? x=2; r₁=0

A semi realistic example

```cpp
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.

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

A semi realistic example, more precise fencing

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

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

The resulting static \(\rightarrow_{po}\) relation is as follows.

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 \(\rightarrow_{po}\) 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 \(\rightarrow_{po}\) step of those.

Simplification:
- Insert fences between all pairs of racy accesses with different locations (notice that \(\rightarrow_{com}\) always includes a write).

Optimisation
- Forbid specific (critical) cycles by specific means (lightweight barriers, dependencies).

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

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

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

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

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

Cycle 4

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

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

X86: no fence needed.
Cycle 5

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

X86: no fence needed.

Sufficient fencing, X86

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

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

Cycle 6

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

X86: no fence needed.

Sufficient fencing, Power

for (int k = N ; k >= 0 ; k--) {
    a: x = k ;
    sync() ;
    b: go = 1 ;
    c: while (go == 1) ;
    lwsync() ;
    d: while (go == 0) ;
    sync() ;
    e: int t = x; sum += t;
    ctrlisync(t) ;
    f: go = 0 ;
}
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
        0:
        isync
        :: [t] "r" (t) : "memory") ;
}
Notice: Inserting full memory fence between racy accesses is much more simple.
```

TSO — The Model of X86 machines

![Diagram of TSO model]

The write buffer explains how "reads can pass over writes".

Part 3.

Axiomatic TSO

An experimental study of x86

Demo: (in demo/TSO1) Compiling:

```bash
% litmus -mach ./x86 ../diy/src2/@all -o run
% make -C run -j 4
```

