Not so practical multicore programming

A simple model for sequential consistency, extended...
Part 1.

Axiomatic Sequential Consistency
Shared memory computer

Thread\textsubscript{1} \quad \cdots \quad Thread\textsubscript{n}

\[ \begin{align*}
&\text{W} & \text{R} & \text{W} & \text{R} \\
\end{align*} \]

Shared Memory
Sequential consistency

Original definition: (Leslie Lamport)

[...] The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

(And stores take effect immediately).

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

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

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 \((a) : d[\ell] v\), where:

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

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

Relation $\mathord{\stackrel{\text{po}}{\rightarrow}}$ represents the sequential execution of events by one processor that follows the usual processor execution model, where instructions are executed by following the order given in program.
Formalism: program order

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

Relation $\xrightarrow{\text{po}}$ 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

```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): $W[x]_1$
Formalism: program order

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

Relation $\xrightarrow{\text{po}}$ 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

```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:

$$(a): \text{W}[x]1 \xrightarrow{\text{po}} (b): \text{R}[t+0]2$$
Formalism: program order

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

Relation $\rightarrow_{\text{po}}$ 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

```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:

$(a) : \text{W}[x]_1 \rightarrow_{\text{po}} (b) : \text{R}[t + 0]_2 \rightarrow_{\text{po}} (c) : \text{R}[t + 4]_3$
Formalism: program order

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

Relation $\mathrel{\text{po}}$ 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, } */

```c
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 :

$$
\begin{align*}
(a): \mathrel{\text{W}}[x]1 & \mathrel{\text{po}} (b): \mathrel{\text{R}}[t + 0]2 \\
& \mathrel{\text{po}} (c): \mathrel{\text{R}}[t + 4]3 \\
& \mathrel{\text{po}} (d): \mathrel{\text{W}}[y]5
\end{align*}
$$
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:

1. **Order** $<$ is compatible with program order:
   
   $$ e_1 \xrightarrow{\text{po}} e_2 \implies e_1 < e_2. $$

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

   $$ rf < \xrightarrow{\text{Def}} \left\{ (w, r) | w = \max(w', \text{loc}(w')) = \text{loc}(r) \land w' < r \right\}. $$
Example of a question on SC

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

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

How do we know? Let us enumerate all interleavings:

\( a, b, c, d \)
Example of a question on SC

<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:

\[ a, b, c, d \quad y=2; r_0=1; \]
\[ a, c, b, d \]
Example of a question on SC

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

Observed? $y=2; \ r0=0$

How do we know? Let us enumerate all interleavings:

- $a, b, c, d \quad y=2; \ r0=1$
- $a, c, b, d \quad y=1; \ r0=1$
- $a, c, d, b \quad$
Example of a question on 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) y \leftarrow 2 )</td>
</tr>
<tr>
<td>( b )</td>
<td>( 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:

- \( a, b, c, d \) \( \quad y=2; r_0=1; \)
- \( a, c, b, d \) \( \quad y=1; r_0=1; \)
- \( a, c, d, b \) \( \quad y=1; r_0=1; \)
- \( c, d, a, b \)
Example of a question on SC

<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:

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

**Remark:** If $b < c$ then $y = 2$, if $a < d$ then $r_0 = 1$. 
Example of a question on 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>$(c) y \leftarrow 2$</td>
</tr>
<tr>
<td>$(b)$</td>
<td>$y \leftarrow 1$</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:

- $a, b, c, d$: $y=2; r_0=1$
- $a, c, b, d$: $y=1; r_0=1$
- $a, c, d, b$: $y=1; r_0=1$
- $c, d, a, b$: $y=1; r_0=0$
- $c, a, b, d$: $y=1; r_0=1$
- $c, a, d, b$
Example of a question on SC

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

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

How do we know? Let us enumerate all interleavings:

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

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

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

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

Collecting constraints on the scheduling order $<$:

We respect program order, thus
Let us be a bit more clever

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

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

Collecting constraints on the scheduling order $<$:

We respect program order, thus $a < b$, $c < d$.
We observe $r0=0$, thus
Let us be a bit more clever

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

Observed? $y=2; r_0=0$

Collecting constraints on the scheduling order $<$:

We respect program order, thus $a < b$, $c < d$.

We observe $r_0=0$, thus $d < a$, as $d$ reads initial value, which is overwritten by $a$.

We observe $y=2$, thus
Let us be a bit more clever

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

Observed? $y=2; r0=0$

Collecting constraints on the scheduling order $<$:

We respect program order, thus $a < b$, $c < d$.

We observe $r0=0$, 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!

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

**Conclusion:** No SC execution would ever yield the output “$y=2; r0=0$;”.
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** ($\xrightarrow{\text{rf}}$): Relates write events to read events that read the stored value (initial writes left implicit in diagrams).

  $$\forall r, \exists! w, w \xrightarrow{\text{rf}} r$$

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

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

  For any location $\ell$, the restriction of $\xrightarrow{\text{co}}$ to write events to location $\ell$ ($W_\ell$) is a total strict order.
Coherence as a characteristics of shared memory

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

\[ \begin{align*}
W_{x0} \rightarrow_{\text{co}} W_{x1} \rightarrow_{\text{co}} x = 2 \rightarrow_{\text{co}} W_{x3} \rightarrow_{\text{co}} \ldots
\end{align*} \]

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

Example of $\xrightarrow{rf}$

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

<table>
<thead>
<tr>
<th></th>
<th>$r_0=1; r_1=1;$</th>
<th>$r_0=1; r_1=0;$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r_0=0; r_1=1;$</td>
<td>$r_0=0; r_1=0;$</td>
</tr>
</tbody>
</table>
Example of $\rightarrow^{rf}$

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

Observe: $r_0; r_1$

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

- $r_0=1; r_1=1$
  - a: $Rx=1$
  - b: $Wy=1$
- $r_0=1; r_1=0$
  - c: $Ry=1$
  - d: $Wx=1$
- $r_0=0; r_1=1$
- $r_0=0; r_1=0$
Example of $\rightarrow_{rf}$

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

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

<table>
<thead>
<tr>
<th></th>
<th>$r_0=1$; $r_1=1$;</th>
<th>$r_0=1$; $r_1=0$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>$Rx=1$</td>
<td>c: $Ry=1$</td>
</tr>
<tr>
<td>b:</td>
<td>$Wy=1$</td>
<td>d: $Wx=1$</td>
</tr>
<tr>
<td></td>
<td>$r_0=0$; $r_1=1$;</td>
<td>$r_0=0$; $r_1=0$;</td>
</tr>
</tbody>
</table>
Example of $\rightarrow^\text{rf}$

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

Observe: $r_0, r_1$;

