Not so practical multicore programming

A simple model for sequential consistency, extended...

Luc Maranget  Luc.Maranget@inria.fr

Breaking news

The class scheduled on Thursday Feb. 7 is cancelled.


(Tuesday Feb. 21 exercises, Tuesday Feb. 28 exam.)

Part 1.

Axiomatic Sequential Consistency

Shared memory computer

\begin{center}
\begin{tikzpicture}
  \node[draw] (thread1) {Thread$_1$};
  \node[draw] at (thread1.east) {Thread$_n$};
  \node[draw, below of=thread1] (shared) {Shared Memory};
  \draw[->] (thread1) -- (shared) node[midway, left] {W};
  \draw[->] (thread1) -- (shared) node[midway, right] {R};
  \draw[dashed, ->] (thread1) -- (shared) node[midway, below] {W};
  \draw[dashed, ->] (thread1) -- (shared) node[midway, above] {R};
  \node[above of=shared, xshift=-0.5cm] \ldots;
  \node[above of=shared, xshift=0.5cm] \ldots;
\end{tikzpicture}
\end{center}
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.

Example of program-order

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

Events and program order:

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

Events

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

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

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

Relation \(\xrightarrow{po}\) represents the sequential execution of events by one processor that follows the uniprocessor model: the usual processor execution model, where instruction are executed by following the order given in program.

A definition of SC

**Definition (SC 1)**

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

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

2. **Reads read from the closest write upwards:**
   
   \[ r \xrightarrow{rf} \overset{\text{Def}}{=} \left\{ (w, r) | w = \max(w', \text{loc}(w') = \text{loc}(r) \land w' < r) \right\}. \]
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) $x \leftarrow 1$</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? $y=2$; $r_0=0$

How do we know? Let us enumerate all interleavings and observe if $b < c$ then $y=2$, if $a < d$ then $r_0=1$.

- $a, b, c, d$: $y=2$; $r_0=0$
- $a, c, b, d$: $y=1$; $r_0=1$
- $c, d, a, b$: $y=1$; $r_0=0$
- $c, a, b, d$: $y=1$; $r_0=1$
- $c, a, d, b$: $y=1$; $r_0=1$

Systematic approach

For a particular execution we assume two relations:

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

  \[ \forall r, \exists! w, w \rightarrow_{rf} r \]

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

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

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

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

Let us be a bit more clever

Collecting constraints on the scheduling order $\prec$:

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 $\prec$, 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 $\rightarrow_{rf}$

<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: $y=2$; $r_0=0$

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

- $r_0=1$; $r_1=1$
- $r_0=0$; $r_1=0$

Example of $\rightarrow_{rb}$

<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: $r_0=1$; $r_1=1$
Example of $\mathbf{\rightarrow_{co}}$

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

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

$x=1$; $y=2$;

a: $Wx=2$  c: $Wy=2$

b: $Wy=1$  d: $Wx=1$

$x=2$; $y=1$;

a: $Wx=2$  c: $Wy=2$

b: $Wy=1$  d: $Wx=1$

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

Playing with $\mathbf{\rightarrow_{fr}}$

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

<table>
<thead>
<tr>
<th>MP</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
</tr>
<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>
<tr>
<td>Observed? $r_0=1$; $r_1=0$</td>
<td>Observed? $r_0=0$; $r_1=0$</td>
</tr>
</tbody>
</table>

Observed? $r_0=1$; $r_1=0$;

Observed? $r_0=0$; $r_1=0$;

One more relation: $\mathbf{\rightarrow_{fr}}$

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

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

We have

(Or: $\rightarrow_{fr} \overset{\text{Def}}{=} \left(\rightarrow_{rf}\right)^{-1} \cap \rightarrow_{co}$)

Second definition of SC

Definition (SC 2)

An execution is SC when:

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

And of course:

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

Assume the existence of the total order "<".

