Not so practical multicore programming

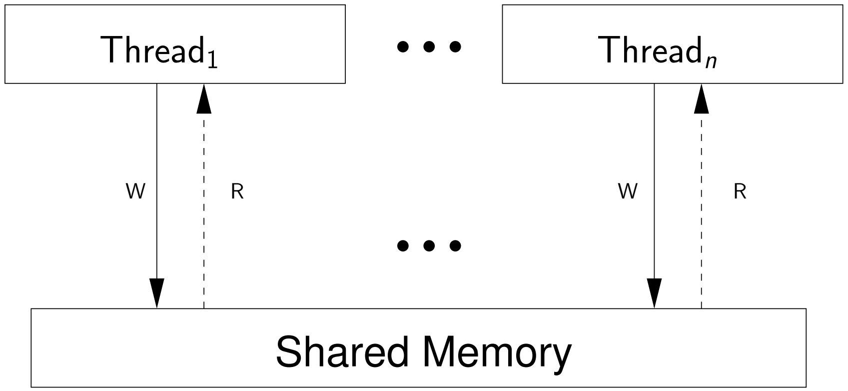
A simple model for sequential consistency, extended. . .

Luc Maranget Luc.Maranget@inria.fr

Part 1.

Axiomatic Sequential Consistency

Shared memory computer



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):$\mathbf{d}[\ell]v$, where:

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

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

```
/* 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 :

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

A definition of SC

A transcription of L. Lamport's definition.

Definition (SC 1)

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

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"):
$$\xrightarrow{\text{rf}_<} \overset{\text{Def}}{=} \left\{(w,r) | w = \max_<(w', \text{loc}(w') = \text{loc}(r) \wedge w' < r)\right\}.$$

Example of a question on SC

R

T_0	T_1
(a) x ← 1	(c) y ← 2
(b) y ← 1	(d) r0 ← x

Observed? y=2; r0=0

How do we know? Let us enumerate all interleavings:

a, b, c, d	y=2; r0=1;
a, c, b, d	y=1; r0=1;
a, c, d, b	y=1; r0=1;
c, d, a, b	y=1; r0=0;
c, a, b, d	y=1; r0=1;
c, a, d, b	y=1; r0=1;

Remark: if $b < c$ then y=2, if $a < d$ then r0=1.

Let us be a bit more clever

R

T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `2`
(b) `y` $\leftarrow$ `1`	(d) `r0` $\leftarrow$ `x`

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** ($\xrightarrow{\textbf{rf}}$)**:** 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** ($\xrightarrow{\textbf{co}}$)**:** Relates write events to the same location.

  For any location ℓ, the restriction of $\xrightarrow{\text{co}}$ to write events to location ℓ ($\mathbf{W}_\ell$) is a total strict order.

Coherence as a characteristics of shared memory

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

$$Wx0 \xrightarrow{\text{co}} Wx1 \xrightarrow{\text{co}} \boxed{x = 2} \xrightarrow{\text{co}} Wx3 \xrightarrow{\text{co}} \cdots$$

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

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

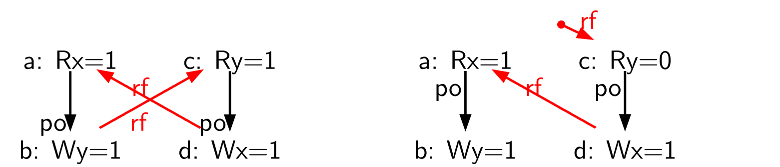
LB

T_0	T_1
(a) `r0` $\leftarrow$ `x`	(c) `r1` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `x` $\leftarrow$ `1`

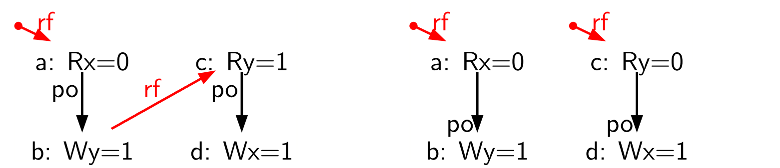
Observe: `r0; r1;`

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

`r0=1; r1=1;` | `r0=1; r1=0;`



`r0=0; r1=1;` | `r0=0; r1=0;`

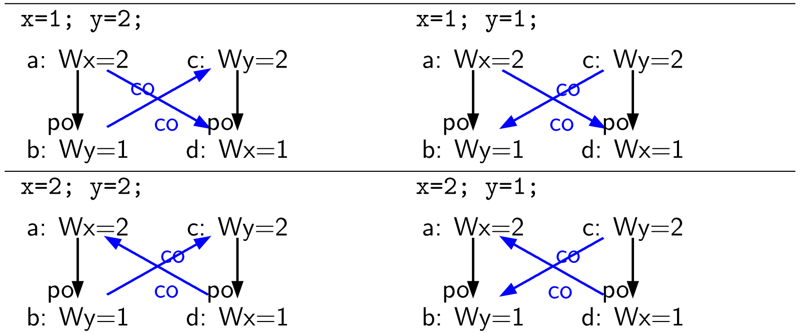


Example of $\xrightarrow{\text{co}}$

2+2W

T_0	T_1
$(a)\ \texttt{x} \leftarrow 2$	$(c)\ \texttt{y} \leftarrow 2$
$(b)\ \texttt{y} \leftarrow 1$	$(d)\ \texttt{x} \leftarrow 1$

Observed? `x=2; y=2;`



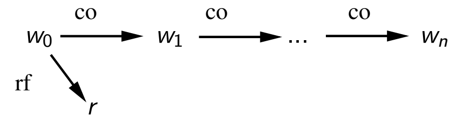
Notice: In this simple case of two stores, the value finally observed in locations determines $\xrightarrow{\text{co}}$ for them.

One more relation: $\xrightarrow{\text{fr}}$

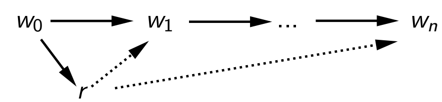
The new relation $\xrightarrow{\text{fr}}$ (from read) relates reads to "younger writes" (younger w.r.t. $\xrightarrow{\text{co}}$).

$$r \xrightarrow{\text{fr}} w \quad \stackrel{\text{Def}}{=} \quad w' \xrightarrow{\text{rf}} r \wedge w' \xrightarrow{\text{co}} w$$

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



We have



(Or: $\xrightarrow{\text{fr}} \stackrel{\text{Def}}{=} \left(\xrightarrow{\text{rf}}\right)^{-1} ; \xrightarrow{\text{co}}$)

Playing with $\xrightarrow{\text{fr}}$

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

MP

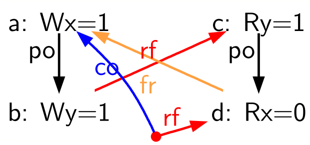
T_0	T_1
$(a)\ \texttt{x} \leftarrow 1$	$(c)\ \texttt{r0} \leftarrow \texttt{y}$
$(b)\ \texttt{y} \leftarrow 1$	$(d)\ \texttt{r1} \leftarrow \texttt{x}$

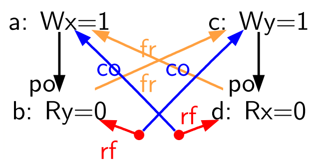
Observed? `r0=1; r1=0`

SB

T_0	T_1
$(a)\ \texttt{x} \leftarrow 1$	$(c)\ \texttt{y} \leftarrow 1$
$(b)\ \texttt{r0} \leftarrow \texttt{y}$	$(d)\ \texttt{r1} \leftarrow \texttt{x}$

Observed? `r0=0; r1=0`





Second definition of SC

Definition (SC 2)

An execution is SC when:

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

And of course:

Theorem

The two definitions of SC are equivalent.

SC 1 $\implies$ SC 2

Assume the existence of the total order "$<$".
Define:

$$\xrightarrow{\text{co}} \stackrel{\text{Def}}{=} \{(w_1, w_2) | \text{loc}(w_1) = \text{loc}(w_2) \wedge w_1 < w_1\},$$

Notice that $\xrightarrow{\text{rf}}$ is already defined: $\xrightarrow{\text{rf}} \stackrel{\text{Def}}{=} \xrightarrow{\text{rf}_<}$. Also notice $\xrightarrow{\text{po}} \subseteq <$, $\xrightarrow{\text{co}} \subseteq <$ and $\xrightarrow{\text{rf}} \subseteq <$.

Proof:

Define $\xrightarrow{\text{fr}} \stackrel{\text{Def}}{=} \xrightarrow{\text{rf}}{}^{-1} ; \xrightarrow{\text{co}}$, and prove $\xrightarrow{\text{fr}} \subseteq <$.

Let $r \xrightarrow{\text{fr}} w$. Let further $w_0 \xrightarrow{\text{rf}_<} r$, then, by definition of $\xrightarrow{\text{fr}}$, we have $w_0 \xrightarrow{\text{co}} w$ and thus $w_0 < w$.
But, w_0 is maximal amongst all $w' < r$. That is: "$w < r \implies w \leq w_0$" or, "$w_0 < w \implies r < w$" QED,

Hence, a cycle in $\xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{po}}$ would be a cycle in order "$<$"

SC 2 $\implies$ SC 1

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

From $<$ define $\xrightarrow{\text{rf}_<}$:

$$\xrightarrow{\text{rf}_<} \stackrel{\text{Def}}{=} \left\{(w, r) | w = \max_<(w', \text{loc}(w') = \text{loc}(r) \wedge w' < r)\right\}.$$

and show $\xrightarrow{\text{rf}} = \xrightarrow{\text{rf}_<}$.

1. Let $w_0 \xrightarrow{\text{rf}} r$ and let $w \in \mathbf{W}_\ell, w \neq w_0$ then ($\xrightarrow{\text{co}}$ total order on $\mathbf{W}_\ell$):
   1. Either $w \xrightarrow{\text{co}} w_0$ and $w < w_0 < r$.
   2. Or, $w_0 \xrightarrow{\text{co}} w$, and $r \xrightarrow{\text{fr}} w$, and thus $r < w$.

   Finally $w_0 \xrightarrow{\text{rf}_<} r$.
2. Let $w \stackrel{\text{rf}}{\not\longrightarrow} r$ (*i.e.* $w \in \mathbf{W}_\ell, w \neq w_0$), then
   1. Either $w \xrightarrow{\text{co}} w_0$, and thus ($\xrightarrow{\text{co}} \subseteq <$) $w \stackrel{\text{rf}_<}{\not\longrightarrow} r$.
   2. Or $w_0 \xrightarrow{\text{co}} w$, thus $r \xrightarrow{\text{fr}} w$, and thus ($\xrightarrow{\text{fr}} \subseteq <$) $w \stackrel{\text{rf}_<}{\not\longrightarrow} r$.

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

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

Introducing herd, a memory model simulator

A model `sc.cat`:

```
% 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
```

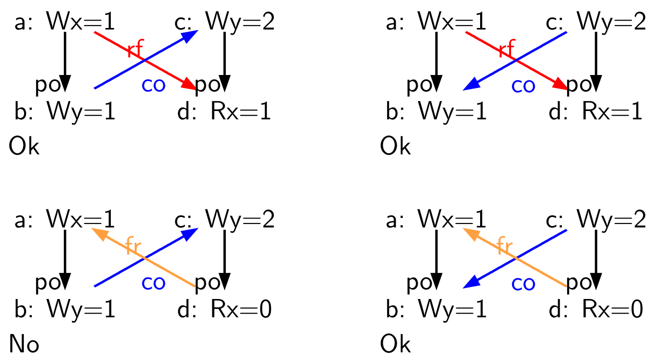
Running **R** on SC (demo in `demo/herd`):

```
% 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 $\xrightarrow{po}$, $\xrightarrow{rf}$ and $\xrightarrow{co}$ ($\xrightarrow{fr}$ deduced):



- Apply model checks to each candidate execution.

Part 2.

Studying Non-Sequentially Consistent Executions.

Violations of SC

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

However, the cycle does not describe:

- How many threads are involved.
- How many memory locations are involved.

We now aim at:

- Extract a subset of *significant* cycles.
- Generate *one* program out of one cycle.

Simplifying cycles: $\xrightarrow{po}$ and $\xrightarrow{\widehat{com}}$ steps alternate

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

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

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

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

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

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

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

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

- Either $e_1 \xrightarrow{po} e_2$ and we consider this $\xrightarrow{po}$ step in the cycle, in place of the $\xrightarrow{\widehat{com}}$ step (further merging $\xrightarrow{po}$ steps to get a smaller cycle).
- Or $e_2 \xrightarrow{po} e_1$, then we have a very simple cycle $e_2 \xrightarrow{po} e_1 \xrightarrow{\widehat{com}} e_2$. Such cycles are "*violations of coherence*" (more on them later).
- Case $e1 = e2$ is impossible ($\xrightarrow{com}$ is acyclic, see later)

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

Simplifying cycles – Threads

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

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

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

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

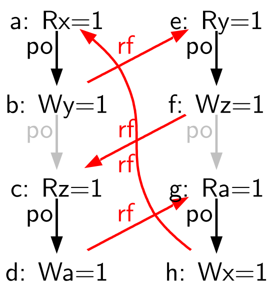
(Conclude with $\xrightarrow{po}$ being transitive)
If $e_1 = e_3$, we also have a simpler cycle:

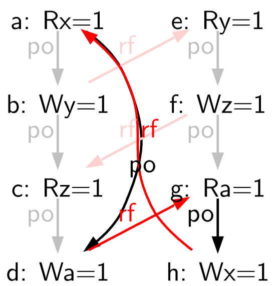
$$e_1 \xrightarrow{po} e_2 (\xrightarrow{\widehat{com}}; \xrightarrow{po})^*; \xrightarrow{\widehat{com}} e_3 = e_1$$

Conclusion: Cycle visit a thread at most once.

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



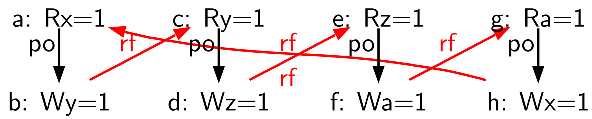


Generally: one passage per thread

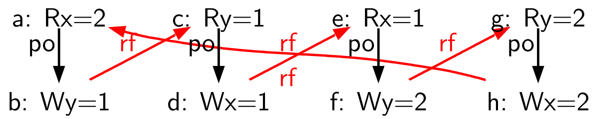
Test from cycles — Locations

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

- One interpretation (four locations):

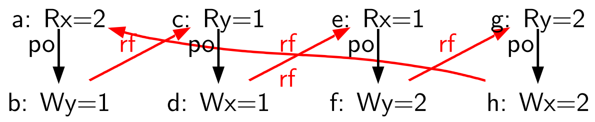


- Another interpretation (two locations):



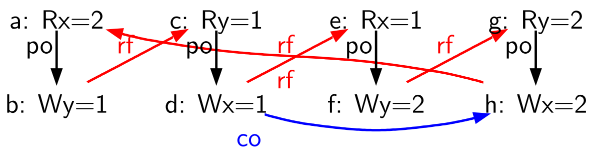
The second interpretation is not "minimal"

Reminding the interpretation with two locations:



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

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



We have a smaller cycle: $d \xrightarrow{\text{co}} h \xrightarrow{\text{rf}} a \xrightarrow{\text{po}} b \xrightarrow{\text{rf}} c \xrightarrow{\text{po}} d$.
Choosing Wx2 $\xrightarrow{\text{co}}$ Wx1 would yield another smaller cycle.
Generally: do not repeat locations in cycles.

Simplifying cycles, a lemma

Lemma (Identical locations)

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

1. *either $e_1 \xrightarrow{\widehat{com}} e_2$,*
2. *or $e_2 \xrightarrow{\widehat{com}} e_1$,*
3. *or $w \xrightarrow{rf} e_1$ and $w \xrightarrow{rf} e_2$.*

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.