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

<table>
<thead>
<tr>
<th></th>
<th>$r_0=1; r_1=1;$</th>
<th>$r_0=1; r_1=0;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>$Rx=1$</td>
<td>$Rx=1$</td>
</tr>
<tr>
<td>b:</td>
<td>$Wy=1$</td>
<td>$Wy=1$</td>
</tr>
<tr>
<td>c:</td>
<td>$Ry=1$</td>
<td>$Ry=0$</td>
</tr>
<tr>
<td>d:</td>
<td>$Wx=1$</td>
<td>$Wx=1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$r_0=0; r_1=1;$</th>
<th>$r_0=0; r_1=0;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>$Rx=1$</td>
<td>$Rx=1$</td>
</tr>
<tr>
<td>b:</td>
<td>$Wy=1$</td>
<td>$Wy=1$</td>
</tr>
<tr>
<td>c:</td>
<td>$Ry=0$</td>
<td>$Ry=0$</td>
</tr>
<tr>
<td>d:</td>
<td>$Wx=1$</td>
<td>$Wx=1$</td>
</tr>
</tbody>
</table>
Example of $\rightarrow^\text{rf}$

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

Observe: $r0; r1$;

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

---

- **Case 1:** $r0=1; r1=1$
  
  - a: $Rx=1$  
  - c: $Ry=1$
  
  - po
  
  - rf

- **Case 2:** $r0=1; r1=0$
  
  - a: $Rx=1$
  
  - c: $Ry=0$
  
  - po
  
  - rf

---

- **Case 3:** $r0=0; r1=1$
  
  - a: $Rx=0$
  
  - c: $Ry=1$
  
  - po
  
  - rf

- **Case 4:** $r0=0; r1=0$
  
  - a: $Rx=1$
  
  - c: $Ry=1$
  
  - po
  
  - rf

---

- **Case 5:** $r0=0; r1=1$
  
  - a: $Rx=0$
  
  - c: $Ry=1$
  
  - po
  
  - rf

- **Case 6:** $r0=1; r1=0$
  
  - a: $Rx=1$
  
  - c: $Ry=0$
  
  - po
  
  - rf
Example of $\rightarrow^{rf}$

<table>
<thead>
<tr>
<th></th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_0$</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 $\rightarrow^{rf}$ relations (initial value is 0).

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

<table>
<thead>
<tr>
<th></th>
<th>$a: , Rx=1$</th>
<th>$a: , Rx=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$b: , Wy=1$</td>
<td>$b: , Wy=1$</td>
</tr>
<tr>
<td></td>
<td>$c: , Ry=1$</td>
<td>$c: , Ry=0$</td>
</tr>
<tr>
<td></td>
<td>$d: , Wx=1$</td>
<td>$d: , Wx=1$</td>
</tr>
<tr>
<td>$a: , Rx=0$</td>
<td>$b: , Wy=1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$c: , Ry=1$</td>
<td>$c: , Ry=0$</td>
</tr>
<tr>
<td></td>
<td>$d: , Wx=1$</td>
<td>$d: , Wx=1$</td>
</tr>
</tbody>
</table>
Example of  

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

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

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

$x=2; y=2$;  
$x=2; y=1$;
Example of $\text{co}$

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

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

---

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

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

- $x=2; y=2;$
- $x=2; y=1;$
Example of $\overrightarrow{co}$

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

- $x=1; y=2;$
  - a: $Wx=2$
  - b: $Wy=1$
- $x=1; y=1;$
  - a: $Wx=2$
  - b: $Wy=1$

- $x=2; y=2;$
  - c: $Wy=2$
  - d: $Wx=1$
- $x=2; y=1;$
  - c: $Wy=2$
  - d: $Wx=1$
Example of $\rightarrow^\text{co}$

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

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

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

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

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

- a: $Wx=2$
- b: $Wy=1$
- c: $Wy=2$
- d: $Wx=1$
Example of \( \xrightarrow{\text{co}} \)

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

\[
\begin{array}{ll}
x=1; y=2; & \\
ap: Wx=2 & c: Wy=2 \\
b: Wy=1 & d: Wx=1 \\
\end{array}
\]

\[
\begin{array}{ll}
x=1; y=1; & \\
ap: Wx=2 & c: Wy=2 \\
b: Wy=1 & d: Wx=1 \\
\end{array}
\]

\[
\begin{array}{ll}
x=2; y=2; & \\
ap: Wx=2 & c: Wy=2 \\
b: Wy=1 & d: Wx=1 \\
\end{array}
\]

\[
\begin{array}{ll}
x=2; y=1; & \\
ap: Wx=2 & c: Wy=2 \\
b: Wy=1 & d: Wx=1 \\
\end{array}
\]
Example of $\langle \text{co} \rangle$

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

$x=1; \ y=2$

<table>
<thead>
<tr>
<th></th>
<th>$W_x=2$</th>
<th>$W_y=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>$W_y=1$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>$W_y=2$</td>
</tr>
<tr>
<td>d</td>
<td>$W_x=1$</td>
<td></td>
</tr>
</tbody>
</table>

$x=1; \ y=1$

<table>
<thead>
<tr>
<th></th>
<th>$W_x=2$</th>
<th>$W_y=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>$W_y=1$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>$W_y=2$</td>
</tr>
<tr>
<td>d</td>
<td>$W_x=1$</td>
<td></td>
</tr>
</tbody>
</table>

$x=2; \ y=2$

<table>
<thead>
<tr>
<th></th>
<th>$W_x=2$</th>
<th>$W_y=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>$W_y=1$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>$W_y=2$</td>
</tr>
<tr>
<td>d</td>
<td>$W_x=1$</td>
<td></td>
</tr>
</tbody>
</table>

$x=2; \ y=1$

<table>
<thead>
<tr>
<th></th>
<th>$W_x=2$</th>
<th>$W_y=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>$W_y=1$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>$W_y=2$</td>
</tr>
<tr>
<td>d</td>
<td>$W_x=1$</td>
<td></td>
</tr>
</tbody>
</table>

**Notice:** In this simple case of two stores, the value finally observed in locations determines $\langle \text{co} \rangle$ for them.
One more relation: $\text{fr}$

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

\[
r \xrightarrow{\text{fr}} w \quad \text{Def} \quad w' \xrightarrow{\text{rf}} r \land w' \xrightarrow{\text{co}} w
\]

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

Given

\[
\begin{array}{cccc}
  w_0 & \xrightarrow{\text{co}} & w_1 & \xrightarrow{\text{co}} \ldots & \xrightarrow{\text{co}} & w_n \\
    & \text{rf} &    &    &    & \text{rf} \\
  r &    &    &    &    & r
\end{array}
\]

We have
One more relation: \( \rightarrow_{fr} \)

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

\[
  r \rightarrow_{fr} w \ \overset{\text{Def}}{=} \ w' \rightarrow_{rf} r \land w' \rightarrow_{co} w
\]

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

Given

We have

(Or: \( \rightarrow_{fr} \))
One more relation: \( \mathfrak{fr} \rightarrow \)

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

\[
\begin{align*}
\mathfrak{r} \mathfrak{fr} \rightarrow \mathfrak{w} & \overset{\text{Def}}{=} \mathfrak{w}' \mathfrak{rf} \rightarrow \mathfrak{r} \land \mathfrak{w}' \mathfrak{co} \rightarrow \mathfrak{w}
\end{align*}
\]

