Not so practical multicore programming

A simple model for sequential consistency, extended...

Luc Maranget
Luc.Maranget@inria.fr

Part 1.

Axiomatic Sequential Consistency

Shared memory computer

Thread\(_1\)  \ldots  \text{Thread}\(_n\)

\[
\begin{array}{c}
\text{W} \\
\text{R} \\
\text{W} \\
\text{R} \\
\text{W} \\
\end{array}
\begin{array}{c}
\text{W} \\
\text{R} \\
\text{W} \\
\text{R} \\
\text{W} \\
\end{array}
\]

Sequential consistency

**Original definition:** (Leslie Lamport)

[... \text{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”.

At 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 $(a) \cdot d[\ell] v$, 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: $T_0, T_1$ (usually omitted)

**Formalism: program order**

The program order $\rightarrow_{po}$ is a total strict order amongst the events originating from the same processor.

Relation $\rightarrow_{po}$ 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:

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

**A definition of SC**

A transcription of L. Lamport’s definition.

**Definition (SC 1)**

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

- Order $<$ is compatible with program order:

  \[ e_1 \rightarrow_{po} e_2 \implies e_1 < e_2. \]

- Reads read from the closest write upwards (a.k.a. “most recent”):

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

**Example of a question on SC**

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

Observed? $y=2, r_0=0$

How do we know? Let us enumerate all interleavings:

\[ a, b, c, d \quad y=2; r_0=1; \]
\[ a, c, b, d \quad y=1; r_0=1; \]
\[ a, c, d, b \quad y=1; r_0=1; \]
\[ c, d, a, b \quad y=1; r_0=0; \]
\[ c, a, b, d \quad y=1; r_0=1; \]
\[ c, a, d, b \quad y=1; r_0=1; \]

**Remark:** if $b < c$ then $y=2$, if $a < d$ then $r_0=1$. 
Let us be a bit more clever

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

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

Collecting constraints on the scheduling order $<$:

We respect program order, thus $a < b$, $c < d$.
We observe $r_0=0$, thus $d < a$, as $d$ reads initial value, which is overwritten by $a$.
We observe $y=2$, thus $b < c$.
Hence we have a cycle in $<$, which prevents it from being an order!

$$a < b < c < d < a$$

**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** ($\xrightarrow{rf}$): Relates write events to read events that read the stored value (initial writes left implicit in diagrams).
  $$\forall r, \exists! w, w \xrightarrow{rf} r$$
  *(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 strict order.

---

**Coherence as a characteristics of shared memory**

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

$$W_0 \xrightarrow{co} W_1 \xrightarrow{co} x = 2 \xrightarrow{co} W_3 \xrightarrow{co} \cdots$$

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

---

**Example of $\xrightarrow{rf}$**

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

Observe: $r_0; r_1$

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0=1$; $r_1=1$;</td>
<td>$r_0=1$; $r_1=0$;</td>
</tr>
</tbody>
</table>

- $a$: $R_x=1$
- $b$: $W_y=1$
- $c$: $R_y=1$
- $d$: $W_x=1$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0=0$; $r_1=1$;</td>
<td>$r_0=0$; $r_1=0$;</td>
</tr>
</tbody>
</table>

- $a$: $R_x=0$
- $b$: $W_y=1$
- $c$: $R_y=0$
- $d$: $W_x=1$
**Example of \( \rightarrow_{\mathcal{C}} \)**

<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>
<tr>
<td><strong>Observed?</strong></td>
<td>( x=2; \ y=2; )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( x=2; \ y=2; )</th>
<th>( x=1; \ y=1; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a: Wx=2 )</td>
<td>( a: Wx=2 )</td>
</tr>
<tr>
<td>( c: Wy=2 )</td>
<td>( c: Wy=2 )</td>
</tr>
<tr>
<td>( b: Wy=1 )</td>
<td>( a: Wx=2 )</td>
</tr>
<tr>
<td>( d: Wx=1 )</td>
<td>( c: Wy=2 )</td>
</tr>
<tr>
<td><strong>Observed?</strong></td>
<td>( r_0=1; \ r_1=0 )</td>
</tr>
</tbody>
</table>

\( x=2; \ y=1; \)

<table>
<thead>
<tr>
<th>( x=2; \ y=2; )</th>
<th>( x=2; \ y=1; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a: Wx=2 )</td>
<td>( a: Wx=2 )</td>
</tr>
<tr>
<td>( c: Wy=2 )</td>
<td>( c: Wy=2 )</td>
</tr>
<tr>
<td>( b: Wy=1 )</td>
<td>( a: Wx=2 )</td>
</tr>
<tr>
<td>( d: Wx=1 )</td>
<td>( c: Wy=2 )</td>
</tr>
<tr>
<td><strong>Observed?</strong></td>
<td>( r_0=0; \ r_1=0 )</td>
</tr>
</tbody>
</table>