- If e_1 and e_2 are from different threads. By hypothesis, e_1 and e_2 are related by complex steps (*i.e.* at least one $\xrightarrow{\text{po}}$ and one $\xrightarrow{\widehat{\text{com}}}$) in both directions. By the identical locations lemma:
  - Either, $e_1 \xrightarrow{\widehat{\text{com}}} e_2$ or $e_2 \xrightarrow{\widehat{\text{com}}} e_1$, and we have a simpler cycle.
  - or, $w \xrightarrow{\text{rf}} e_1$ and $w \xrightarrow{\text{rf}} e_2$, — see next page!.
- If e_1 and e_2 are from the same thread, *i.e.* for instance $e_1 \xrightarrow{\text{po}} e_2$, while e_2 relates to e_1 by complex steps:
  - either $e_1 \xrightarrow{\widehat{\text{com}}} e_2$ and we replace the $\xrightarrow{\text{po}}$ step in cycle, yielding a simpler cycle (one ($\xrightarrow{\text{po}}; \xrightarrow{\widehat{\text{com}}}$) step less)
  - or $e_2 \xrightarrow{\widehat{\text{com}}} e_2$ and we have a very simple cycle $e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\widehat{\text{com}}} e_1$.
  - Or $w \xrightarrow{\text{rf}} e_1$ and $w \xrightarrow{\text{rf}} e_2$, we short-circuit the cycle — as the cycle must be $\cdots w \xrightarrow{\text{rf}} e_1 \xrightarrow{\text{po}} e_2 \cdots$, which we reduce into $\cdots w \xrightarrow{\text{rf}} e2 \cdots$.

Next page

So let us assume a cycle that includes r_1 and r_2, related in both directions by complex steps and such that $w \xrightarrow{\text{rf}} r_1$ and $w \xrightarrow{\text{rf}} r_2$. We consider:

- If $w \xrightarrow{\text{rf}} r_1$ is in cycle, then there is an obvious short-circuit: replace $\xrightarrow{\text{rf}}$ followed by the complex steps from r_1 to r_2 by a single $w \xrightarrow{\text{rf}} r_2$ step.
- If $w \xrightarrow{\text{rf}} r_2$ is in cycle, symmetrical case.
- Otherwise, it must be that both r_1 and r_2 are the target of $\xrightarrow{\text{po}}$ steps and the source of $\xrightarrow{\text{fr}}$ steps: let w_1 and w_2 be the targets of those steps.
  Then, in all possible three situations: $w_1 = w_2$, $w_1 \xrightarrow{\text{co}} w_2$ and $w_2 \xrightarrow{\text{co}} w_1$ we construct a simpler cycle that does not contain r_1 or r_2.

… Simplifying cycles — Conclusion

In a non SC execution we find:

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

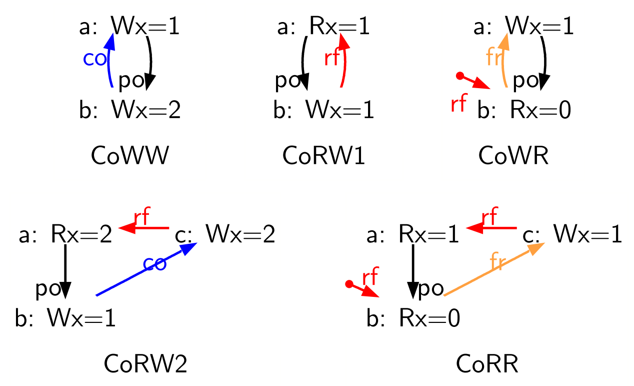
  Notice: By the last condition, such cycles have four steps or more and pass through two threads or more.

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

Violations of coherence

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

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



Application, all possible SC violations on two threads

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

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

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`.

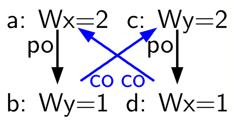
```
% diy7 -conf 2.conf
Generator produced 6 tests
% ls
2+2W.litmus  2.conf  @all  LB.litmus
MP.litmus  R.litmus  SB.litmus  S.litmus
% diy7 -conf 4.conf
Generator produced 68 tests...
```

Three violations of SC

2+2W

T_0	T_1
(a) `x` $\leftarrow$ 2	(c) `y` $\leftarrow$ 2
(b) `y` $\leftarrow$ 1	(d) `x` $\leftarrow$ 1

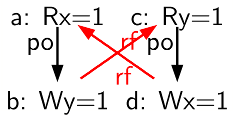
Observed? `x=2; y=2;`



LB

T_0	T_1
(a) `r0` $\leftarrow$ `x`	(c) `r1` $\leftarrow$ `y`
(b) `y` $\leftarrow$ 1	(d) `x` $\leftarrow$ 1

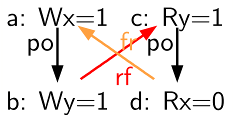
Observe: `r0=1; r1=1;`



MP

T_0	T_1
(a) `x` $\leftarrow$ 1	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ 1	(d) `r1` $\leftarrow$ `x`

Observed? `r0=1; r1=0`

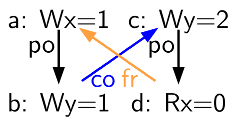


Three more violations of SC

R

T_0	T_1
(a) `x` $\leftarrow$ 1	(c) `y` $\leftarrow$ 2
(b) `y` $\leftarrow$ 1	(d) `r0` $\leftarrow$ `x`

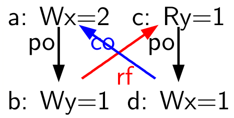
Observed? `y=2; r0=0`



S

T_0	T_1
(a) `x` $\leftarrow$ 2	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ 1	(d) `x` $\leftarrow$ 1

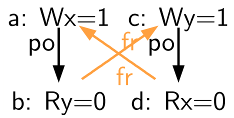
Observed? `x=2; r0=1`



SB

T_0	T_1
(a) `x` $\leftarrow$ 1	(c) `y` $\leftarrow$ 1
(b) `r0` $\leftarrow$ `y`	(d) `r1` $\leftarrow$ `x`

Observed? `r0=0; r1=0`



Application

We assume the following on modern shared memory architectures:

- No valid execution includes a violation of coherence.
- No valid execution includes a cycle whose $\xrightarrow{po}$ steps include the adequate fence instruction between source and target instructions.
- The full memory barrier is always adequate.