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

Given

\[
\begin{array}{cccc}
\mathfrak{w}_0 & \mathfrak{co} & \mathfrak{w}_1 & \mathfrak{co} \\
\downarrow & & \downarrow & \\
\mathfrak{r} \mathfrak{rf} & & \mathfrak{w}_1 \mathfrak{co} & \ldots \mathfrak{co} \\
\downarrow & & \downarrow & \\
\mathfrak{r} & & \mathfrak{w}_n & \\
\end{array}
\]

We have

\[
\begin{array}{cccc}
\mathfrak{w}_0 & \mathfrak{co} & \mathfrak{w}_1 & \mathfrak{co} \\
\downarrow & & \downarrow & \\
\mathfrak{r} & & \mathfrak{w}_1 \mathfrak{co} & \ldots \mathfrak{co} \\
\downarrow & & \downarrow & \\
\mathfrak{r} & & \mathfrak{w}_n & \\
\end{array}
\]

(Or: \( \mathfrak{fr} \overset{\text{Def}}{=} \left( \mathfrak{rf} \right)^{-1} ; \mathfrak{co} \))
Playing with $\rightarrow^f r$

Particular, easy, case: a read from the initial state is in $\rightarrow^f r$ with writes by the program.

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

Observed? $r_0=1; r_1=0$

a: $W_x=1$

b: $W_y=1$

c: $R_y=1$

d: $R_x=0$
Playing with \( \rightarrow \)

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

<table>
<thead>
<tr>
<th>MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_0 )</td>
</tr>
<tr>
<td>( (a) x \leftarrow 1 )</td>
</tr>
<tr>
<td>( (b) y \leftarrow 1 )</td>
</tr>
<tr>
<td>Observed?</td>
</tr>
</tbody>
</table>

\[ a: Wx=1 \]
\[ b: Wy=1 \]
\[ c: Ry=1 \]
\[ d: Rx=0 \]
Particular, easy, case: a read from the initial state is in $\rightarrow$ with writes by the program.

<table>
<thead>
<tr>
<th>MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
</tr>
<tr>
<td>(a) $x \leftarrow 1$</td>
</tr>
<tr>
<td>(b) $y \leftarrow 1$</td>
</tr>
<tr>
<td>Observed?</td>
</tr>
</tbody>
</table>

a: $Wx=1$

b: $Wy=1$

c: $Ry=1$

d: $Rx=0$
Particular, easy, case: a read from the initial state is in $\rightarrow$ with writes by the program.

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

Observed? $r_0 = 1; r_1 = 0$

- **a:** $Wx = 1$
- **b:** $Wy = 1$
- **c:** $Ry = 1$
- **d:** $Rx = 0$
Playing with $\rightarrow$

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

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

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

a: $W_x=1$

b: $W_y=1$
c: $R_y=1$
d: $R_x=0$

a: $W_x=1$
b: $R_y=0$
c: $W_y=1$
d: $R_x=0$
Particular, easy, case: a read from the initial state is in $\xrightarrow{fr}$ with writes by the program.

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

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

a: $W_x=1$

b: $W_y=1$
c: $R_y=1$
d: $R_x=0$

p: p

r: rf

c: co

p: p

c: co

rf

rf

15
Playing with $\rightarrow_{fr}$

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>(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?</td>
<td>$r_0=1$; $r_1=0$</td>
</tr>
<tr>
<td>Observed?</td>
<td>$r_0=0$; $r_1=0$</td>
</tr>
</tbody>
</table>

a: $W_x=1$
b: $W_y=1$
c: $R_y=1$
d: $R_x=0$
a: $W_x=1$
b: $R_y=0$
c: $W_y=1$
d: $R_x=0$
Playing with $\text{fr}$

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

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

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

a: $W_x=1$  c: $R_y=1$

b: $W_y=1$  d: $R_x=0$

a: $W_x=1$  c: $W_y=1$

b: $R_y=0$  d: $R_x=0$
Second definition of SC

**Definition (SC 2)**

An execution is SC when:

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

And of course:

Theorem

The two definitions of SC are equivalent.
Second definition of SC

Definition (SC 2)
An execution is SC when:

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

And of course:

Theorem

The two definitions of SC are equivalent.
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: \( \text{rf} \overset{\text{Def}}{=} \text{rf}< \). Also notice \( \text{po} \subseteq < \), \( \text{co} \subseteq < \) and \( \text{rf} \subseteq < \).

Proof:

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

Let \( r \overset{\text{fr}}{\to} w \). Let further \( w_0 \overset{\text{rf}<}{\to} r \), then, by definition of \( \text{fr} \), we have \( w_0 \overset{\text{co}}{\to} w \) 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 “<”
SC 2 $\iff$ SC 1

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

From $<$ define $\rightarrow^{rf<}$:

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

and show $\rightarrow^{rf} = \rightarrow^{rf<}$.

1. Let $w_0 \rightarrow^{rf} r$ and let $w \in W_\ell, w \neq w_0$ then ($\rightarrow^{co}$ total order on $W_\ell$):

   1. Either $w \rightarrow^{co} w_0$ and $w < w_0 < r$.
   2. Or, $w_0 \rightarrow^{co} w$, and $r \rightarrow^{fr} w$, and thus $r < w$.

Finally $w_0 \rightarrow^{rf<} r$.

2. Let $w \not\rightarrow^{rf} r$ (i.e. $w \in W_\ell, w \neq w_0$), then

   1. Either $w \rightarrow^{co} w_0$, and thus ($\rightarrow^{co} \subseteq <$) $w \not\rightarrow^{rf<} r$.
   2. Or $w_0 \rightarrow^{co} w$, thus $r \rightarrow^{fr} w$, and thus ($\rightarrow^{fr} \subseteq <$) $w \not\rightarrow^{rf<} r$. 
Simulating SC

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

**SC 1** Enumerate interleavings.

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

Answer: we view SC 2 as being more convenient, since the generated objects usually are smaller.
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. $\text{po} \rightarrow, \text{rf} \rightarrow, \text{co} \rightarrow$); check the acyclicity of $\text{rf} \rightarrow \cup \text{co} \rightarrow \cup \text{fr} \rightarrow \cup \text{po} \rightarrow$.