---

**Playing with \( \rightarrow_{fr} \)**

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

**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>
<tr>
<td><strong>Observed?</strong></td>
<td>( r_0=1; \ r_1=0 )</td>
</tr>
</tbody>
</table>

**SB**

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

---

**Second definition of SC**

**Definition (SC 2)**

An execution is SC when:

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

And of course:

**Theorem**

The two definitions of SC are equivalent.
SC 1 $\Rightarrow$ 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 $\text{rf} \Rightarrow$ is already defined: $\text{rf} \overset{\text{Def}}{=} \text{rf} <$. Also notice $\text{po} \subseteq <$, $\text{co} \subseteq <$, and $\text{rf} \subseteq <$.

Proof:
Define $\text{fr} \overset{\text{Def}}{=} \text{rf}^{-1} \cap \text{co}$, and prove $\text{fr} \subseteq <$.

Let $\text{fr} \Rightarrow \text{w}$. Let further $\text{w}_0 \text{fr} < \Rightarrow \text{r}$, then, by definition of $\text{fr} \Rightarrow$, we have $\text{w}_0 \text{co} \Rightarrow \text{w}$ and thus $\text{w}_0 < \text{w}$.

But, $\text{w}_0$ is maximal amongst all $\text{w}' < \text{r}$. That is: “$\text{w} < \text{r} \Rightarrow \text{w} \leq \text{w}_0$” or, “$\text{w}_0 < \text{w} \Rightarrow \text{r} < \text{w}”$ QED.

Hence, a cycle in $\text{fr} \cup \text{co} \cup \text{fr} \cup \text{po}$ 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. $\text{po}$, $\text{rf}$, $\text{co}$); check the acyclicity of $\text{fr} \cup \text{co} \cap \text{fr} \cup \text{po}$.

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

Introducing herd, a memory model simulator

A model sc.cat:

```plaintext
% cat sc.cat
#include "cos.cat" #define co (and fr as "rf^-1; co")
let com = rf | co | fr #communication
acyclic po | com as hb #validity condition
```

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

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

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

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

- 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 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: $\xrightarrow{po}$ 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 \texttt{com} \rightarrow steps are external

Given a cycle, we consider that all \texttt{com} \rightarrow and \hat{\texttt{com}} \rightarrow steps are external, (i.e. source and target events are from pairwise distinct threads).

Given \( e_1 \hat{\texttt{com}} \rightarrow e_2 \), s.t. \( e_1 \) and \( e_2 \) are from the same thread:

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

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

Notice: A similar reasoning applies to individual \texttt{com} \rightarrow steps in composite \hat{\texttt{com}} \rightarrow.

Test from cycles — Threads

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

\[ e_1 \texttt{po} \rightarrow e_2 \hat{\texttt{com}} \rightarrow \texttt{po} \rightarrow e_3 \hat{\texttt{com}} \rightarrow \texttt{po} \rightarrow e_4 \hat{\texttt{com}} \rightarrow \texttt{po} \rightarrow e_1 \]

Assuming for instance, \( e_1 \texttt{po} \rightarrow e_3 \) then we have a "simpler" cycle:

\[ e_1 \texttt{po} \rightarrow e_3 \texttt{po} \rightarrow e_4 \hat{\texttt{com}} \rightarrow \texttt{po} \rightarrow e_1 \]

(Conclude with \texttt{po} \rightarrow being transitive)

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

\[ e_1 \texttt{po} \rightarrow e_2 \hat{\texttt{com}} \rightarrow \texttt{po} \rightarrow e_1 \]

Conclusion: Cycle visit a thread at most once.

Test from cycles — Locations

Consider a test execution on two threads:

The test execution features a smaller cycle

\[
\begin{align*}
\text{Cycle: } & R \texttt{po} \rightarrow W \texttt{rf} \rightarrow R \texttt{po} \rightarrow W \texttt{rf} \rightarrow R \texttt{po} \rightarrow W \texttt{rf} \\
& \text{Consider a test execution on two threads:} \\
& \text{The test execution features a smaller cycle}
\end{align*}
\]

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

Reminding the interpretation with two locations:

\[
\begin{align*}
a: R_x=2 & \quad \text{po} & \quad f: R_x=1 & \quad \text{po} & \quad g: R_y=2 & \quad \text{po} \\
b: W_y=1 & \quad \text{rf} & \quad d: W_x=1 & \quad \text{rf} & \quad e: R_x=1 & \quad \text{po} \\
c: R_y=1 & \quad \text{po} & \quad f: W_y=2 & \quad \text{po} & \quad g: R_y=2 & \quad \text{po} \\
d: W_x=1 & \quad \text{rf} & \quad e: R_x=1 & \quad \text{po} & \quad h: W_x=2 & \quad \text{rf}
\end{align*}
\]