To guarantee SC:

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

Simplification:

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

Optimisation

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

A semi realistic example

```
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.

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

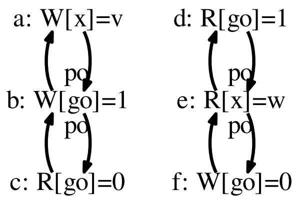
```
int sum = 0 ;
for (int k = N ; k >= 0 ; k--) {
d:  while (go == 0) ;
    sync() ;
e:  int t = x; sum += t;
    sync() ;
f:  go = 0 ;
}
```

A semi realistic example, more precise fencing

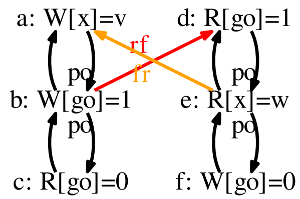
```
for (int k = N ; k >= 0 ; k--) {
a: x = k ;
b: go = 1 ;
c: while (go == 1) ;
}
```

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

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



Cycle 1

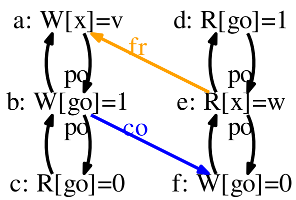


Analysis based upon Sekar *et al.* Power model (PLDI'11). Test **MP**

$$a \xrightarrow{\text{lwsync}} b,\ d \xrightarrow{\text{ctrlisync}} e$$

X86: no fence needed.

Cycle 2

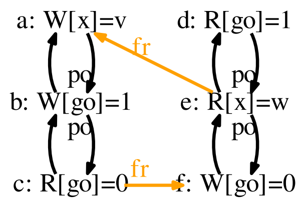


Analysis based upon Sekar *et al.* Power model (PLDI'11). Test **R**

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

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

Cycle 3

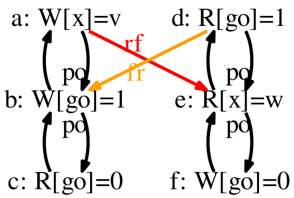


Analysis based upon Sekar *et al.* Power model (PLDI'11). Test **SB**

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

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

Cycle 4

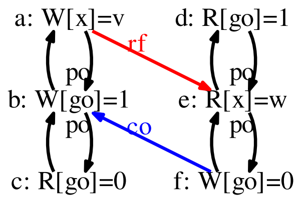


Analysis based upon Sekar *et al.* Power model (PLDI'11). Test **MP**

$$b \xrightarrow{\text{lwsync}} a,\ e \xrightarrow{\text{ctrlisync}} d$$

X86: no fence needed.

Cycle 5

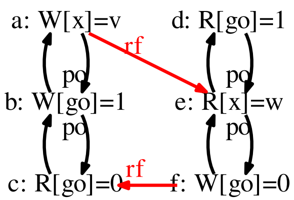


Analysis based upon Sekar *et al.* Power model (PLDI'11). Test **S**

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

X86: no fence needed.

Cycle 6



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

$$c \xrightarrow{\text{ctrl}} a,\ e \xrightarrow{\text{ctrl}} f$$

X86: no fence needed.

Sufficient fencing, X86

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

```
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.

Sufficient fencing, Power

$$
\begin{gathered}
a \xrightarrow{\text{lwsync}} b,\ d \xrightarrow{\text{ctrlisync}} e, \\
a \xrightarrow{\text{sync}} b,\ f \xrightarrow{\text{sync}} e, \\
a \xrightarrow{\text{sync}} c,\ f \xrightarrow{\text{sync}} e, \\
b \xrightarrow{\text{lwsync}} a,\ e \xrightarrow{\text{ctrlisync}} d, \\
b \xrightarrow{\text{lwsync}} a,\ e \xrightarrow{\text{ctrl}} f, \\
c \xrightarrow{\text{ctrl}} a,\ e \xrightarrow{\text{ctrl}} f
\end{gathered}
$$

```
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

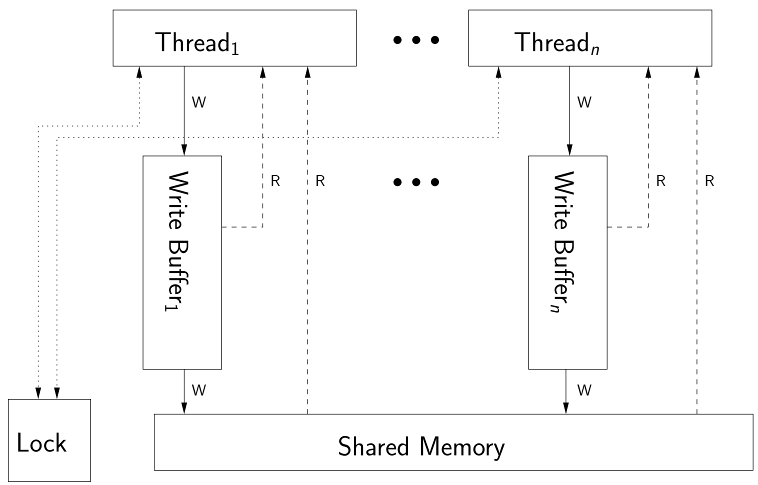
```
inline static void sync() {
  asm __volatile__ ("sync" ::: "memory") ;
}

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

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

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

Part 3.