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

A model sc.cat:

```
% cat 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/herd):

```
% herd7 -cat sc.cat R.litmus
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 $\xrightarrow{\text{po}}$, $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{co}}$ ($\xrightarrow{\text{fr}}$ deduced):

```
  a: Wx=1  c: Wy=2
  |    |    |
  |    |    |
  |    |    |
  b: Wy=1  d: Rx=1
```

```
  a: Wx=1  c: Wy=2
  |    |    |
  |    |    |
  |    |    |
  b: Wy=1  d: Rx=0
```
Herd structure

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

  - a: $W_x=1$
  - b: $W_y=1$
  - c: $W_y=2$
  - d: $R_x=1$
  
  Ok

  - a: $W_x=1$
  - b: $W_y=1$
  - c: $W_y=2$
  - d: $R_x=0$
  
  No

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

Studying Non-Sequentially Consistent Executions.
Violations of SC

A cycle of $\text{po} \rightarrow, \text{rf} \rightarrow, \text{co} \rightarrow, \text{fr} \rightarrow$ 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.
Simplifying cycles: \( \rightarrow_{\text{po}} \) and \( \rightarrow_{\text{com}} \) steps alternate

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

- \( \rightarrow_{\text{po}} \) is transitive \( \rightarrow_{\text{po}}^+ \subseteq \rightarrow_{\text{po}} \).
- \( \rightarrow_{\text{com}}^+ \) is the union of the five following relations:

\[
\rightarrow_{\text{com}} \rightarrow_{\text{RF}} \cup \rightarrow_{\text{CO}} \cup \rightarrow_{\text{FR}} \cup \rightarrow_{\text{RF}} \cup \rightarrow_{\text{CO}} \cup \rightarrow_{\text{FR}} \cup \rightarrow_{\text{RF}} \cup \rightarrow_{\text{CO}} \cup \rightarrow_{\text{FR}} \cup \rightarrow_{\text{RF}} \cup \rightarrow_{\text{CO}} \cup \rightarrow_{\text{FR}}
\]
Simplifying cycles: $\xrightarrow{\text{po}}$ and $\xrightarrow{\text{com}}$ steps alternate

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

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

$$\xrightarrow{\text{com}}^+ = \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}} \cup \left( \xrightarrow{\text{co}}; \xrightarrow{\text{rf}} \right) \cup$$
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 $(\text{po} \rightarrow^+; \text{com} \rightarrow^+)$ (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:

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

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

**Conclusion:** Any cyclic $\text{com} \rightarrow \cup \text{po} \rightarrow$ includes a cycle in $(\text{po} \rightarrow; \text{com} \rightarrow)$ — 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 $\xrightarrow{\hat{\text{com}}}$ steps are external, (i.e. source and target events are from pairwise distinct threads).

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 $e_2 \xrightarrow{\text{po}} e_1$, 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 $\xrightarrow{\hat{\text{com}}}$. 
Simplifying cycles – Threads

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

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

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

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

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

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

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

**Conclusion:** Cycle visit a thread at most once.
Consider a test execution on two threads:

```
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Rx=1</td>
<td>e: Ry=1</td>
</tr>
<tr>
<td>b: Wy=1</td>
<td>f: Wz=1</td>
</tr>
<tr>
<td>c: Rz=1</td>
<td>g: Ra=1</td>
</tr>
<tr>
<td>d: Wa=1</td>
<td>h: Wx=1</td>
</tr>
</tbody>
</table>
```
Test from cycles — Threads

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

Consider a test execution on two threads:
The test execution features a smaller cycle

a: $Rx=1$
b: $Wy=1$
c: $Rz=1$
d: $Wa=1$
e: $Ry=1$
f: $Wz=1$
g: $Ra=1$
h: $Wx=1$
Test from cycles — Threads

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

Consider a test execution on two threads:

a: $Rx=1$

b: $Wy=1$

c: $Rz=1$

d: $Wa=1$

e: $Ry=1$

f: $Wz=1$

g: $Ra=1$

h: $Wx=1$

Generally: one passage per thread
Test from cycles — Locations

Cycle:  $R \xrightarrow{p_0} W \xrightarrow{r_f} R \xrightarrow{p_0} W \xrightarrow{r_f} R \xrightarrow{p_0} W \xrightarrow{r_f} R \xrightarrow{p_0} W \xrightarrow{r_f}$

- One interpretation (four locations):
  
  a: $R_x=1$  
  b: $W_y=1$  
  c: $R_y=1$  
  d: $W_z=1$  
  e: $R_z=1$  
  f: $W_a=1$  
  g: $R_a=1$  
  h: $W_x=1$

- Another interpretation (two locations):
  
  a: $R_x=2$  
  b: $W_y=1$  
  c: $R_y=1$  
  d: $W_x=1$  
  e: $R_x=1$  
  f: $W_y=2$  
  g: $R_y=2$  
  h: $W_x=2$
The second interpretation is not “minimal”

Reminding the interpretation with two locations:

- **a**: $R_x=2$
- **b**: $W_y=1$
- **c**: $R_y=1$
- **d**: $W_x=1$
- **e**: $R_x=1$
- **f**: $W_y=2$
- **g**: $R_y=2$
- **h**: $W_x=2$

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

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

a: $Rx=2$

b: $Wy=1$

c: $Ry=1$

d: $Wx=1$

e: $Rx=1$

f: $Wy=2$

g: $Ry=2$

h: $Wx=2$

We have a smaller cycle: $d \xrightarrow{co} h \xrightarrow{rf} a \xrightarrow{po} b \xrightarrow{rf} c \xrightarrow{po} d$. 
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 $\rightarrow$ totally orders write events to a given location.

Let us choose: $Wx2 \rightarrow Wx1$:

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

We have a smaller cycle: $h \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h$. 
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 $\rightarrow$ totally orders write events to a given location.

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

**Lemma (Identical locations)**

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

1. either \( e_1 \xrightarrow{\text{com}} e_2 \),
2. or \( e_2 \xrightarrow{\text{com}} e_1 \),
3. 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 \( r_1 \xrightarrow{\text{fr}} w_2 \xrightarrow{\text{rf}} r_2 \).
- \( r_1, w_2 \), 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 \( r_1 \xrightarrow{\text{fr}} w_2 \); or \( w_2 \xrightarrow{\text{co}} w_1 \) and \( w_2 \xrightarrow{\text{co}} \xrightarrow{\text{rf}} r_1 \).
Simplifying cycles, a lemma

**Lemma (Identical locations)**

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

1. **either** $e_1 \xrightarrow{\text{com}} e_2$,
2. **or** $e_2 \xrightarrow{\text{com}} e_1$,
3. **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 $r_1 \xrightarrow{\text{fr}} w_2 \xrightarrow{\text{rf}} r_2$.
- $r_1, w_2$, 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 $r_1 \xrightarrow{\text{fr}} w_2$; or $w_2 \xrightarrow{\text{co}} w_1$ and $w_2 \xrightarrow{\text{co}} \xrightarrow{\text{rf}} r_1$.

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

1. If $e_1$ and $e_2$ are from different threads. By hypothesis, $e_1$ and $e_2$ are related by complex steps (\textit{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, $w \xrightarrow{rf} e_1$ and $w \xrightarrow{rf} e_2$, — see next page!.

2. If $e_1$ and $e_2$ are from the same thread, \textit{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 $e_2 \xrightarrow{\text{com}} e_2$ and we have a very simple cycle $e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}} e_1$.
   - Or $w \xrightarrow{rf} e_1$ and $w \xrightarrow{rf} e_2$, we short-circuit the cycle — as the cycle must be $\cdots w \xrightarrow{rf} e_1 \xrightarrow{\text{po}} e_2 \cdots$, which we reduce into $\cdots w \xrightarrow{rf} e_2 \cdots$. 
So let us assume a cycle that includes $r_1$ and $r_2$, related in both directions by complex steps and such that $w \xrightarrow{rf} r_1$ and $w \xrightarrow{rf} r_2$. We consider:

- If $w \xrightarrow{rf} r_1$ is in cycle, then there is an obvious short-circuit: replace $\xrightarrow{rf}$ followed by the complex steps from $r_1$ to $r_2$ by a single $w \xrightarrow{rf} r_2$ step.