Define:
\[
\text{co} \overset{\text{Def}}{=} \{(w_1, w_2) | \text{loc}(w_1) = \text{loc}(w_2) \land w_1 < w_1\}.
\]

Notice that \( \text{rf} \) is already defined: \( \text{rf} \overset{\text{Def}}{=} \text{rf} \land \text{rc} \). Also notice \( \text{po} \subseteq < \), \( \text{co} \subseteq < \) and \( \text{rf} \subseteq < \).

**Proof:**

Define \( \text{fr}^{-1} \overset{\text{Def}}{=} \text{rf}^{-1} \) and prove \( \text{fr} \subseteq < \).

Let \( r \overset{\text{fr}}{\rightarrow} w \). Let further \( w_0 \overset{\text{fr}}{\rightarrow} r \), then, by definition of \( \text{fr}^{-1} \), we have \( w_0 \overset{\text{co}}{\rightarrow} w \) 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 \( \text{rf} \cup \text{co} \cup \text{fr} \cup \text{po} \) 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. \( \text{po} \), \( \text{fr} \), \( \text{co} \)); check the acyclicity of \( \text{rf} \cup \text{co} \cup \text{fr} \cup \text{po} \).

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

SC 2 \implies SC 1

Since \( \text{rf} \cup \text{co} \cup \text{fr} \cup \text{po} \) is a partial order < that "extends" it (no question on mathematical foundations, . . . ).

From < define \( \text{rf} \)

\[
\text{rf} \overset{\text{Def}}{=} \{(w, r) | w = \max(w', \text{loc}(w') = \text{loc}(r) \land w' < r)\}.
\]

and show \( \text{rf} = \text{rf}^{-1} \).

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

Finally \( w_0 \overset{\text{rf}}{\rightarrow} r \).

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

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 \( \land \) 1:EAX=0)
Observation R Never 0 3

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

- Generate all candidate executions, i.e. all possible $\text{po} \rightarrow$, $\text{fr} \rightarrow$ and $\text{co} \rightarrow$ ($\text{fr} \rightarrow$ deduced):
  
  \[
  \begin{align*}
  \text{a: } Wx &= 1, & \text{b: } Wy &= 1, & \text{c: } Wy &= 2, & \text{d: } Rx &= 1, & \text{Ok} \\
  \text{a: } Wx &= 1, & \text{b: } Wy &= 1, & \text{c: } Wy &= 2, & \text{d: } Rx &= 0, & \text{No}
  \end{align*}
  \]

- Apply model checks to each candidate execution.

Violations of SC

A cycle of $\text{po} \rightarrow$, $\text{fr} \rightarrow$, $\text{co} \rightarrow$, $\text{fr} \rightarrow$ describes a violation of SC. From such a cycle, one may easily generate programs that potentially violate SC, and run them on actual machines.

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

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

Part 2.

Studying Non-Sequentially Consistent Executions.

Simplifying cycles: $\text{po} \rightarrow$ and $\text{com} \rightarrow$ steps alternate

A cycle in $\text{com} \rightarrow \cup \text{po} \rightarrow$ is a cycle in $\left( \text{po} \rightarrow^+ ; \text{com} \rightarrow^+ \right)$ (group $\text{po} \rightarrow$ and $\text{com} \rightarrow$ steps together). Then:
- $\text{po} \rightarrow$ is transitive $\text{po} \rightarrow^+ \subseteq \text{po} \rightarrow$.
- $\text{com} \rightarrow^+$ is the union of the five following relations:
  
  $\text{com} \rightarrow = \text{rf} \rightarrow \cup \text{co} \rightarrow \cup \text{fr} \rightarrow \cup \left( \text{co} \rightarrow; \text{rf} \rightarrow \right) \cup \left( \text{fr} \rightarrow; \text{rf} \rightarrow \right)$.