Running:

```bash
% cd run
% sh run.sh > X.00
```

Analysis:

```bash
% 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

A: \text{Wx}=1 \quad \text{C}: \text{Wy}=2
\text{po} \quad \text{mb}
\text{b: Wy}=1 \quad \text{d: Rx}=0
\text{R: Ok}
\text{S: No}
\text{SB: Ok}

B: \text{Wx}=2 \quad \text{C}: \text{Ry}=1
\text{po} \quad \text{mb}
\text{b: Wy}=1 \quad \text{d: Wx}=1
\text{R: Ok}
\text{S: No}
\text{SB: Ok}

C: \text{Wx}=1 \quad \text{C}: \text{Wy}=1
\text{po} \quad \text{mb}
\text{b: Wy}=1 \quad \text{d: Wx}=1
\text{R: Ok}
\text{S: No}
\text{SB: Ok}

D: \text{Rx}=0
\text{co} \quad \text{mb}
\text{b: Wy}=1 \quad \text{d: Wx}=1
\text{R: Ok}
\text{S: No}
\text{SB: Ok}

E: \text{Wx}=1
\text{po} \quad \text{mb}
\text{b: Wy}=1 \quad \text{d: Wx}=1
\text{R: Ok}
\text{S: No}
\text{SB: Ok}

F: \text{Ry}=1
\text{fr} \quad \text{mb}
\text{b: Wy}=1 \quad \text{d: Wx}=1
\text{R: Ok}
\text{S: No}
\text{SB: Ok}

\text{MP: No}
\text{LB: No}
\text{2+2W: No}

\text{R+po+mfence}

<table>
<thead>
<tr>
<th>\text{T0}</th>
<th>\text{T1}</th>
<th>\text{Observed?}</th>
<th>\text{y}=2; \text{r0}=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{a: Wx}=1 \quad \text{c: Wy}=2 \quad \text{mfence}</td>
<td>\text{po} \quad \text{mb}</td>
<td>\text{b: Wy}=1 \quad \text{d: R}=0 \quad \text{No}</td>
<td></td>
</tr>
</tbody>
</table>

\text{SB+mfences}

<table>
<thead>
<tr>
<th>\text{T0}</th>
<th>\text{T1}</th>
<th>\text{Observed?}</th>
<th>\text{r0}=0; \text{r1}=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{a: Wx}=1 \quad \text{c: Wy}=1 \quad \text{mfence}</td>
<td>\text{po} \quad \text{mb}</td>
<td>\text{b: Ry}=0 \quad \text{d: R}=0 \quad \text{No}</td>
<td></td>
</tr>
</tbody>
</table>

Axiomatic TSO, model TSO 1

- Remember SC:

\[
\text{Acyclic} \left( \frac{\text{rf}}{\cup} \frac{\text{co}}{\cup} \frac{\text{fr}}{\cup} \frac{\text{po}}{\cup} \right)
\]

- A model for herd, our generic simulator:

\[
\text{let ppo} = \text{po} \# \text{ppo stands for 'preserved program-order'}
\]
\[
\text{let com-hb} = \text{fr} \cup \text{rf} \cup \text{co} \# \text{All communications create order}
\]
\[
\text{acyclic} (\text{ppo} \cup \text{com-hb})
\]

- In TSO:

  - Write-to-read does not create order:

\[
\text{let ppo} = \text{RR(po)} \cup \text{RW(po)} \cup \text{WW(po)} \# \text{WR(po) omitted from ppo}
\]
\[
\text{Communication creates order}
\]
\[
\text{let com-hb} = \text{fr} \cup \text{co} \cup \text{rf}
\]

- TSO "happens-before" (\text{hb}) check:

\[
\text{acyclic} (\text{ppo} \cup \text{com-hb} \cup \text{mfence}) \text{ as \text{hb}}
\]

\text{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.}

Our TSO 1 model is wrong!

Consider:

<table>
<thead>
<tr>
<th>\text{T0}</th>
<th>\text{T1}</th>
<th>\text{Observed?}</th>
<th>\text{r0}=1; \text{r1}=0; \text{r2}=1; \text{r3}=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{a: Wx}=1 \quad \text{d: Wy}=1 \quad \text{mfence}</td>
<td>\text{po} \quad \text{mb}</td>
<td>\text{b: R}=1 \quad \text{e: R}=1 \quad \text{f: R}=0 \quad \text{No}</td>
<td></td>
</tr>
</tbody>
</table>

According to model ? No. As we have the \text{hb} cycle:

\[
a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow a
\]

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.
Observation of **SB+rfi-pos**

Demo in demo/TSO2.

- Create test from cycle:
  ```
  % diyone -norm -arch X86 Rfi PodRR Fre Rfi PodRR Fre
  % ls
  SB+rfi-pos.litmus
  ```
- Run test:
  ```
  % litmus -mach x86.cfg src/SB+rfi-pos.litmus
  ```

---

**Corrected model: TSO 2**

Internal \( \overset{rf}{\rightarrow} (\overset{rfi}{\rightarrow}) \) does not create order, external \( \overset{rf}{\rightarrow} (\overset{rfe}{\rightarrow}) \) does:

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

The new hb is no longer cyclic:

![Diagram](https://via.placeholder.com/150)

(Also consider that \( a \overset{po}{\rightarrow} W c \) and \( d \overset{po}{\rightarrow} WR f \) are non-global.)

---

**A new check: UNIPROPROC**

We add a specific **UNIPROPROC** check to rule out coherence violations:

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

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

```text
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.

---

**We are not done yet...**

Our TSO 2 model:

```text
let ppo = RR(po) | RW(po) | WW(po) # WR(po) omitted
let com-hb = rfe | fr | co # rfi omitted
acyclic (ppo | com-hb | mfence) as hb
```

Allows two violations of coherence:

- CoRW1
- CoWR

Although TSO2 is not invalidated by hardware. Those "surprising" behaviours **must** be rejected by our TSO model.
Our final TSO model

Let \( \text{comhat} = r_f | f_r | c_0 | (c_0; r_f) | (f_r; r_f) \)
irreflexive \((p_0; \text{comhat})\) as uniproc

Let \( \text{ppo} = R(p_0) | R(w_0) | W(p_0) \) # \( W(p_0) \) omitted
Let \( \text{com-hb} = r_f e | f_r | c_0 \)
acyclic \( \text{ppo} | mfence | \text{com-hb} \) as \( \text{hb} \)

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

More precisely, we do not explain how \( \text{uniproc} \) is implemented. The “point in global time” interpretation for reads and writes does not apply.