- If $w \xrightarrow{rf} r_2$ is in cycle, symmetrical case.

- Otherwise, it must be that both $r_1$ and $r_2$ are the target of $\xrightarrow{po}$ steps and the source of $\xrightarrow{fr}$ steps: let $w_1$ and $w_2$ be the targets of those steps.

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

Simplifying cycles — Conclusion

In a non SC execution we find:

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

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

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

A violation of coherence is a cycle $e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}} e_1$. Given the definition of $\xrightarrow{\text{com}}$, there are five such cycles, which can occur as the following executions: $\xrightarrow{\text{po}}$ contradicts
Violations of coherence

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

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

\[
\begin{align*}
a: \ & \text{Wx=1} \\
\text{b: \ Wx=2}
\end{align*}
\]

CoWW
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 \( \xrightarrow{\text{com}} \), there are five such cycles, which can occur as the following executions: \( \xrightarrow{\text{po}} \) contradicts \( \xrightarrow{\text{co}}, \xrightarrow{\text{rf}} \),

\[
\begin{align*}
ap: \ W x=1 & \quad a: \ Rx=1 \\
b: \ W x=2 & \quad b: \ W x=1 \\
\text{CoWW} & \quad \text{CoRW1}
\end{align*}
\]
Violations of coherence

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

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

- CoWWW: a: Wx=1, b: Wx=2
- CoRW1: a: Rx=1, b: Wx=1
- CoWR: a: Wx=1, b: Rx=0
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 \( \xrightarrow{\text{com}} \), there are five such cycles, which can occur as the following executions: \( \xrightarrow{\text{po}} \) contradicts \( \xrightarrow{\text{co}} \), \( \xrightarrow{\text{rf}} \), \( \xrightarrow{\text{fr}} \), “\( \xrightarrow{\text{co}} \); \( \xrightarrow{\text{rf}} \)”,

- **CoWW**
  - a: \( Wx=1 \)
  - b: \( Wx=2 \)
- **CoRW1**
  - a: \( Rx=1 \)
  - b: \( Wx=1 \)
- **CoWR**
  - a: \( Wx=1 \)
  - b: \( Rx=0 \)
  - c: \( Wx=2 \)
- **CoRW2**
  - a: \( Rx=2 \)
  - b: \( Wx=1 \)
  - c: \( Wx=2 \)
Violations of coherence

A violation of coherence is a cycle $e_1 \xrightarrow{po} e_2 \xrightarrow{\text{com}} e_1$. Given the definition of $\xrightarrow{\text{com}}$, there are five such cycles, which can occur as the following executions: $\xrightarrow{po}$ contradicts $\xrightarrow{co}$, $\xrightarrow{rf}$, $\xrightarrow{fr}$, “$\xrightarrow{co}$; $\xrightarrow{rf}$”, “$\xrightarrow{fr}$; $\xrightarrow{rf}$”.

![Diagrams of CoWW, CoRW1, CoWR, CoRW2, and CoRR]
Application, all possible SC violations on two threads

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

- **2+2W**
  - po → co → po → co
  - po → rf → po → rf

- **LB**
  - po → rf → po → rf

- **MP**
  - po → rf → po → fr

- **R**
  - po → co → po → fr

- **S**
  - po → rf → po → co

- **SB**
  - po → fr → po → 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>2+2W</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_0$</td>
</tr>
<tr>
<td></td>
<td>$(a) x \leftarrow 2$</td>
</tr>
<tr>
<td></td>
<td>$(b) y \leftarrow 1$</td>
</tr>
<tr>
<td>Observed?</td>
<td>$x=2$; $y=2$;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_0$</td>
</tr>
<tr>
<td></td>
<td>$(a) r0 \leftarrow x$</td>
</tr>
<tr>
<td></td>
<td>$(b) y \leftarrow 1$</td>
</tr>
<tr>
<td>Observe:</td>
<td>$r0=1$; $r1=1$;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_0$</td>
</tr>
<tr>
<td></td>
<td>$(a) x \leftarrow 1$</td>
</tr>
<tr>
<td></td>
<td>$(b) y \leftarrow 1$</td>
</tr>
<tr>
<td>Observed?</td>
<td>$r0=1$; $r1=0$;</td>
</tr>
</tbody>
</table>
# Three more violations of SC

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T₀</strong></td>
<td>x ← 1</td>
<td>x ← 2</td>
<td>x ← 1</td>
</tr>
<tr>
<td><strong>T₁</strong></td>
<td>y ← 2</td>
<td>r0 ← y</td>
<td>r0 ← y</td>
</tr>
<tr>
<td><strong>Observed?</strong></td>
<td>y=2; r₀=0</td>
<td>x=2; r₀=1</td>
<td>r₀=0; r₁=0</td>
</tr>
<tr>
<td><strong>a</strong></td>
<td>Wₓ=1</td>
<td>Wₓ=2</td>
<td>Wₓ=1</td>
</tr>
<tr>
<td><strong>b</strong></td>
<td>Wᵧ=1</td>
<td>Rₓ=0</td>
<td>Rᵧ=0</td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>Wᵧ=2</td>
<td>Rᵧ=1</td>
<td>Wᵧ=1</td>
</tr>
<tr>
<td><strong>d</strong></td>
<td>Rₓ=0</td>
<td>Wₓ=1</td>
<td>Rx=0</td>
</tr>
</tbody>
</table>

### Observation

- **R**
  - x ← 1 (a)
  - y ← 1 (b)
- **S**
  - x ← 2 (a)
  - y ← 1 (b)
- **SB**
  - r₀ ← y (a)
  - r₁ ← x (b)
Application

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

To guarantee SC:
- Find all possible critical cycles of all possible executions on the architecture.
- Insert a fence in every $\rightarrow_{po}$ step of those.

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

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

```c
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 ;
}
```

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

```c
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: int t = x; sum += t;
    f: go = 0 ;
}
```
A semi realistic example

```c
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 ;
}

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) ;
}
```

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

```c
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 ;
}
```

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() ;
}
```

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

```c
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 ;
}
```

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() ;
}

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

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

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

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

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

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

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

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

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

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

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

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

\textbf{X86}:
There are six cycles

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

\[ a \xrightarrow{\text{lwsync}} b, \quad d \xrightarrow{\text{ctrlisync}} e, \]
\[ a \xrightarrow{\text{sync}} b, \quad f \xrightarrow{\text{sync}} e, \]

X86: \[ f \xrightarrow{\text{mfence}} e, \]
There are six cycles

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

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

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

X86: \begin{align*}
& f \xrightarrow{\text{mfence}} e, \\
& a \xrightarrow{\text{mfence}} c, \\
& f \xrightarrow{\text{mfence}} e
\end{align*}
There are six cycles

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

X86: \[ f \xrightarrow{\text{mfence}} e, \ a \xrightarrow{\text{mfence}} c, \ f \xrightarrow{\text{mfence}} e \]
There are six cycles

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

X86: $f \xrightarrow{mfence} e$, $a \xrightarrow{mfence} c$, $f \xrightarrow{mfence} e$
There are six cycles

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

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

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

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

Fence $f,e$

```
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: int t = x; sum += t;
    f: go = 0 ;
        mfence() ;
}
```
Sufficient fencing, X86

Fence $a,c$

```c
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() ;
}
```
Sufficient fencing, X86

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() ;
}

Notice: Inserting full memory fence between racy writes gives the same result.
Fence \(a, b\) (and \(a, c\))

```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;
    f: go = 0;
}
```
Sufficient fencing, Power

Fence $c,a$ (and $c,a$)

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

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

Fence $d,e$ (and $f,e$)

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

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

Fence $e,f$ (and $e,d$)

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

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

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

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

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

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

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

Notice: Inserting full memory fence between racy accesses is much more simple.
```
Part 3.