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

**Conclusion:** Any cyclic $\text{com} \rightarrow \cup \text{po} \rightarrow$ includes a cycle in $\left( \text{po} \rightarrow; \text{com} \rightarrow \right)$ — i.e. that alternates $\text{po} \rightarrow$ steps and $\text{com} \rightarrow$ steps.
Simplifying cycles: all $\xrightarrow{\text{com}}$ steps are external

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

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

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

---

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: $Rx=1$ $\xrightarrow{\text{po}}$ b: $Wy=1$ $\xrightarrow{\text{rf}}$ c: $Rz=1$ $\xrightarrow{\text{po}}$ d: $Wa=1$ $\xrightarrow{\text{rf}}$
- e: $Ry=1$ $\xrightarrow{\text{po}}$ f: $Wz=1$ $\xrightarrow{\text{po}}$ g: $Ra=1$ $\xrightarrow{\text{rf}}$
- h: $Wx=1$

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: $Rx=1$ $\xrightarrow{\text{po}}$ b: $Wy=1$ $\xrightarrow{\text{po}}$ c: $Rz=1$ $\xrightarrow{\text{rf}}$ d: $Wa=1$ $\xrightarrow{\text{po}}$
- e: $Ry=1$ $\xrightarrow{\text{po}}$ f: $Wz=1$ $\xrightarrow{\text{po}}$ g: $Ra=1$ $\xrightarrow{\text{rf}}$
- h: $Wx=1$

Another interpretation (two locations):

- a: $Rx=2$ $\xrightarrow{\text{po}}$ b: $Wy=1$ $\xrightarrow{\text{po}}$
- c: $Ry=1$ $\xrightarrow{\text{po}}$ d: $Wx=1$ $\xrightarrow{\text{po}}$
- e: $Rx=1$ $\xrightarrow{\text{po}}$ f: $Wy=2$ $\xrightarrow{\text{po}}$
- g: $Ry=2$ $\xrightarrow{\text{po}}$
- h: $Wx=2$

---

Simplifying cycles — Threads

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

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

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

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

Conclusion: Pass through each thread only once.
The second interpretation is not “minimal”

Reminding the interpretation with two locations:

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

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

Let us choose: Wx1 $\xrightarrow{\text{co}}$ Wx2:

- a: Rx=2
- c: Ry=1
- e: Rx=1
- g: Ry=2
- b: Wy=1
- d: Wx=1
- f: Wy=2
- h: 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 Wx2 $\xrightarrow{\text{co}}$ 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 $\xrightarrow{\text{com}}$ 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 $\xrightarrow{\text{po}}$ and one $\xrightarrow{\text{com}}$) in both directions. By the identical locations lemma:
  - Either, $e_1 \xrightarrow{\text{com}} e_2$ or $e_2 \xrightarrow{\text{com}} e_1$, and we have a simpler cycle.
  - or, $w \xrightarrow{\text{rf}} e_1$ and $w \xrightarrow{\text{rf}} e_2$, — see next page.

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

Next page

Simplifying cycles, a lemma

Lemma (Identical locations)

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

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

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 $w_1 \xrightarrow{\text{rf}} r_1$ and $w_2 \xrightarrow{\text{rf}} r_2$. 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 $r_1 \xrightarrow{\text{rf}} w_2 \xrightarrow{\text{rf}} r_2$.
- $r_1, r_2$, let $w_1 \xrightarrow{\text{rf}} r_1$. Then, either $w_1 = w_2$ and $w_2 \xrightarrow{\text{rf}} r_1$; or $w_1 \xrightarrow{\text{co}} w_2$ and $r_1 \xrightarrow{\text{rf}} w_2$; or $w_2 \xrightarrow{\text{co}} w_1$ and $w_2 \xrightarrow{\text{co}} r_1$.