---

Equivalence of uniproc definitions

Uniproc 1 \( \implies \) Uniproc 2 is obvious, as \((p_0; \text{com})\) is included in \((p_0 \cup \text{com})^+\) (since \( \text{com} = (\text{com})^+ \)).

Conversely, proof is easy from “Identical locations” lemma.

Consider a cycle in \((p_0 \cup \text{com})\), s.t. for all \( e_1 \xrightarrow{p_0} e_2 \) steps we do not have \( e_2 \xrightarrow{\text{com}} e_1 \). Then, for a given \( e_1 \xrightarrow{p_0} e_2 \) step:

- Either, \( r_1 \xrightarrow{p_0} r_2 \), with \( w \xrightarrow{r_f} r_1 \) and \( w \xrightarrow{r_f} r_2 \). We short-circuit the \( p_0 \) step, replacing \( w \xrightarrow{r_f} r_1 \xrightarrow{p_0} r_2 \) by \( w \xrightarrow{r_f} r_2 \).

- Or, \( e_1 \xrightarrow{\text{com}} e_2 \). We replace the \( p_0 \) 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 \( r \xrightarrow{fr*} w \). Additionally, \( r \) and \( w \) are tagged “atomic”.

Example: \( xchgl \ r, x \).

We further enforce:

- Writes \( w' \) to the location are either before the pair or after it:

\[
(r \xrightarrow{fr*} w) \implies (w' \xrightarrow{r \lor w' \xrightarrow{co} r \lor w \xrightarrow{co} w'})
\]

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

\[fr* \cap (fr; co) = \emptyset\]

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

The ATOM check forbids this execution:

<table>
<thead>
<tr>
<th>EXCH</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
<td></td>
</tr>
<tr>
<td>(a) $x \leftarrow 1$</td>
<td>r $\leftarrow 2$</td>
<td></td>
</tr>
<tr>
<td>(b/c) $r_1 \leftrightarrow x$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed? $r=0$; $y=2$

<table>
<thead>
<tr>
<th>Implied fences</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R+p_0+p_{0a}$</td>
</tr>
</tbody>
</table>

<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/d) $r \leftrightarrow y$</td>
</tr>
<tr>
<td>(b) $y \leftarrow 1$</td>
<td>(e) $r_0 \leftarrow x$</td>
</tr>
</tbody>
</table>

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

\[ a: Wx=1 \quad b: Rx*=0 \quad c: Wx*=2 \]

\[ a: Wx=1 \quad b: Wy=1 \quad c: Ry*=1 \quad d: Wx*=2 \quad e: Rx=0 \]

**Cycle:** $a \xrightarrow{p_0} b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$.

---

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 atom & (fre;coe) as atom

(* Implied fences (restricted to WR pairs) *)

let poWR = WR(po)
let implied = PA(poWR)|AP(poWR) #A is atomic, P is plain

(* Happens-before *)

let ppo = RR(po) | RW(po) | WW(po) #WR(po) omitted
let com-hb = rfe | fr | co

acyclic ppo | mfence | implied | com-hb as hb

---

Part 4.