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

**Demo:** (in demo/TS01) 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 test on this machine
Results for running the six test on this machine

- **a:** $W_x=1$
  - $W_y=1$
- **b:** $W_x=1$
  - $W_y=1$
- **c:** $W_y=2$
- **d:** $R_x=0$

- **a:** $W_x=2$
  - $R_y=1$
- **b:** $W_x=1$
  - $W_x=1$
- **c:** $R_y=1$
- **d:** $W_x=1$

- **R:** Ok

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

- **a:** $W_x=1$
  - $W_x=1$
  - $W_x=2$
  - $W_x=1$
- **b:** $W_x=1$
  - $W_x=1$
- **c:** $W_y=2$
- **d:** $W_x=1$


Results for running the six test on this machine

**R:** Ok

**S:** No
Results for running the six test on this machine

R: Ok

S: No

SB: Ok
Results for running the six test on this machine

R: Ok

S: No

SB: Ok

MP: No

LB: No

2+2W: No
Axiomatic TSO, model TSO 1

- Remember SC:

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

A model for herd, our generic simulator:

let ppo = po # ppo stands for 'preserved program-order'
let com-hb = fr | rf | co # All communications create order
acyclic (ppo | com-hb)

- In TSO:
  - Write-to-read does not create order:
    let ppo = (R*M | W*W) & po # All pairs except W*R pairs
  - Communication create order
    let com-hb = rf | co | fr

- TSO “happens-before” (HB) check:

acyclic (ppo | com-hb | mfence) as hb
Axiomatic TSO, model TSO 1

- Remember SC:

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

A model for herd, our generic simulator:

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

- In TSO:
  - Write-to-read does not create order:
    let ppo = (R*M | W*W) & po \# All pairs except W*R pairs
  - Communication create order
    let com-hb = rf | co | fr

- TSO "happens-before" (HB) check:
  acyclic (ppo | com-hb | mfence) as hb

Notice: Relations can be interpreted as being between the points in time where a load binds its value and where a written value reaches memory.
Restoring SC with \texttt{mfence}

Replace “relaxed” (not in HB) \( WR(\xrightarrow{\text{po}}) \) by \( \xrightarrow{\text{mfence}} \) (in HB).

\begin{align*}
\text{R+po+mfence} \\
\begin{array}{c|c}
T_0 & T_1 \\
\hline
(a) x \leftarrow 1 & (c) y \leftarrow 2 \\
(b) y \leftarrow 1 & \text{mfence} \\
(d) r0 \leftarrow x & \\
\hline
\text{Observed?} & y=2; \ r0=0
\end{array}
\end{align*}

\begin{align*}
\text{a: Wx=1} & \quad \text{b: Wy=1} & \quad \text{c: Wy=2} & \quad \text{d: Rx=0} \\
\text{po} & \quad \text{mfence} & \quad \text{fr} & \quad \text{co}
\end{align*}
Restoring SC with mfence

Replace “relaxed” (not in HB) WR(\(\xrightarrow{\text{po}}\)) by \(\xrightarrow{\text{mfence}}\) (in HB).

### R+po+mfence

<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>(\text{mfence})</td>
</tr>
<tr>
<td>(d) (r_0 \leftarrow x)</td>
<td>(\text{mfence})</td>
</tr>
</tbody>
</table>

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

### SB+mfences

<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>(\text{mfence})</td>
<td>(\text{mfence})</td>
</tr>
<tr>
<td>(b) (r_0 \leftarrow y)</td>
<td>(d) (r_1 \leftarrow x)</td>
</tr>
</tbody>
</table>

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

---

**Diagram:**

- **a:** \(Wx=1\)
- **b:** \(Wy=1\)
- **c:** \(Wy=2\)
- **d:** \(Rx=0\)

**Observed?**

- **No**
Restoring SC with mfence

Replace “relaxed” (not in HB) $WR(\xrightarrow{p_0})$ by $\xrightarrow{mfence}$ (in HB).

### $R+p_0+mfence$

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

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

### $SB+mfences$

<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>$mfence$</td>
</tr>
<tr>
<td>(b)</td>
<td>$r_0 \leftarrow y$</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>$y \leftarrow 1$</td>
<td>$mfence$</td>
</tr>
<tr>
<td>(d)</td>
<td>$r_1 \leftarrow x$</td>
<td></td>
</tr>
</tbody>
</table>

Observed? $r_0=0$; $r_1=0$
Our TSO 1 model is wrong!

Consider:

<table>
<thead>
<tr>
<th>SB+rfi-pos</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 ?

a: $Wx=1$

b: $Rx=1$

c: $Ry=0$

d: $Wy=1$

e: $Ry=1$

f: $Rx=0$
Our TSO 1 model is wrong!

Consider:

<table>
<thead>
<tr>
<th>SB+rfi-pos</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_0 )</td>
<td>( T_1 )</td>
</tr>
<tr>
<td>(a) ( x \leftarrow 1 )</td>
<td>(d) ( y \leftarrow 1 )</td>
</tr>
<tr>
<td>(b) ( r_0 \leftarrow x )</td>
<td>(e) ( r_2 \leftarrow y )</td>
</tr>
<tr>
<td>(c) ( 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:

\[
\begin{align*}
a & \xrightarrow{rf} b \xrightarrow{po}_{RR} c \xrightarrow{fr} d \xrightarrow{rf} e \xrightarrow{po}_{RR} f \xrightarrow{fr} a
\end{align*}
\]

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

The effect originates from “store forwarding”: A thread can read its own writes from its store buffer, i.e. before they reach memory.
Observation of **SB+rfi-pos**

Demo in demo/TS02.

- Create test from cycle:

  % diyone7 -norm -arch X86 Rfi PodRR Fre Rfi PodRR Fre
  % ls
  SB+rfi-pos.litmus

- Run test:

  % litmus7 -mach x86.cfg src/SB+rfi-pos.litmus

...
Corrected model: TSO 2

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

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

The new hb is no longer cyclic:

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

CoRW1

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

CoWR

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

CoRW1

\begin{align*}
\text{a: } & R_x=1 \\
\text{b: } & W_x=1
\end{align*}

CoWR

\begin{align*}
\text{a: } & W_x=1 \\
\text{b: } & R_x=0
\end{align*}

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

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

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

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

Let \(\text{complus} = \text{rf} | \text{fr} | \text{co} | (\text{co};\text{rf}) | (\text{fr};\text{rf})\)

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 **HB** check.
Our final TSO model

TS03

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

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

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

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

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

**Definition (Uniproc 1)**

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

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

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

**Definition (Uniproc 2)**

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

Definitions are equivalent.
Equivalence of uniproc definitions

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

Conversely, we use the “Identical locations” lemma.