Corollary: $\xrightarrow{\text{com}}$ is acyclic.
Application, all possible SC violations on two threads

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

- **2+2W**: \( \text{po} \rightarrow \text{co} \rightarrow \text{po} \rightarrow \text{co} \)
- **LB**: \( \text{po} \rightarrow \text{rf} \rightarrow \text{po} \rightarrow \text{fr} \)
- **MP**: \( \text{po} \rightarrow \text{fr} \rightarrow \text{po} \rightarrow \text{fr} \)
- **R**: \( \text{po} \rightarrow \text{co} \rightarrow \text{po} \rightarrow \text{fr} \)
- **S**: \( \text{po} \rightarrow \text{rf} \rightarrow \text{po} \rightarrow \text{co} \)
- **SB**: \( \text{po} \rightarrow \text{fr} \rightarrow \text{po} \rightarrow \text{fr} \)

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

**Notice**: coherence violations neglected.

Violations of coherence

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

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

### Example executions:

- **CoWW**: a: \( Wx=1 \) b: \( Wx=2 \) c: \( Wx=1 \) d: \( Wx=2 \)
- **CoRW1**: a: \( Wx=1 \) b: \( Wx=1 \) c: \( Wx=1 \) d: \( Wx=0 \)
- **CoRW**: a: \( Wx=1 \) b: \( Wx=1 \) c: \( Wx=1 \) d: \( Wx=0 \)
- **CoRW2**: a: \( Wx=2 \) b: \( Wx=1 \) c: \( Wx=2 \) d: \( Wx=1 \)
- **CoRR**: a: \( Rx=1 \) b: \( Rx=0 \) c: \( Rx=2 \) d: \( Rx=1 \)

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...
```
Three violations of SC

<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>(x \leftarrow 1)</td>
</tr>
<tr>
<td>(b)</td>
<td>(y \leftarrow 1)</td>
<td>(x \leftarrow 1)</td>
</tr>
<tr>
<td>(c)</td>
<td>(y \leftarrow 2)</td>
<td>(c)</td>
</tr>
<tr>
<td>(d)</td>
<td>(x \leftarrow 1)</td>
<td>(d)</td>
</tr>
</tbody>
</table>

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

\(2+2W\)

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

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

LB

\(T_0\) | \(T_1\)
---|---
\(a\) | \(x \leftarrow 1\) | \(c\) |
\(b\) | \(y \leftarrow 1\) | \(d\) |
\(c\) | \(x \leftarrow 1\) | \(d\)
\(d\) | \(y \leftarrow 1\) | \(d\)

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

MP

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 \(p \rightarrow o\) 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 \(p \rightarrow o\) step of those.

Simplification:

Insert fences between all pairs of racy accesses with different locations (notice that \(\hat{c}om\) always includes a write).

Optimisation

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

Three more violations of SC

<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>
<tr>
<td>(c)</td>
<td>(y \leftarrow 2)</td>
<td>(d)</td>
</tr>
</tbody>
</table>
| \(d\) | \(x \leftarrow 1\) | \(d\)

Observed? \(y=2; r_0=0\);

R

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

Observed? \(x=2; r_0=1\);

S

\(T_0\) | \(T_1\)
---|---
\(a\) | \(x \leftarrow 2\) | \(c\) |
\(b\) | \(y \leftarrow 1\) | \(d\) |
\(c\) | \(r_0 \leftarrow y\) | \(d\)
\(d\) | \(r_1 \leftarrow x\) | \(d\)

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

SB

A semi realistic example

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

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

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

```c
for (int k = N ; k >= 0 ; k--) {
    a: x = k ;
    sync() ;
    b: go = 1 ;
    c: while (go == 1) ;
    sync() ;
    d: while (go == 0) ;
    sync() ;
    e: int t = x; sum += t; sync() ;
    f: go = 0 ;
}
```
A semi realistic example, more precise fencing

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

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

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

\[
\begin{array}{ll}
a: W[x]=v & d: R[go]=1 \\
b: W[go]=1 & e: R[x]=w \\
c: R[go]=0 & f: W[go]=0 \\
\end{array}
\]

Cycle 1

Analysis based upon Sekar et al. Power model (PLDI'11). Test MP $a \xrightarrow{\text{lwasync}} b$, $d \xrightarrow{\text{ctrlisync}} e$

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

X86: $a \xrightarrow{\text{mfence}} c$, $f \xrightarrow{\text{mfence}} e$
Analysis based upon Sekar et al. Power model (PLDI’11). Test MP

\begin{equation}
\begin{align*}
&b \xrightarrow{\text{lw} syn c} a, \\
&e \xrightarrow{\text{ct} r\text{ls}yn c} d
\end{align*}
\end{equation}

X86: no fence needed.

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

\begin{equation}
\begin{align*}
c \xrightarrow{\text{ct} r\text{l}} a, \\
&f \xrightarrow{\text{ct} r\text{l}} f
\end{align*}
\end{equation}

X86: no fence needed.

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

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

Notice: Inserting full memory fence between racy writes gives the same result.
Part 3.

Axiomatic TSO

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

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

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

The write buffer explains how “reads can pass over writes”.

TSO — The Model of X86 machines

Inline assembler for fences and ctrlisync