Axiomatic TSO

TSO — The Model of X86 machines



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

An experimental study of x86

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

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

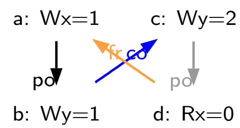
Running:

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

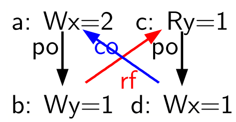
Analysis:

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

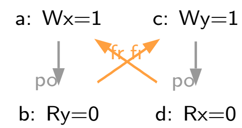

Results for running the six test on this machine



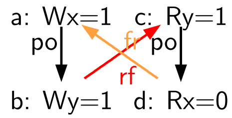
R: Ok



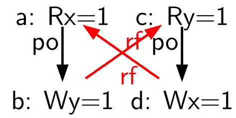
S: No



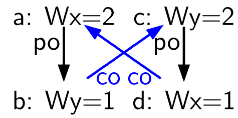
SB: Ok



MP: No



LB: No



2+2W: No

Axiomatic TSO, model TSO 1

- Remember SC:

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

  A model for herd, our generic simulator:

```
let ppo = po # ppo stands for 'preserved program-order'
let com-hb = fr | rf | co # All comunications 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.

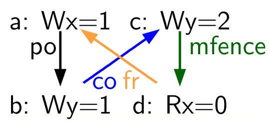
Restoring SC with `mfence`

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

R+po+mfence

T_0	T_1
(a) x ← 1 (b) y ← 1	(c) y ← 2 mfence (d) r0 ← x

Observed? y=2; r0=0

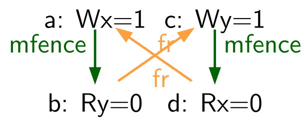


No

SB+mfences

T_0	T_1
(a) x ← 1 mfence (b) r0 ← y	(c) y ← 1 mfence (d) r1 ← x

Observed? r0=0; r1=0



No

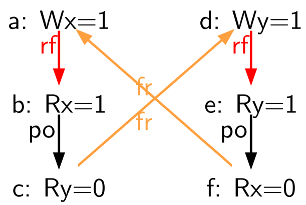
Our TSO 1 model is wrong!

Consider:

SB+rfi-pos

T_0	T_1
(a) `x` $\leftarrow$ `1`	(d) `y` $\leftarrow$ `1`
(b) `r0` $\leftarrow$ `x`	(e) `r2` $\leftarrow$ `y`
(c) `r1` $\leftarrow$ `y`	(f) `r3` $\leftarrow$ `x`

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



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

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

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

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

Observation of **SB+rfi-pos**

Demo in `demo/TSO2`.

- Create test from cycle:

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

- Run test:

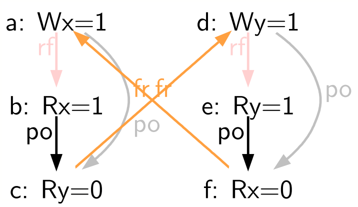
```
% litmus7 -mach x86.cfg src/SB+rfi-pos.litmus
...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for src/SB+rfi-pos.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
X86 SB+rfi-pos

 P0            | P1            ;
 MOV [x],$1    | MOV [y],$1    ;
 MOV EAX,[x]   | MOV EAX,[y]   ;
 MOV EBX,[y]   | MOV EBX,[x]   ;

exists (0:EAX=1 /\ 0:EBX=0 /\ 1:EAX=1 /\ 1:EBX=0)
...
Test SB+rfi-pos Allowed
Histogram (4 states)
12440 *>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=0;
3992819:>0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=0;
3994289:>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=1;
452   :>0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=1;
Ok
...
```


Corrected model: TSO 2

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

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

The new hb is no longer cyclic:



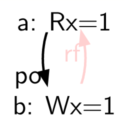
(Also consider that $a \xrightarrow{\text{po}}_{WR} c$ and $d \xrightarrow{\text{po}}_{WR} f$ are non-global.)

This is not over yet...

Our TSO 2 model:

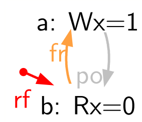
```
let ppo = (R*M | W*W) & po   # (W*R) & po absent
let com-hb = rfe | fr | co   # rfi absent
acyclic (ppo | com-hb | mfence) as hb
```

Allows two violations of coherence:



CoRW1

$\xrightarrow{\text{rfi}}$ not in $\xrightarrow{\text{hb}}$



CoWR

$W \xrightarrow{\text{po}} R$ not in $\xrightarrow{\text{hb}}$

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

A new check: UNIPROC

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

$$\text{Irreflexive}\left(\xrightarrow{\text{po-loc}}; \widehat{\xrightarrow{\text{com}}}\right)$$

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

```
let complus = rf | fr | co | (co;rf) | (fr;rf)
irreflexive (po-loc; complus) as uniproc
...
```

In the TSO case we can "optimise":

```
irreflexive rf;RW(po-loc)
irreflexive fr;WR(po-loc)
```

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

Our final TSO model

```
TSO3

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

let ppo = (R*M | W*W) & po  # (W*R) & po 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"'. (Jason F. Cantin, Mikko H. Lipasti, James E. Smith ACM Symposium on Parallel Algorithms and Architectures 2004).

Definition (Uniproc 1)

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

with $\xrightarrow{\text{com}} = \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}}$.

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)

$$\text{Irreflexive}\left(\xrightarrow{\text{po-loc}}; \widehat{\xrightarrow{\text{com}}}\right)$$

Definitions are equivalent.

Equivalence of uniproc definitions

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

Conversely, we use the "Identical locations" lemma.