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

Let us choose: \( W_x1 \xrightarrow{\text{co}} W_x2 \):

\[
\begin{align*}
a: R_x=2 & \quad \text{po} & \quad f: R_x=1 & \quad \text{po} & \quad g: R_y=2 & \quad \text{po} \\
b: W_y=1 & \quad \text{rf} & \quad d: W_x=1 & \quad \text{rf} & \quad e: R_x=1 & \quad \text{po} \\
c: R_y=1 & \quad \text{po} & \quad f: W_y=2 & \quad \text{po} & \quad g: R_y=2 & \quad \text{po} \\
d: W_x=1 & \quad \text{rf} & \quad e: R_x=1 & \quad \text{po} & \quad h: W_x=2 & \quad \text{rf}
\end{align*}
\]

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 \( W_x2 \xrightarrow{\text{co}} W_x1 \) 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 \( \text{comp} \) 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 \( \cdots w \xrightarrow{\text{rf}} e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{rf}} e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{rf}} \cdots \), which we reduce into \( \cdots w \xrightarrow{\text{rf}} e_2 \cdots \).

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 \( r_1 \xrightarrow{\text{rf}} w_2 \xrightarrow{\text{rf}} r_2 \).
- \( r_1, w_2 \), let \( w_1 \xrightarrow{\text{rf}} r_1 \) and \( w_2 \xrightarrow{\text{rf}} w_1 \); or \( w_1 \xrightarrow{\text{co}} w_2 \) and \( r_1 \xrightarrow{\text{rf}} w_2 \xrightarrow{\text{rf}} r_2 \).
- or \( w \xrightarrow{\text{rf}} e_1 \) and \( w \xrightarrow{\text{rf}} e_2 \).

**Corollary:** \( \xrightarrow{\text{comp}} \) 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 \( w \xrightarrow{\text{rf}} r_1 \) and \( w \xrightarrow{\text{rf}} r_2 \). We consider:

- If \( w \xrightarrow{\text{rf}} r_1 \) is in cycle, then there is an obvious short-circuit: replace \( \xrightarrow{\text{rf}} \) followed by the complex steps from \( r_1 \) to \( r_2 \) by a single \( w \xrightarrow{\text{rf}} r_2 \) step.
- If \( w \xrightarrow{\text{rf}} r_2 \) 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 \( \xrightarrow{\text{rf}} \) 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 $e_1 \xrightarrow{po} e_2 \xrightarrow{co} e_1$.
- Or a critical cycle that is:
  - The cycle alternates $\xrightarrow{po}$ steps and external $\xrightarrow{com}$ steps.
  - The cycle passes through a given thread at most once.
  - All $\xrightarrow{com}$ steps have pairwise different locations.
  - The source and target of one given $\xrightarrow{po}$ steps have different locations.

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

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

Violations of coherence

A violation of coherence is a cycle $e_1 \xrightarrow{po} e_2 \xrightarrow{co} e_1$.

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

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

Application, all possible SC violations on two threads

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

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

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

Notice: up to coherence violations (previous slide).

Generating two-threads SC violations

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

-arch X86 # 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>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) x ← 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) y ← 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed? x=2; y=2;

2+2W

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

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

Three more violations of SC

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

Observed? y=2; r₀=0

R

<table>
<thead>
<tr>
<th></th>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) x ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) y ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d) r₀ ← x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a: Wx=1   c: Wy=2
b: Wy=1   d: Rx=0

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: sum += x ;
    sync() ;
    f: go = 0 ;
}

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

Application

We assume the following on modern shared memory architectures:
- No valid execution includes a violation of coherence.
- No valid execution includes a cycle whose \( \xrightarrow{po} \) 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 \( \xrightarrow{po} \) step of those.

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

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

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

The resulting static \(\rightarrow^\text{no} \) relation is as follows.

- a: \(W[x]=v\) \(\rightarrow b\)
- b: \(W[go]=1\) \(\rightarrow c\)
- c: \(R[go]=0\) \(\rightarrow d\)
- d: \(R[go]=1\) \(\rightarrow e\)
- e: \(R[x]=w\) \(\rightarrow f\)
- f: \(W[go]=0\) \(\rightarrow a\)

**Cycle 1**

- a: \(W[x]=v\)
- b: \(W[go]=1\)
- c: \(R[go]=0\)
- d: \(R[go]=1\)
- e: \(R[x]=w\)
- f: \(W[go]=0\)

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

- \(a\) \(\xrightarrow{\text{lwsync}} b\), \(d\) \(\xrightarrow{\text{ctrlsync}} e\)

X86: no fence needed.