Axiomatic ARM/Power
A relaxed shared memory computer

More or less visible to user code:
- Cores:
  - Out of order execution
  - Branch speculation
  - Write buffers
- Memory
  - Physically distributed
  - Caches

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, $\rightarrow$ 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:

```bash
$ 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 3571 1996429
Observation 2+2 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 $\rightarrow$ as simply as for TSO.
An experiment on ARM/Power

Consider test MP:

| MP |  
|---|---|
| $T_0$ | $T_1$ |
| $(a) x \leftarrow 1$ | $(c) r_0 \leftarrow y$ |
| $(b) y \leftarrow 1$ | $(d) r_1 \leftarrow x$ |

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\n   PodWW,DMBdWW,DSBdWW,ISBdWW\n   Rfe\n   PodRR,DpCtrl1dR,DpCtrl1sbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR\n   Fre
```

Generator produced 28 tests

- One generates MP as diyone PodWW Rfe PodRR Fre
- diycross $r_1^1, ..., r_N^1, ..., r_M^M$ generates the $N_1 \times \cdots \times N_M$ cycles $r_1^1 \cdots r_{k_1}^1 \cdots r_k^1 \cdots r_{k_M}^M$ by cross-producting the given edge list arguments.

This generates some variations in the MP family.

We then compile and run, and...

### Dependencies (Power)

Address dependency:

| $r_1 \leftarrow x$ | lwz r1,0(r8) # r8 contains the address of 'x' |
| $r_2 \leftarrow t[r1]$ | slwi r7,r1,2 # sizeof(int) = 4 |
| $y \leftarrow r_1+1$ | lwzx r2,r7,r9 # r9 contains the address of 't' |

Data dependency:

| $r_1 \leftarrow x$ | lwz r1,0(r8) # r8 contains the address of 'x' |
| $y \leftarrow r_1+1$ | addi r2,r1,1 |
| $stw r2,0(r9) # r9 contains the address of 'y'$ |

Control dependency: (+isync)

| $r_1 \leftarrow x$ | lwz r1,0(r8) |
| if $r_1=0$ then | cmpwi r1,0 |
| (isync) | bne L1 |
| $li r2,1$ | (isync) |
| $stw r2,0(r9)$ | li r2,1 |
| L1:

### Optimal fencing/dependencies for MP

This diagram illustrates the optimal fencing/dependencies for MP, showing the various combinations of fences and dependencies that can be used to ensure the test is observed or unobserved.

This diagram demonstrates the flexibility and complexity of managing fences and dependencies in ARM/Power environments to achieve desired test outcomes.
Optimal fencing for the 6 two-threads tests (Power)

R+syncs

a: \( W_x = 1 \)

b: \( W_y = 1 \)

c: \( R_y = 0 \)

d: \( R_x = 0 \)

S+\text{lwsync}+\text{addr}

a: \( W_x = 2 \)

b: \( W_y = 1 \)

c: \( R_y = 1 \)

d: \( R_x = 0 \)

SB+syncs

a: \( R_x = 1 \)

b: \( W_y = 1 \)

c: \( R_y = 1 \)

d: \( W_x = 1 \)

MP+\text{lwsync}+\text{addr}

a: \( W_x = 1 \)

b: \( W_y = 1 \)

c: \( R_y = 1 \)

d: \( R_x = 0 \)

LB+\text{addr}

a: \( R_x = 1 \)

b: \( W_y = 1 \)

c: \( R_y = 1 \)

d: \( W_x = 1 \)

2+2W+\text{lwsyncs}

a: \( W_x = 1 \)

b: \( W_y = 1 \)

c: \( W_y = 2 \)

d: \( R_x = 0 \)

Dependencies are enough

<table>
<thead>
<tr>
<th>CAUSAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_0 )</td>
</tr>
<tr>
<td>( (a) r_0 \leftarrow x )</td>
</tr>
<tr>
<td>( (b) y \leftarrow r_0 )</td>
</tr>
</tbody>
</table>

Observed? \( r_0 = 42; \ r_1 = 42 \):

a: \( R_x = 42 \)

b: \( W_y = 42 \)

c: \( R_y = 42 \)

d: \( W_x = 42 \)

LB+\text{datas}

Some observations

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

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

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: \( 2+2W+\text{lwsyncs} \) vs. \( R+\text{lwsync}+\text{sync} \).