Consider a cycle in $\xrightarrow{\text{po-loc}} \cup \xrightarrow{\text{com}}$, s.t. for all $e_1 \xrightarrow{\text{po}} e_2$ steps we do not have $e_2 \widehat{\xrightarrow{\text{com}}} e_1$. 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 $\xrightarrow{\text{po}}$ step, replacing $w \xrightarrow{\text{rf}} r_1 \xrightarrow{\text{po}} r_2$ by $w \xrightarrow{\text{rf}} r_2$.
- Or, $e_1 \widehat{\xrightarrow{\text{com}}} e_2$. We replace the $\xrightarrow{\text{po}}$ step by $\xrightarrow{\text{com}}$ steps.

As a result we have a cycle in $\xrightarrow{\text{com}}$, 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: `xchgl` r,x.

We further enforce:

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

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

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

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

- "Fence semantics": locked instructions act as fences.

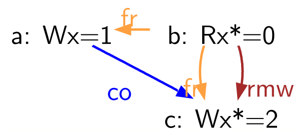
ATOM check

The ATOM check forbids this execution:

EXCH

T_0	T_1
(a) x $\leftarrow$ 1	r $\leftarrow$ 2 (b/c) r1 $\leftrightarrow$ x

Observed? r=0; y=2



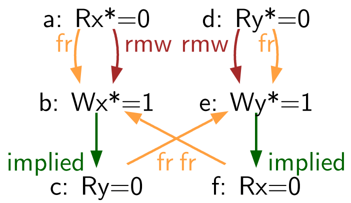
Implied fences

Implied fences forbid this execution

SB+EXCH

T_0	T_1
r $\leftarrow$ 1 (a/b) r $\leftrightarrow$ x (c) r0 $\leftarrow$ y	r $\leftarrow$ 1 (d/e) r $\leftrightarrow$ y (f) r1 $\leftarrow$ x

Observed? r0=0; r1=0



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

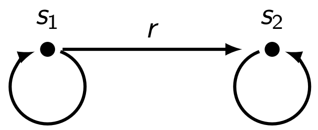
x86-TSO model for herd

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 $r\&(S_1{*}S_2)$ are equal.



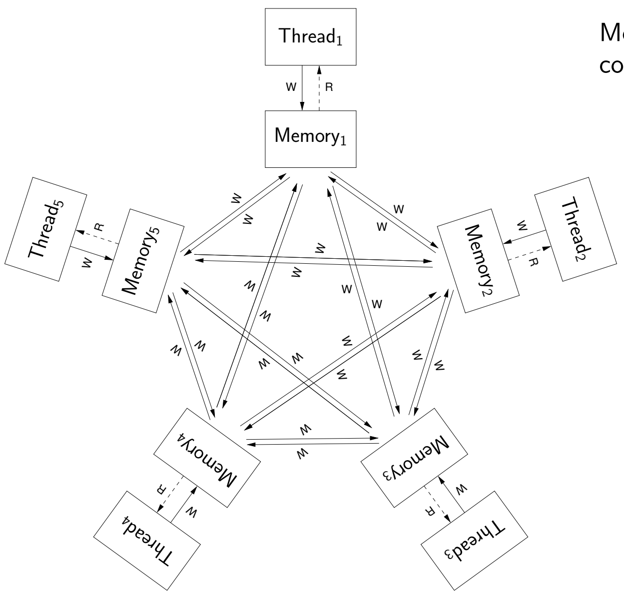
Then, for instance, we may reformulate TSO preserved program order as:

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


Part 4.

Axiomatic ARM/Power

A relaxed shared memory computer



More or less visible to user code:

- Cores:
  - Out of order execution
  - Branch speculation
  - Write buffers
- Memory
  - Physically distributed
  - Caches

Situation of (our) ARM/Power models

- **Architecture public reference** Informal, cannot clearly explain how fences restore SC for instance.
- **Operational model:** (PLDI'11) more precise, developped 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, $\xrightarrow{\text{ppo}}$ 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 `lwsync`) and full (Power `sync`).

Two-threads SC violation for ARM

Generating tests is as simple as:

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

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

```
Observation R Sometimes 5722 1994278
Observation MP Sometimes 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 $\xrightarrow{\text{ppo}}$ as simply as for TSO.

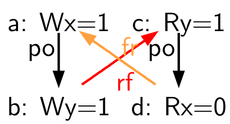
An experiment on ARM/Power

Consider test **MP**:

MP

T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `r1` $\leftarrow$ `x`

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:

`r1` $\leftarrow$ `x`
`r2` $\leftarrow$ `t[r1]`

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

Data dependency:

`r1` $\leftarrow$ `x`
`y` $\leftarrow$ `r1+1`

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

Control dependency: (+`isync`)

`r1` $\leftarrow$ `x`
if r1=0 then
  (`isync`)
  `y` $\leftarrow$ `1`