**Cycle 2**

- a: \(W[x]=v\)
- b: \(W[go]=1\)
- c: \(R[go]=0\)
- d: \(R[go]=1\)
- e: \(R[x]=w\)
- f: \(W[go]=0\)

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**

- a: \(W[x]=v\)
- b: \(W[go]=1\)
- c: \(R[go]=0\)
- d: \(R[go]=1\)
- e: \(R[x]=w\)
- f: \(W[go]=0\)

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

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

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

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

X86: no fence needed.

Cycle 5

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

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

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

X86: no fence needed.

Cycle 6

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

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

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

X86: no fence needed.

Sufficient fencing, X86

\[
\begin{align*}
\text{a: } & x = k ; \\
\text{b: } & \text{mfence}() ; \\
\text{c: } & \text{while} (go == 1) ; \\
\text{d: } & \text{sum} = 0 ; \\
\text{e: } & \text{for} (\text{int} k = N ; k >= 0 ; k--) \{ \text{int} t = x ; \text{sum} += t ; \text{mfence}() ; \}
\end{align*}
\]

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

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

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

### Inline assembler for fences and ctrlisync

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

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

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

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

### Part 3.

**Axiomatic TSO**

**TSO — The Model of X86 machines**

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

![Diagram of TSO model]
An experimental study of x86

**Demo:** (in demo/TSO1) Compiling:

```bash
% litmus7 -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 tests on this machine**

<table>
<thead>
<tr>
<th>Test</th>
<th>Observation R</th>
<th>Observation MP</th>
<th>Observation 2+2W</th>
<th>Observation S</th>
<th>Observation SB</th>
<th>Observation LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Ok</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Ok</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>C</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>F</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

---

**Axiomatic TSO, model TSO 1**

- **Remember SC:**

  ```latex
  \text{Acyclic}\left(\rightarrow_r \cup \rightarrow_{co} \cup \rightarrow_{fr} \cup \rightarrow_{po}\right)
  ```

  A model for herd, our generic simulator:

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

- **In TSO:**

  - Write-to-read does not create order:
    ```latex
    \text{let}\ ppo = (R*M} \cup W*W) \ & \ po \ # \ \text{All pairs except W*R pairs}
    \text{let}\ com-hb = rf \ | \ co \ | \ fr
    ```
  - Communication creates order
    ```latex
    \text{let}\ com-hb = rf \ | \ co \ | \ fr
    ```
  - TSO “happens-before” (HB) check:
    ```latex
    \text{acyclic (ppo} \cup \text{com-hb} \cup \text{mfence)} \ \text{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 HB) \(WR(\rightarrow_{po})\) by \(mfence\) (in HB).

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

**Observed? y=2; r0=0**

<table>
<thead>
<tr>
<th>(SB+mfences)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_0)</td>
</tr>
<tr>
<td>((a)\ x \leftarrow 1)</td>
</tr>
<tr>
<td>(mfence)</td>
</tr>
<tr>
<td>((b)\ r_0 \leftarrow y)</td>
</tr>
</tbody>
</table>

**Observed? r0=0; r1=0**

---

**Results for running the six tests on this machine**

<table>
<thead>
<tr>
<th>Test</th>
<th>Observation R</th>
<th>Observation MP</th>
<th>Observation 2+2W</th>
<th>Observation S</th>
<th>Observation SB</th>
<th>Observation LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Ok</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Ok</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>C</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>F</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

---

**Axiomatic TSO, model TSO 1**

- **Remember SC:**