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

- Either, $r_1 \xrightarrow{\text{po}} r_2$, with $w \xrightarrow{\text{rf}} r_1$ and $w \xrightarrow{\text{rf}} r_2$. We short-circuit the $\text{po}$ step, replacing $w \xrightarrow{\text{rf}} r_1 \xrightarrow{\text{po}} r_2$ by $w \xrightarrow{\text{rf}} r_2$.

- Or, $e_1 \xrightarrow{\text{com}} e_2$. 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 $r \xrightarrow{\text{rmw}} w$. Additionally, $r$ and $w$ are tagged “atomic”.

**Example:** $\text{xchgl } r, x$.

We further enforce:

- Writes $w'$ to the location are either before the pair or after it:

$$\left( r \xrightarrow{\text{rmw}} w \right) \implies \left( w' \xrightarrow{\text{rf}} r \lor w' \xrightarrow{\text{co}} r \lor w \xrightarrow{\text{co}} w' \right)$$

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

$$\text{rmw} \cap (\xrightarrow{\text{fr}}; \xrightarrow{\text{co}}) = \emptyset$$

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

The ATOM check forbids this execution:

<table>
<thead>
<tr>
<th>EXCH</th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $x \leftarrow 1$</td>
<td></td>
<td>$r \leftarrow 2$</td>
</tr>
<tr>
<td>($b/c$) $r1 \leftrightarrow x$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

a: $Wx=1$  b: $Rx*=0$

c: $Wx*=2$
### Implied fences

Implied fences forbid this execution

**SB+EXCH**

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

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

---

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

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

(* Uniproc *)
let comhat = rf | fr | co | (co;rf) | (fr;rf) # or (rf|fr|co)+ irreflexive po; comhat as uniproc

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

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

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

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

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

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

$$\ldots$$

(* let ppo = (R*W|W*W) & po *)
let ppo = [R];po;[M] | [W];po;[W]

$$\ldots$$
Part 4.

Axiomatic ARM/Power
A relaxed shared memory computer

More or less visible to user code:

- **Cores:**
  - Out of order execution
  - Branch speculation
  - Write buffers

- **Memory**
  - Physically distributed
  - Caches
Situation of (our) ARM/Power models

- **Architecture public reference** Informal, cannot clearly explain how fences restore SC for instance.

- **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, $\mapsto_{\text{ppo}}$ 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:

% 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 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 $\text{ppo}$ as simply as for TSO.
An experiment on ARM/Power

Consider test **MP:**

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

Observed? $r0=1; \ r1=0$

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.
Dependencies (Power)

Address dependency:

\[ r_1 \leftarrow x \]
\[ r_2 \leftarrow t[r_1] \]

\[
\begin{align*}
\text{lwz } r_1,0(r_8) & \quad \text{# } r_8 \text{ contains the address of 'x'} \\
\text{slwi } r_7, r_1, 2 & \quad \text{# } \text{sizeof(int)} = 4 \\
\text{lwzx } r_2, r_7, r_9 & \quad \text{# } r_9 \text{ contains the address of 't'}
\end{align*}
\]

Data dependency:

\[
\begin{align*}
\text{lwz } r_1,0(r_8) & \quad \text{# } r_8 \text{ contains the address of 'x'} \\
\text{addi } r_2, r_1, 1 & \\
\text{stw } r_2,0(r_9) & \quad \text{# } r_9 \text{ contains the address of 'y'}
\end{align*}
\]

Control dependency:

\[
\begin{align*}
lwz & \quad r_1,0(r_8) \\
\text{cmpwi } & \quad r_1,0 \\
bne & \quad \text{L1} \\
\text{li } & \quad r_2,1 \\
\text{stw } & \quad r_2,0(r_9) \\
\text{L1: } & \\
\end{align*}
\]
Dependencies (Power)

Address dependency:

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

\[
\begin{align*}
lwz & \ r1,0(r8) \ # \ r8 \ contains \ the \ address \ of \ 'x' \\
slwi & \ r7,1,2 \ # \ sizeof(int) = 4 \\
lwzx & \ r2,r7,r9 \ # \ r9 \ contains \ the \ address \ of \ 't'
\end{align*}
\]

Data dependency:

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

\[
\begin{align*}
lwz & \ r1,0(r8) \ # \ r8 \ contains \ the \ address \ of \ 'x' \\
addi & \ r2,r1,1 \\
stw & \ r2,0(r9) \ # \ r9 \ contains \ the \ address \ of \ 'y'
\end{align*}
\]

Control dependency: (+isync)

\[
\begin{align*}
\text{r1} & \leftarrow x \\
\text{if} \ r1=0 \ \text{then} \\
\ (\text{isync}) \\
y & \leftarrow 1
\end{align*}
\]

\[
\begin{align*}
lwz & \ r1,0(r8) \\
cmpwi & \ r1,0 \\
\text{bne} & \ \text{L1} \\
\ (\text{isync}) \\
li & \ r2,1 \\
stw & \ r2,0(r9)
\end{align*}
\]

L1:
Generating tests (ARM), yet another tool: diycross

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

```
% diycross -arch ARM\
  PodWW,DMBdWW,DSBdWW,ISBdWW\
  Rfe\
  PodRR,DpCtrlIdR,DpCtrlIsbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR\ Fre
Generator produced 28 tests
```

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

This generates some variations in the **MP** family.

We then compile and run, and...
Optimal fencing/dependencies for MP
Optimal fencing for the 6 two-threads tests (Power)

R+syncs

a: \(W_x=1\)  
b: \(W_y=1\)  
c: \(W_y=2\)  
d: \(R_x=0\)

S+lwsync+addr

a: \(W_x=2\)  
b: \(W_y=1\)  
c: \(R_y=1\)  
d: \(W_x=1\)

SB+syncs

a: \(W_x=1\)  
b: \(R_y=0\)  
c: \(W_y=1\)  
d: \(R_x=0\)

MP+lwsync+addr

a: \(W_x=1\)  
b: \(W_y=1\)  
c: \(R_y=1\)  
d: \(R_x=0\)

LB+addrs

a: \(R_x=1\)  
b: \(W_y=1\)  
c: \(R_y=1\)  
d: \(W_x=1\)

2+2W+lwsyncs
Some observations

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

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

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

\[
\begin{align*}
\text{a: } & W_x=2 & \text{c: } & W_y=2 \\
\text{b: } & W_y=1 & \text{d: } & W_x=1
\end{align*}
\]

\text{2+2W+lwsyncs}

\[
\begin{align*}
\text{a: } & W_x=1 & \text{c: } & W_y=2 \\
\text{b: } & W_y=1 & \text{d: } & R_x=0
\end{align*}
\]