```
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\n\t"
                     "beq.0f\n\t"
                     "0:\n\t"
                     "isync\n\t"
                     :: [t] "r" (t) : "memory");
}
```
An experimental study of x86

Demo: (in demo/TSO1)

Compiling:

```sh
% litmus7 -mach ./x86 ../diy/src2/@all -o run
% make -C run -j 4
```

Running:

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

Analysis:

```sh
% 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

- **R**: Ok
- **MP**: No
- **LB**: No
- **2+2W**: No

Axiomatic TSO, model TSO 1

- **Remember SC:**
  
  A model for herd, our generic simulator:
  
  ```
  \text{Acyclic} \left( \text{rf} \cup \text{co} \cup \text{fr} \cup \text{po} \right)
  ```

  A model for herd, our generic simulator:

  ```
  \text{let } \text{ppo} = \text{po} \# \text{ppo stands for }'\text{preserved program-order}'
  \text{let } \text{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:**
  
  ```
  \text{Write-to-read does not create order:}
  \text{let } \text{ppo} = (R*M \cup W*W) \& \text{po} \# \text{W*R pairs omitted}
  \text{Communication create order}
  \text{let } \text{com-hb} = \text{rf} \cup \text{co} \cup \text{fr}
  ```

- **TSO “happens-before” (HB) check:**

  ```
  \text{acyclic} (\text{ppo} \cup \text{com-hb} \cup \text{mfence}) \text{ as } \text{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.
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>$(d) y \leftarrow 1$</td>
</tr>
<tr>
<td>(b)</td>
<td>$r_0 \leftarrow x$</td>
<td>$(e) r_2 \leftarrow y$</td>
</tr>
<tr>
<td>(c)</td>
<td>$r_1 \leftarrow y$</td>
<td>$(f) r_3 \leftarrow x$</td>
</tr>
</tbody>
</table>

Observed? $r_0=1; r_1=0; r_2=1; r_3=0$;

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

$$\text{let com-hb} = rfe \ | \ fr \ | \ co \ # \ rfi \ not \ in \ hb$$
$$\text{acyclic} \ ppo \ | \ com-hb \ | \ mfence$$

The new hb is no longer cyclic:

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

Observation of \SB+rfi-pos\:

Demo in demo/TSO2.

- Create test from cycle:
  $$\% \text{ diyone7 -norm -arch X86 Rfi PodRR Fre Rfi PodRR Fre}$$
  $$\% \text{ is}$$
  $$\text{SB+rfi-pos.gitmus}$$

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

Test SB+rfi-pos Allowed

Histogram (4 states)

- exists (0:EAX=1 \land 0:EAX=0 \land 1:EAX=1 \land 1:EAX=0)

This is not over yet...

Our TSO 2 model:

$$\text{let ppo} = (R*M \ | \ W*W) \ # \ (W*R) \ & \ \text{ppo omitted}$$
$$\text{let com-hb} = rfe \ | \ fr \ | \ co \ # \ \text{rfi omitted}$$
$$\text{acyclic} \ (ppo \ | \ \text{com-hb} \ | \ \text{mfence})$$

as hb

Allows two violations of coherence:

$$\text{CoRW1}$$
$$\text{CoWR}$$

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:

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

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

\[
\text{let complus} = \text{rf} \mid \text{fr} \mid \text{co} \mid (\text{co};\text{rf}) \mid (\text{fr};\text{rf})
\]

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 \( \text{hb} \) check.