2+2W+\text{lwsyncs}

\( a: W_x = 2 \)

\( b: W_y = 1 \)

\( c: W_y = 2 \)

\( d: W_x = 1 \)

R+\text{lwsync}+\text{sync}

\( a: W_x = 1 \)

\( b: W_y = 1 \)

\( c: W_y = 2 \)

\( d: R_x = 0 \)

No

Ok

Dependencies from reads not always enough!

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

\[ \begin{array}{cccc}
\hline
& \text{WRC} & & \\
\hline
\text{T}_0 & \text{T}_1 & \text{T}_2 \\
\hline
(a) x \leftarrow 1 & (b) r_0 \leftarrow x & (d) r_1 \leftarrow y \\
(c) y \leftarrow 1 & (e) r_1 \leftarrow x \\
\hline
\end{array} \]

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

a: \( W_x = 1 \)

b: \( R_x = 1 \)

c: \( W_y = 1 \)

d: \( R_y = 1 \)

e: \( R_x = 0 \)

WRC+\text{data+addr}

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

\text{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.

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 T₁:

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Wx=1</td>
<td>b: Rx=1</td>
<td>d: Ry=1</td>
</tr>
<tr>
<td>c: Wy=1</td>
<td>e: Rx=0</td>
<td></td>
</tr>
</tbody>
</table>

WRC+lwsync+addr

**Observation:** The fence orders the writes a (by T₀) and c (by T₁) for any observer (here T₂).

---

Restoring SC for **IRIW**

Use a full fence on T₁ and T₂:

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Wx=1</td>
<td>b: Rx=1</td>
<td>d: Wy=1</td>
<td>e: Ry=1</td>
</tr>
<tr>
<td>c: Ry=0</td>
<td>f: Rx=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IRIW+syncs

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

---

Another case of unsufficient dependencies

Consider test **IRIW+addrs**:

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) x ← 1</td>
<td>(b) r₀ ← x</td>
<td>(d) y ← 1</td>
<td>(e) r₂ ← y</td>
</tr>
<tr>
<td>(c) r₁ ← y</td>
<td></td>
<td>(f) r₃ ← x</td>
<td></td>
</tr>
</tbody>
</table>

**Observation:** The fence orders the writes a (by T₀) and c (by T₁) for any observer (here T₂).

**Stores are not “multi-copy atomic”:** T₀ and T₁ have a private buffer/cache/memory, T₂ and T₃ also have one.

---

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 $\text{loc}(w) = \text{loc}(w') = x$. 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, $\xrightarrow{rfi}, \xrightarrow{coi}$) and external (different proses, $\xrightarrow{rfe}, \xrightarrow{coe}$) communications.**

**“Execution” relations**

- **Program order:** $e₁ \xrightarrow{po} e₂$, with $\text{proc}(e₁) = \text{proc}(e₂)$.
- **Same location program order:** $e₁ \xrightarrow{po-loc} e₂$.
- **Preserved program order:** $e₁ \xrightarrow{ppq} e₂$, with $\text{ppq} \subseteq \text{po}$. 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 $\xrightarrow{strong}$ and $\xrightarrow{light}$. Effective means that for instance $w \xrightarrow{lwsync} r$ 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:

let propbase = (WW(fence)|(rfe;RW(fence)));hb*
irreflexive fre;propbase as observation

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

let prop = WW(propbase)|(com*;propbase*;strong;hb*)
acyclic co | prop as propagation

---

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. $\text{ppo} \rightarrow$, or the semantics of weak fences).

---

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) | RR(ii)
Can be limited to dependencies...

---

Subtle point

<table>
<thead>
<tr>
<th>$Z_{6.1}$</th>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a)x \leftarrow 2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(b)y \leftarrow 1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(c)y \leftarrow 2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(d)r0 \leftarrow z$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(e)z \leftarrow 1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(f)x \leftarrow 1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed? $x=2; y=2; r0=1$

Unobserved and forbidden by model. May be allowed...
A test of coherence violation

Our setting also finds bugs...
The following execution:

```
a: Wz=1
b: Ry=0
d: Rz=1
e: Wz=2
c: Rz=2
```

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.