\text{R+syncs}

No No
Some observations

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

\[ addr < lwsync < sync \]

Then:

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

\[ \begin{align*}
  & 2+2W+lwsyncs \\
  & R+lwsync+sync \\
\end{align*} \]

No \hspace{1cm} Ok
Dependencies are enough

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

Observed? $r_0=42; r_1=42$

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}:

\begin{tabular}{c|c|c}
\hline
 & \textbf{T}_0 & \textbf{T}_1 \\
\hline
(a) & \texttt{x[0] ← 1} & \texttt{r0 ← x} \\
& & \texttt{y ← r0} \\
(b) & \texttt{rf data} & \texttt{fr addr} \\
\hline
\textbf{WRC+data+addr} & & \\
\hline
\textbf{Observed?} & \texttt{r0=1; r2=0;} & \\
\hline
\end{tabular}

\textbf{Observed?} \texttt{r0=1; r2=0;}

\begin{itemize}
\item \texttt{a: Wx[0]=1}
\item \texttt{b: Rx[0]=1}
\item \texttt{c: Wy[0]=1}
\item \texttt{d: Ry[0]=1}
\item \texttt{e: Rx[0]=0}
\end{itemize}

\textbf{Behaviour is legal on Power 6,7 (observed) and ARMv7 (non observed).}

\textbf{Stores are not “multi-copy atomic”} \texttt{T}_0 \text{ and } \texttt{T}_1 \text{ share a private buffer/cache/memory (e.g. a cache in SMT context).} \texttt{T}_2 \textit{“does not see”} the store by \texttt{T}_0, when \texttt{T}_1 \text{ does.}
Restoring SC for **WRC**

Use a lightweight fence on $T_1$:

$$
T_0 \quad T_1 \quad T_2
$$

a: $Wx=1$  \(\text{rf}\)  

\(\text{lwsync}\)  

c: $Wy=1$

b: $Rx=1$  \(\text{rf}\)  

d: $Ry=1$  \(\text{addr}\)  

e: $Rx=0$

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**

$$
T_0 \quad T_1 \quad T_2
$$

a: $Wx=1$  \(\text{rf}\)  

\(\text{lwsync}\)  

b: $Wy=1$

c: $Ry=1$  \(\text{fr}\)  

d: $Rx=0$  \(\text{addr}\)

MP+lwsync+addr
Another, symmetric, case of insufficient dependencies

Consider test \textbf{IRIW+addrs}:

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
& $T_0$ & $T_1$ & $T_2$ \\
\hline
(a) $x[0] \leftarrow 1$ & (b) $r0 \leftarrow x[0]$ & (d) $y[0] \leftarrow 1$ & (e) $r2 \leftarrow y[0]$ \\
& \hspace{1cm} $t \leftarrow r0 \wedge r0$ & \hspace{1cm} \ & \hspace{1cm} $t \leftarrow r2 \wedge r2$ \\
& \hspace{1.5cm} (c) $r1 \leftarrow y[t]$ & \hspace{1.5cm} \ & \hspace{1.5cm} (f) $r3 \leftarrow x[t]$ \\
\hline
\end{tabular}
\end{center}

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

\textbf{IRIW+addrs}

\begin{itemize}
\item a: $Wx[0]=1$
\item b: $Rx[0]=1$
\item d: $Wy[0]=1$
\item e: $Ry[0]=1$
\item c: $Ry[0]=0$
\item f: $Rx[0]=0$
\end{itemize}

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

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

Use a full fence on $T_1$ and $T_2$:

- a: $W_x=1$
- b: $R_x=1$
- c: $R_y=0$
- d: $W_y=1$
- e: $R_y=1$
- f: $R_x=0$

**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 \( \text{loc}(w) = \text{loc}(w') = x \). 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, \( \xrightarrow{rfi}, \xrightarrow{coi}, \xrightarrow{fri} \)) and external (different procs, \( \xrightarrow{rfe}, \xrightarrow{coe}, \xrightarrow{fre} \)) 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: \( e_1 \xrightarrow{pppo} e_2 \), with \( \xrightarrow{pppo} \subseteq \xrightarrow{po} \). 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 \( \xrightarrow{\text{strong}} \) and \( \xrightarrow{\text{light}} \). Effective means that for instance \( w \xrightarrow{lwsync} r \) does not implies \( w \xrightarrow{\text{light}} r \).
A model in four checks (TOPLAS’14)

UNIPROC
acyclic poloc | com as uniprocn

NO-THIN-AIR

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

OBSERVATION Fences (any fences) order writes:

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

PROPAGATION Strong fences order 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 -> 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 model

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 \( \text{WRC+data+addr} \) and \( \text{IRIW+addr} \) are forbidden (but \( \text{SB+rfi-addrs} \), cf. slide 52, is still allowed).

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

In effect, NO-THIN-AIR, OBSERVATION and PROPAGATION can be performed by one single check, here called “EXTERNAL”.
ARMv8 model: aarch64.cat from herd distribution.

irreflexive po;com+ as internal
empty rmw & (fre;coe) as atomic

let lob =  # locally ordered before, aka inclusive ppo
        ... 
        | [M];po-loc;[W]  # same as fri|coi

let obs =  # 'external' observation
          rfe|fre|coe

let rec ob = # ordered-before, aka happens-before
             obs
             | lob
             | ob; ob  # Recursive formulation for transitive closure

irreflexive ob as 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 — STLR, 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.
Bug or feature?

Once we have a model or while looking for it... The following execution: 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 as easily.

- Definitively a hardware anomaly.
- Not observed on ARMv8
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 C\( ^{\textit{AT}} \) 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
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

- The C11 model have evolved since first release, some points (essentially NO-THIN-AIR) still debated.

We present “Repaired C11”

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

Interestingly, “coherence” above regroups both \textsc{uniproc} (sb included in hb) and generalised \textsc{observation} (communication vs. hb).
Out of thin-air values cannot be neglected

- If any value can pop-up at any time no program proof is possible.
- Allowing **LB+datas** over non-atomics (for instance) hinders the DRF theorem.
- Out-of-thin-air values are not precisely defined, partly because dependencies are difficult to define in a (optimised) programming language.

```c
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);
```

---

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

Allow `x=42`, `y=42`? (include sophisticated, a.k.a “semantical” control dependencies definition in `hb`) Forbid? (hinders optimisation?)
RC11 radical stance against out-of-thin-air

Forbid any “LB” shape.

acyclic sb | rfe as no-thin-air

To be compared with machine level NO-THIN-AIR

acyclic ppo | fence | rfe as no-thin-air

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.
Restoring SC: the big deal of RC11

For SC atomics:

let sb-xy = sb \ loc # sb, different locations  
# SC-before  
let scb = sb | sb-xy; hb; sb-xy | hb&loc | co | fr

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
  acyclic [F]; hb; eco; sb; [F] # or acyclic eco; sb; [F]; hb

- C11 fence semantics significantly strengthened w.r.t. previous models.

- Complex definition of scb. Weaker than simply including hb in scb.  
  But then, SC atomics can be compiled by using hardware fences.
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