---

A word on **UNIPROC**


**Definition (Uniproc 1)**

\[
\text{Acyclic } \left( \text{po-loc} \xrightarrow{\cdot} \cup \text{com} \xrightarrow{\cdot} \right)
\]

with \( \text{com} \xrightarrow{\cdot} = \text{rf} \cup \text{co} \cup (\text{co};\text{rf}) \cup (\text{fr};\text{rf}) \)

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

**Definition (Uniproc 2)**

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

Definitions are equivalent.

---

Our final TSO model

**TS03**

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

irreflexive (po-loc; comhat) as uniproc

\[
\text{let ppo} = (\text{R*M} \mid \text{W*W}) \& \text{po} \# (\text{W*R}) \& \text{po omitted}
\]

\[
\text{let com-hb} = \text{rfe} \mid \text{fr} \mid \text{co} \# \text{rfi omitted}
\]

acyclic ppo \mid mfence \mid com-hb as hb

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

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

---

Equivalence of uniproc definitions

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

Conversely, we use the “Identical locations” lemma.

Consider a cycle in \( \text{po-loc} \xrightarrow{\cdot} \cup \text{com} \xrightarrow{\cdot} \), s.t. for all \( e_{1} \xrightarrow{\text{po}} e_{2} \) steps we do not have \( e_{2} \xrightarrow{\text{com}} e_{1} \). Then, for a given \( e_{1} \xrightarrow{\text{po}} e_{2} \) step:

- Either, \( r_{1} \xrightarrow{\text{po}} r_{2} \), with \( w \xrightarrow{\text{rf}} r_{1} \) and \( w \xrightarrow{\text{rf}} r_{2} \). We short-circuit the \( \text{po} \xrightarrow{\cdot} \) step, replacing \( w \xrightarrow{\text{rf}} r_{1} \xrightarrow{\text{po}} r_{2} \) by \( w \xrightarrow{\text{rf}} r_{1} \xrightarrow{\text{po}} r_{2} \).
- Or, \( e_{1} \xrightarrow{\text{com}} e_{2} \). We replace the \( \text{po} \xrightarrow{\cdot} \) step by \( \text{com} \xrightarrow{\cdot} \) steps.

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

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

We further enforce:

- Writes \( w' \) to the location are either before the pair or after it:
  \[
  \left( r \xrightarrow{rmw} w \right) \implies \left( w' \xrightarrow{fr} r \lor w' \xrightarrow{co} w \right)
  \]
  Or more concisely, we forbid \( r \xrightarrow{fr} w' \xrightarrow{co} w \), that is no \( w' \) in-between.

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

Example:

\[
\text{\texttt{xchgl} } r, x.
\]

Implied fences

**Implied fences forbid this execution**

\[
\begin{array}{ccc}
\text{SB+EXCH} & \text{T\textsubscript{0}} & \text{T\textsubscript{1}} \\
\hline
r \leftarrow 1 & r \leftarrow 1 & \\
(a/b) r \leftarrow x & (d/e) r \leftarrow y & \\
(c) r0 \leftarrow y & (f) r1 \leftarrow x & \\
\hline
\text{Observed?} & r0=0; r1=0
\end{array}
\]

\[
\text{Cycle: } b \xrightarrow{\text{implied}} c \xrightarrow{fr} e \xrightarrow{\text{implied}} f \xrightarrow{fr}.
\]

x86-TSO model for herd

"X86 TSO"

\[
\begin{align*}
\text{(* Uniproc *)} & \quad \text{let comhat = } rf | fr | co | (co;rf) | (fr;rf) \\
& \quad \text{irreflexive (po-loc; comhat) as uniproc} \\
\text{(* Atomic pairs *)} & \quad \text{let poWR = } (W*R) & po \\
& \quad \text{let implied = } (M*A | A*M) & poWR \\
\text{(* Happens-before *)} & \quad \text{let ppo = } (R*M | W*W) & po \\
& \quad \text{let com-hb = } rfe | fr | co \\
& \quad \text{acyclic ppo | mfence | implied | com-hb as hb}
\end{align*}
\]