  ```latex
  \text{Acyclic}\left(\rightarrow_r \cup \rightarrow_{co} \cup \rightarrow_{fr} \cup \rightarrow_{po}\right)
  ```

  A model for herd, our generic simulator:

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

- **In TSO:**

  - Write-to-read does not create order:
    ```latex
    \text{let}\ ppo = (R*M} \cup W*W) \ & \ po \ # \ \text{All pairs except W*R pairs}
    \text{let}\ com-hb = rf \ | \ co \ | \ fr
    ```
  - Communication creates order
    ```latex
    \text{let}\ com-hb = rf \ | \ co \ | \ fr
    ```
  - TSO “happens-before” (HB) check:
    ```latex
    \text{acyclic (ppo} \cup \text{com-hb} \cup \text{mfence)} \ \text{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.
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) $x \leftarrow 1$</td>
<td>(d) $y \leftarrow 1$</td>
<td></td>
</tr>
<tr>
<td>(b) $r_0 \leftarrow x$</td>
<td>(e) $r_2 \leftarrow y$</td>
<td></td>
</tr>
<tr>
<td>(c) $r_1 \leftarrow y$</td>
<td>(f) $r_3 \leftarrow x$</td>
<td></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:

\[
a \xrightarrow{rf} b \xrightarrow{po} c \xrightarrow{fr} d \xrightarrow{rfe} e \xrightarrow{po} f \xrightarrow{fr} a
\]

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 $\xrightarrow{rf}$ ($\xrightarrow{rfe}$) 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 $a \xrightarrow{po}WR c$ and $d \xrightarrow{po}WR f$ are non-global.)

Observation of SB+rfi-pos

Demo in demo/TSO2.

- Create test from cycle:
  \%
  % diyone7 -norm -arch X86 Rfi PodRR Fre Rfi PodRR Fre
  % is SB+rfi-pos.litmus
- Run test:

  \%
  % Results for src/SB+rfi-pos.litmus
  %
  1 litmus7 -mach x86.cfg src/SB+rfi-pos.litmus

This is not over yet...

Our TSO 2 model:

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

Allows two violations of coherence:

Although TSO2 is not invalidated by hardware. Those “surprising” behaviours must be rejected by our TSO model.
A new check: \texttt{UNIPROC}

We add a specific \texttt{UNIPROC} check to rule out coherence violations:

\[
\text{Irreflexive } \left( \frac{\text{po-loc} \to \, \text{com}}{\text{com} \to} \right)
\]

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

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

\[
\text{irreflexive } (\text{po-loc}; \text{complus}) \text{ as uniproc}
\]

In the TSO case we can “optimise”:

\[
\text{irreflexive rf;RW(po-loc)}
\]

\[
\text{irreflexive fr;WR(po-loc)}
\]

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

A word on \texttt{UNIPROC}


\begin{itemize}
  \item \textbf{Definition (Uniproc 1)}
    \[
    \text{Acyclic } \left( \frac{\text{po-loc} \cup \text{com}}{\text{com}} \right)
    \]
    with \(\text{com} = \text{rf} \cup \text{co} \cup \text{fr}\).
  \end{itemize}

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

\begin{itemize}
  \item \textbf{Definition (Uniproc 2)}
    \[
    \text{Irreflexive } \left( \frac{\text{po-loc}, \text{com}}{\text{com}} \right)
    \]
  \end{itemize}

Definitions are equivalent.

Our final TSO model

\[
\text{TS03}
\]

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

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

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

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

\[
\text{acyclic ppo | mfence | com-hb as hb}
\]

\textbf{Notice:} There are two checks... The axiomatic frameworks defines \textit{principles} that the operational model/hardware implement.

For instead, we do not explain how \texttt{UNIPROC} is implemented. Instead, we specify admissible behaviours.

Equivalence of uniproc definitions

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

Conversely, we use the “Identical locations” lemma.

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

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

As a result we have a cycle in \(\text{com}\), which is impossible.
From TSO to x86-TSO: locked instructions

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

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

We further enforce:

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

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

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

ATOM check

The \texttt{ATOM} check forbids this execution:

\[
\begin{array}{c|c}
T_0 & T_1 \\
\hline
(a)x \leftarrow 1 & r \leftarrow 2 \\
(b/c)r_1 \leftarrow x & \\
\end{array}
\]

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

Implied fences

Implied fences forbid this execution

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

Observed? \( r=0; r=1 \)

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

Cycle: \textit{b} \xrightarrow{implied} \textit{c} \xrightarrow{fr} \textit{e} \xrightarrow{implied} \textit{f} \xrightarrow{fr}.

x86-TSO model for herd

Predefined sets: \( W, R, M \) (any memory event), \( A \) (“atomic” memory event).

(* Uniproc *)

let \texttt{comhat} = rf | fr | co | (co;rf) | (fr;rf) \# or (rf|fr|co)+
irreflexive po; \texttt{comhat} as uniproc

(* Atomic pairs *)

empty \( rmw \) \& (fre;coe) as atom

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

let \texttt{pWR} = (W*R) \& po
let \texttt{implied} = (M*A | A*M) \& \texttt{pWR}

(* Happens-before *)

let \texttt{ppo} = (R*M | W*W) \& po \# W*R pairs omitted
let \texttt{com-hb} = rfe | fr | co \# rfi omitted

acyclic \texttt{ppo} | \texttt{mfence} | \texttt{implied} | \texttt{com-hb} as \texttt{hb}
Alternative formulation, or constrained domains and codomains

Given set $S$, $[S]$ is identity on $S$.
As a consequence, $[S_1] ; r ; [S_2]$ and $r & (S_1 * S_2)$ are equal.

Then, for instance, we may reformulate TSO preserved program order as:

\[
(* \text{let } \text{ppo} = (R\cdot M|W\cdot W) \& \text{po} *)
\]
\[
\text{let } \text{ppo} = [R];\text{po};[M] \mid [W];\text{po};[W]
\]

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

- 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: (…)
  - (TOPLAS’14) ARMv7 (ARM) and Power (PPC), more precise (proved to be more relaxed than PLDI’11, experimentally equivalent). A more simple axiomatic model.
  - ARMv8 (AArch64), official model, endorsed by ARM Ltd.

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, $\text{ppo} \rightarrow$ will now account for core constraints, such as dependencies.
- Communication relations alone do not define happen-before steps.
- A variety of memory fences: lightweight (Power $\text{lwsync}$) and full (Power $\text{sync}$).

Two-threads SC violation for ARM

Generating tests is as simple as:

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

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

- Observation R Sometimes 5722 1994278
- Observation MP Sometimes 17439 1982561
- Observation S Sometimes 7270 1992730
- Observation SB Sometimes 9788 1990212
- Observation LB Sometimes 4782 1995218

All Non-SC behaviours observed!

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

An experiment on ARM/Power

Consider test MP:

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

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

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

Dependencies (Power)

Address dependency:

- $r_1 \leftarrow x$
- $l\text{w}z \ r_1,0(r8)$ 
- $s\text{lwi} \ r_7,r_1,2$
- $l\text{w}z \ r_8\ r_7,r_9$

Data dependency:

- $r_1 \leftarrow x$
- $l\text{w}z \ r_1,0(r8)$
- $a\text{ddi} \ r_2,r_1,1$
- $s\text{tw} \ r_2,0(r9)$

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

- $l\text{w}z \ r_1,0(r8)$
- $c\text{mpwi} \ r_1,0$
- $b\text{ne} \ L_1$
- $l\text{i} \ r_2,1$
- $s\text{tw} \ r_2,0(r9)$
- $L_1$
Generating tests (ARM), yet another tool: diycross

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

% diycross -arch ARM
PodWW,DMBdWW,DSBdWW,ISBdWW\nRfe\nPodRR,DpCtrl1dR,DpCtrl1sbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR\nFre
Generator produced 28 tests

▶ One generates MP as diyone PodWW Rfe PodRR Fre
▶ diycross $r_1^k \ldots r_{N_1}^k \ldots r^M_{kM}$ generates the $N_1 \times \ldots \times N_M$ cycles $r_{k1}^1 \ldots r_{kM}^M$ by cross-producting the given edge list arguments.

This generates some variations in the MP family.

We then compile and run, and...

---

Optimal fencing/dependencies for MP

---

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

---

Some observations

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

$$addr < lwSsync < sync$$

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: $2 + 2W + lwSsyncs$ vs. $R + lwSsync + sync$.

---

No

Ok
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: $W_x=42$
- d: $R_y=42$

LB+datas

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>WRC+data+addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
</tr>
<tr>
<td>$(a) x[0] \leftarrow 1$</td>
</tr>
</tbody>
</table>
| | $(c) y \leftarrow r_0$ | $t \leftarrow r_1 \& 4$
| | | $(e) r_2 \leftarrow x[t]$ |

Observed? $r_0=1$; $r_1=0$
- a: $W_x[0]=1$
- b: $R_x[0]=1$
- c: $W_y[0]=1$
- d: $R_y[0]=1$
- e: $R_x[0]=0$

WRC+data+addr

Behaviour is legal on Power 6,7 (observed) and ARMv7 (non observed).

**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 $T_1$:

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
</table>
| a: $W_x=1$ | b: $R_x=1$ | d: $R_y=1$
| c: $W_y=1$ | e: $R_x=0$

WRC+lwsync+addr

**Observation**: The fence orders the writes $a$ (by $T_0$) and $c$ (by $T_1$) for any observer (here $T_2$). Similar to more simple **MP**

| a: $W_x=1$ | c: $R_y=1$
| b: $W_y=1$ | d: $R_x=0$

MP+lwsync+addr

---

**Another, symetric, case of insufficient dependencies**

Consider test **IRIW+addrs**:

<table>
<thead>
<tr>
<th>IRIW+addrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
</tr>
<tr>
<td>$(a) x[0] \leftarrow 1$</td>
</tr>
</tbody>
</table>
| | $t \leftarrow r_0 \& 0$ | | $t \leftarrow r_2 \& r_2$
| | $(c) r_1 \leftarrow y[t]$ | | $(f) r_3 \leftarrow x[t]$ |

Observed? $r_0=1$; $r_1=0$; $r_2=1$; $r_3=0$
- a: $W_x[0]=1$
- b: $R_x[0]=1$
- c: $R_y[0]=1$
- d: $W_y[0]=1$
- e: $R_x[0]=0$

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 T₁ and T₂:

\[
\begin{array}{c}
T₀ & \overset{rf}{\rightarrow} & T₁ \\
\text{a: } Wx=1 & \text{sync} & \text{b: } Rx=1 \\
& \text{sync} & \text{d: } Wy=1 \\
\text{c: } Ry=0 & \overset{fr}{\rightarrow} & \text{e: } Ry=1 \\
\text{f: } Rx=0 & \overset{fr}{\rightarrow} & \\
\end{array}
\]

IRIW+syncs

Propagation: Full fences order all communications.

Relation summary

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

“Execution” relations
- ▶ Program order: \( e₁ \overset{po}{\rightarrow} e₂ \), with \( \text{proc}(e₁) = \text{proc}(e₂) \).
- ▶ Same location program order: \( e₁ \overset{po-loc}{\rightarrow} e₂ \).
- ▶ Preserved program order: \( e₁ \overset{ppo}{\rightarrow} e₂ \), with \( \overset{ppo}{\rightarrow} \subseteq \overset{po}{\rightarrow} \). Computed from other relations, includes (effective) dependencies (control dependency from read to read is not effective).
- ▶ Fences: effective strong and lightweight fences in between \( \overset{strong}{\rightarrow} \) and \( \overset{light}{\rightarrow} \). Effective means that for instance \( w \overset{lwsync}{\rightarrow} r \) does not implies \( w \overset{light}{\rightarrow} r \).