ATOM check

The ATOM check forbids this execution:

\[
\begin{array}{ccc}
\text{EXCH} & \text{T\textsubscript{0}} & \text{T\textsubscript{1}} \\
\hline
(a) x \leftarrow 1 & r \leftarrow 2 & \\
(b/c) r1 \leftarrow x & \\
\hline
\text{Observed?} & r0=0; y=2
\end{array}
\]

\[
\begin{array}{c}
a: \text{Wx}=1 \xrightarrow{rmw} \\
b: \text{Rx}^* = 0 \xrightarrow{fr} \\
c: \text{Wx}^* = 2 \xrightarrow{co} \\
\end{array}
\]
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, \( \text{ppo} \) 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+2W Sometimes 7270 1992730
Observation SB Sometimes 9788 1990212
Observation LB Sometimes 4782 1995218

All Non-SC behaviours observed!

No hope to define \( ppo \) as simply as for TSO.

Dependencies (Power)

Address dependency:

\[
\begin{align*}
    r1 & \leftarrow x \\
    r2 & \leftarrow t[r1]
\end{align*}
\]

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

Data dependency:

\[
\begin{align*}
    r1 & \leftarrow x \\
    y & \leftarrow r1+1
\end{align*}
\]

```bash
lwz r1,0(r8)  # r8 contains the address of 'x'
addi r2,r1,1
stw r2,0(r9)  # r9 contains the address of 'y'
```

Control dependency: (+isync)

\[
\begin{align*}
    r1 & \leftarrow x \\
    \text{if } r1=0 \text{ then} \\
    & \quad \text{(isync)}
\end{align*}
\]

```bash
lwz r1,0(r8)  
cmpeq r1,0
bne L1
(risync)
li r2,1
stw r2,0(r9)
L1:
```

Generating tests (ARM), yet another tool: diycross

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

```bash
% diycross -arch ARM\nPodWW,DMBdWW,DSBdWW,ISBdWW\nRfe\nPodRR,DpCtrl1,bdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR\nFre
```

Generator produced 28 tests

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

This generates some variations in the MP family.

We then compile and run, and...

An experiment on ARM/Power

Consider test MP:

<table>
<thead>
<tr>
<th></th>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>( x \leftarrow 1 )</td>
<td>(c)</td>
</tr>
<tr>
<td>(b)</td>
<td>( y \leftarrow 1 )</td>
<td>(d)</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.
Optimal fencing/dependencies for MP

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

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:
  \[ 2+2W+lwsyncs \text{ vs. } R+lwsync+sync \]

No Ok

Dependencies are enough

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

Observer: \( r_0 = 42; \ r_1 = 42 \)

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 \textbf{WRC+data+addr}:

|\textbf{WRC} | \textbf{T}_0 & \textbf{T}_1 & \textbf{T}_2 |
|--------------|----------|----------|----------|
|\(a\) \(x \leftarrow 1\) & \((b) r_0 \leftarrow x\) & \((d) r_1 \leftarrow y\) & \((e) r_1 \leftarrow x\) |
|\textbf{Observed?} \(r_0=1; r_1=0;\) |
|a: \textbf{WRC+data+addr} | \(\text{Wx}=1\) & \(\text{Rx}=1\) & \(\text{Wy}=1\) & \(\text{Ry}=1\) & \(\text{Rx}=0\) |

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). \(T_2\) “\textit{does not see}” the store by \(T_0\), when \(T_1\) does.