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 = (((W*W) & fence)|(rfe; ((R*W) & fence)));hb*
irreflexive fre;propbase as observation

PROPAGATION Strong fences wait for all communications. Simple formulation:
let com = rf|fr|co
acyclic com|strong as propagation

In actual model, a more strict condition:
let prop = (W*W)&propbase|(com*;propbase*;strong;hb*)
acyclic co | prop as propagation

ARM/Power preserved program order

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

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

(* Initial value *)
let ci0 = ctrlisync | detour
let ii0 = dd | rfi | rdw
let cc0 = dd | po-loc | ctrl | addrpo
let ic0 = 0

(* Fixpoint from i \rightarrow c in instructions and transitivity *)
let rec ci = ci0 | (ci;ii) | (cc;ci) and ii = ii0 | ci | (ic;ci) | (ii;ii)
and cc = cc0 | ci | (ci;ic) | (cc;cc)
and ic = ic0 | ii | cc | (ic;cc) | (ii ; ic)

let ppo = [R]; ic; [W] | [R]; ii; [R]

Can be limited to dependencies...
ARMv8 is an "other multicopy atomic" architecture.

That is, writes are "performed" for all participants, as soon as "performed" for one (external) participant.

As regards tests, this means that, say WRC+data+addr and IRIW+addrs are forbidden (but SB+rfi-addrs, cf. slide 57, is still allowed).

From the axiomatic point of view, rfe (as well as fr and co) is part of happens-before. And the CAT model is simplified.

In effect, NO-THIN-AIR, OBSERVATION and PROPAGATION can be performed by one single check, here called "EXTERNAL".

A few details

Armv8 features load-acquire instructions — two of them, Acquire (LDAR) and AcquirePC (LDAPR), events A and Q; and store-release instructions — STLAR, events L.

(* Barrier-ordered-before *)

let bob = ... # Fences left out
| [A | Q]; po  # Acquire
| po; [L]    # Release
| [L]; po; [A] #

let lob = ... | bob | ... 

let ob = rfe | fre | coe | ... | lob | ...

Those rules, plus external communication being part of ordered-before entails that using load-acquire and store-releases restores SC.

Part 5.

Axiomatic C11
The C11, memory model, quick starter

C11 features "atomic" scalar types atomic_int, etc. and "atomic" operations atomic_store_explicit(p,v,m), atomic_load_explicit(p,m) (and more...).

It also feature fences atomic_thread_fence(m).

Where m is a "memory-order", relaxed, acquire, release, sequential consistent (and consume, neglected), with annoyingly long names memory_order_relaxed, ..., memory_order_seq_cst.

In C11 memory-order specifications result in sets of events RLX, ACQ, ...., SC. Those events can be reads or writes (sets R and W) but also fences (set F).

Significant differences, w.r.t. hardware models

- No real preserved-program-order, as po is part of happens-before hb. Defining dependencies is impossible for the sake of compiler optimisations.
- As a result, general communications cannot be part of hb. If so we define SC!
- C favors atomic accesses overs fences. This resulted in (initial) weak semantics of SC fences.
- In case of data-race: undefined behaviour:
  let conflict = ((W * _) | (_ * W)) & loc & ext
  let dr = conflict \ (hb | hb^-1 | A * A)
  # A = atomic access
  flag ~empty dr as DataRace

We present "Repaired C11"
"Repairing Sequential Consistency in C/C++11" Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur and Derek Dreyer: PLDI 2017.

(Repaired) C11 model. happens-before

The happens-before, hb relation is build from sb (sequenced-before, C-style program-order). and sw (synchronize-with).

We present a simplified view of actual synchronised-with... Notice that this sequence is similar to critical sections ordering: Lock is akin to load-acquire, UnLock to store-release.

RC11 happens-before, the full story

We have release-sequence, rs:

let RLX-OR-MORE = RLX | REL | ACQ_REL | ACQ | SC
let sb-loc = sb & loc
let rs = [W]; sb-loc?; [W & RLX-OR-MORE]; (rf;rmw)*

Notice that rs includes [W & REL], the most simple "release sequence".

Then, full synchronise-with:

let REL-OR-MORE = REL | ACQ_REL | SC
and ACQ-OR-MORE = ACQ | ACQ_REL | SC
let sw = [REL-OR-MORE]; ([F]; sb)?; rs;
rf;
[R & RLX-OR-MORE]; (sb; [F])?; [ACQ-OR-MORE]

let hb = (sb | sw)+
RC11, uniproc and propagation

let eco = (rf|fr|co)+ // Our old friend \(\text{com}\) irreflexive hb; eco? as coherence

Interestingly, “coherence” above regroups both UNIPROC (sb included in hb) and some generalised OBSERVATION (communication vs. hb)

<table>
<thead>
<tr>
<th>RC11 radical stance agains out-of-thin-air</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forbid any “LB” shape.</td>
</tr>
<tr>
<td>acyclic sb</td>
</tr>
<tr>
<td>To be compared with machine level NO-THIN-AIR</td>
</tr>
<tr>
<td>acyclic ppo</td>
</tr>
</tbody>
</table>

As a result, “causality” cycles are radically excluded.

Still in discussion, because such a solution entails a (light in our opinion) runtime penalty.

At present, alternative solutions are complex, roughly in operational semantics terms: they rely on forging values for reads (promises), and then checking that promises are fulfilled by any possible reduction in any context.

<table>
<thead>
<tr>
<th>Out of thin-air values cannot be neglected</th>
</tr>
</thead>
<tbody>
<tr>
<td>If any value can pop-up at any time no program proof is possible.</td>
</tr>
<tr>
<td>Allowing LB+datas over non-atomics (for instance) hinders the DRF theorem.</td>
</tr>
<tr>
<td>Out-of-thin-air values are not precisely defined, partly because dependencies are difficult to define in a (optimised) programming language.</td>
</tr>
</tbody>
</table>

```plaintext
int r0 = atomic_load_explicit(x, memory_order_relaxed) ;
int r1 = 0 ;
if (r0 == 42) { r1 = 42; } else { r1 = 42; }
atomic_store_explicit(y, r1, memory_order_relaxed) ;
```

```plaintext
int r2 = atomic_load_explicit(y, memory_order_relaxed) ;
atomic_store_explicit(x, r2, memory_order_relaxed) ;
```

Allow? (include sophisticated control dependencies definition in hb)
Forbid? (hinders optimisation?)

<table>
<thead>
<tr>
<th>Restoring SC: the big deal of RC11</th>
</tr>
</thead>
<tbody>
<tr>
<td>For SC atomics:</td>
</tr>
<tr>
<td>let sb-xy = sb \ loc # sb, different locations</td>
</tr>
<tr>
<td># SC-before</td>
</tr>
<tr>
<td>let sb = sb \ sb-xy; hb; sb-xy \ hb&amp;loc</td>
</tr>
</tbody>
</table>

```plaintext
let pscb = ([SC] \ [F & SC]; hb?) ; scb ; ([SC] | hb? \ [F & SC]);
let pscf = [F & SC]; (hb | hb; eco; hb) ; [F & SC]
```

acyclic pscb | pscf as sc

Given for completeness, some points

- Acyclicity of pscf entails "simple" strong fence SC-preserving condition
- Relation scb always includes a sb step in case included in hb\loc.
  This is weaker than simply including hb in scb. But it provides po steps compilation schemes that use strong fences for implementing SC atomics.
- Fence semantics significantly strengthened w.r.t. previous C11 formulations.
How good are our models?

Are they sound?
▶ Proofs of equivalence or at least of axiomatic models being weaker than operational ones.
▶ Proof of compilation correctness (from RC11 to .).
▶ Experiments
  ▶ Soundness w.r.t. hardware (ARMv7 being a bit problematic because of acknowledged read-after-read hazard).
  ▶ Experimental equivalence with our previous models.

Above all:
▶ Vendor approval (ARM Ltd. for ARMv8).
▶ Comitee acceptance (almost for RC11).

In any case:
▶ Simulation is fast.
▶ 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.

Some valuable readings you now have been introduced to


Bug or feature?

The following execution:

\[
\begin{align*}
\text{a: } Wz &= 1 \\
\text{b: } Ry &= 0 \\
\text{c: } Rz &= 2 \\
\text{d: } Rz &= 1 \\
\text{e: } Wz &= 2
\end{align*}
\]

is observed on all (tested) ARMv7 machines. It features a CoRR-style coherence violation (i.e. \( \text{po} \rightarrow \) contradicts \( \text{fr} \rightarrow ; \text{rf} \rightarrow \)). Notice: CoRR is not observed directly.
▶ Definitively a hardware anomaly.
▶ Not observed on ARMv8.