Another case of unsufficient dependencies

Consider test \textbf{IRIW+addrs}:

|\textbf{IRIW} | \textbf{T}_0 & \textbf{T}_1 & \textbf{T}_2 & \textbf{T}_3 |
|--------------|----------|----------|----------|----------|
|\(a\) \(x \leftarrow 1\) & \((b) r_0 \leftarrow x\) & \((d) y \leftarrow 1\) & \((e) r_2 \leftarrow y\) & \((f) r_3 \leftarrow x\) |
|\textbf{Observed?} \(r_0=1; r_1=0; r_2=1; r_3=0;\) |
|a: \textbf{IRIW+addrs} | \(\text{Wx}=1\) & \(\text{Rx}=1\) & \(\text{Wy}=1\) & \(\text{Ry}=1\) & \(\text{Rx}=0\) |

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 \textbf{WRC}:

Use a lightweight fence on \(T_1\):

\(T_0\): \textbf{Wx}=1 \quad T_1: \textbf{Rx}=1 \quad T_2: \textbf{Ry}=1 \quad \textbf{WRC+lwsync+addr}\)

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

Restoring SC for \textbf{IRIW}:

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

\(T_0\): \textbf{Wx}=1 \quad T_1: \textbf{Rx}=1 \quad T_2: \textbf{Ry}=1 \quad T_3: \textbf{Rx}=0 \quad \textbf{IRIW+syncs}\)

\textbf{Propagation}: Full fences order all communications.
Relation summary

Communication relations:
- **Read-from:** \( w \xrightarrow{rf} r \), with \( \text{loc}(w) = \text{loc}(r), \text{val}(w) = \text{val}(r) \).
- **Coherence:** \( w \xrightarrow{co} w' \), with \( \text{loc}(w) = \text{loc}(w') \Rightarrow x \). 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, \( \xrightarrow{rfi}, \xrightarrow{coi}, \xrightarrow{fr} \)) and external (different proc, \( \xrightarrow{rfe}, \xrightarrow{coe}, \xrightarrow{fre} \)) communications.

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

ARM/Power preserved program order

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

\[
\begin{align*}
(* \text{ Utilities } *) & \\
\text{let } dd = \text{addr} | \text{data} & \quad \text{let } rduw = \text{po-loc} & \& (\text{fre};\text{rfe}) \\
\text{let detour} = \text{po-loc} & \& (\text{coe} ; \text{rfe}) & \text{let addrpo} = \text{addr};\text{po} \\
\end{align*}
\]

\[
\begin{align*}
(* \text{ Initial value } *) & \\
\text{let } ci0 = \text{ctrlisync} & \& \text{detour} & \text{let } i0 = dd & \& \text{rfi} & \& rduw \\
\text{let } cc0 = dd & \& \text{po-loc} & \& \text{ctrl} & \& \text{addrpo} & \text{let } ic0 = 0 \\
\end{align*}
\]

\[
\begin{align*}
(* \text{ Fixpoint from } i \rightarrow c \text{ in instructions and transitivity } *) & \\
\text{let rec } ci = ci0 & \& (ci;i) & \& (cc;ci) & \text{and } ii = i0 & \& ci & \& (ic;ci) & \& (ii;ii) \\
\text{and } cc = cc0 & \& ci & \& (ci;ic) & \& (cc;cc) & \text{and } ic = ic0 & \& ii & \& cc & \& (ic;cc) & \& (ii ; ic) \\
\text{let } ppo = \text{RW}(ic) & \& \text{RB}(ii) \\
\end{align*}
\]

Can be limited to dependencies...

A model in four checks (TOPLAS’14)

**UNIPROC**

acyclic poloc | com as uniproc

**No-Thin-Air**

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

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

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

irreflexive fre;propbase as observation

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

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

acyclic co | prop as propagation

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 lw-sync.

In any case:
- Simulation is fast \((\times 1000 \text{ w.r.t. } \text{PLDI’11}) \times 10 \text{ w.r.t. } \text{CAV’12})\).
- The existence of four checks \text{UNIPROC}, \text{HB OBSERVATION} and \text{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}, or the semantics of weak fences).
A test of coherence violation

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

![Diagram of a test of coherence violation]

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

Notice: CoRR is not observed directly.