```
 lwz r1,0(r8)
 cmpwi r1,0
 bne L1
 (isync)
 li r2,1
 stw r2,0(r9)
L1:
```

Generating tests (ARM), yet another tool: diycross

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

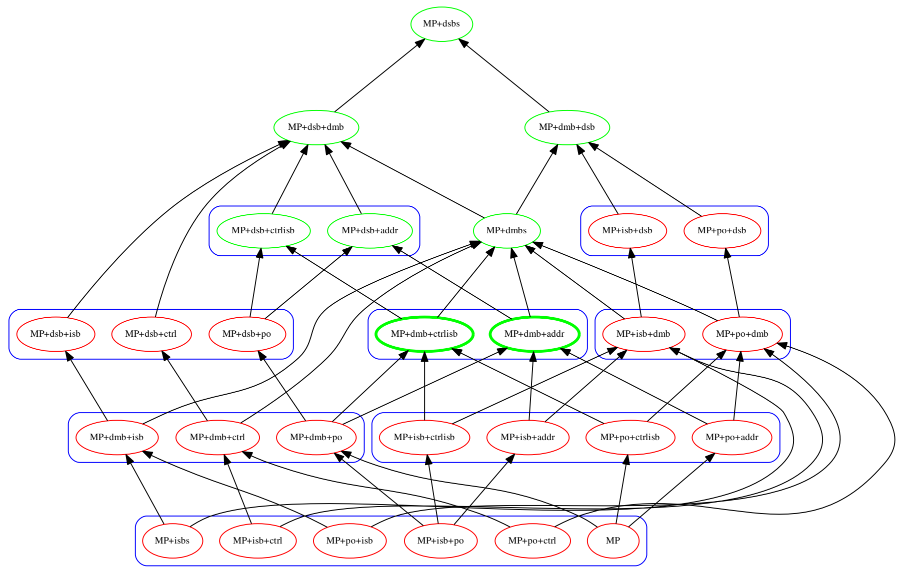
```
% diycross -arch ARM\
 PodWW,DMBdWW,DSBdWW,ISBdWW\
 Rfe\
 PodRR,DpCtrldR,DpCtrlIsbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR\
 Fre
Generator produced 28 tests
```

- One generates **MP** as `diyone PodWW Rfe PodRR Fre`
- `diycross` $r^1_1, \dots, r^1_{N_1} \ \cdots \ r^M, \dots, r^M_{N_M}$, generates the $N_1 \times \cdots \times N_M$ cycles $r^1_{k_1} \cdots r^\ell_{k_\ell} \cdots r^M_{k_M}$ by *cross-producting* the given edge list arguments.

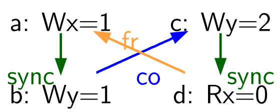
This generates some variations in the **MP** family.

We then compile and run, and…

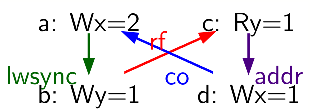
Optimal fencing/dependencies for MP



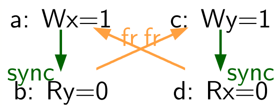
Optimal fencing for the 6 two-threads tests (Power)



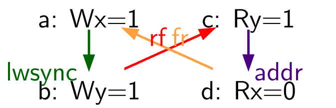
R+syncs



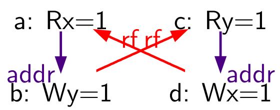
S+lwsync+addr



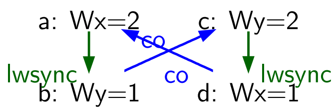
SB+syncs



MP+lwsync+addr



LB+addrs



2+2W+lwsyncs

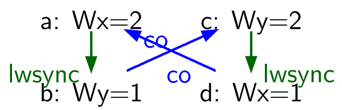
Some observations

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

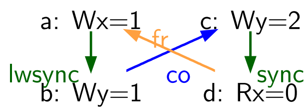
$$addr < \texttt{lwsync} < \texttt{sync}$$

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**.



2+2W+lwsyncs



R+lwsync+sync

No

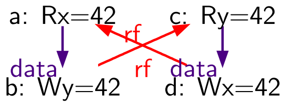
Ok

Dependencies are enough

CAUSAL

T_0	T_1
(a) `r0` ← `x` (b) `y` ← `r0`	(c) `r1` ← `y` (d) `x` ← `r1`

Observed? `r0=42; r1=42;`



LB+datas

Of course we never observe this behaviour (values out of thin air) and any (hardware) model should forbid it.
Happens-before If we order: (1) stores: the point in time when the value is made available to other threads (2) loads: the point when the value is read by core.

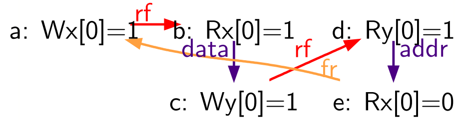
Dependencies from reads not always enough!

Consider test **WRC+data+addr**:

WRC+data+addr

T_0	T_1	T_2
(a) `x[0]` ← `1`	(b) `r0` ← `x` (c) `y` ← `r0`	(d) `r1` ← `y` `t` ← `r1&4` (e) `r2` ← `x[t]`

Observed? `r0=1; r2=0;`



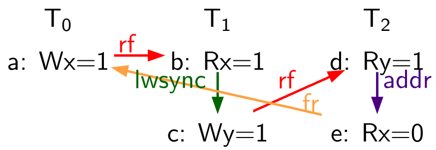
WRC+data+addr

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

Stores are not "multi-copy atomic" T_0 and T_1 share a private buffer/cache/memory (*e.g.* a cache in SMT context). T_2 "*does not see*" the store by T_0, when T_1 does.

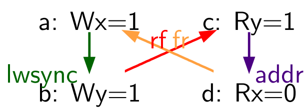
Restoring SC for **WRC**

Use a lightweight fence on T_1:



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**



MP+lwsync+addr

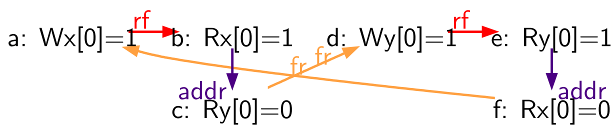
Another, symetric, case of insufficient dependencies

Consider test **IRIW+addrs**:

IRIW

T_0	T_1	T_2	T_3
(a) `x[0]` ← `1`	(b) `r0` ← `x[0]` `t` ← `r0^r0` (c) `r1` ← `y[t]`	(d) `y[0]` ← `1`	(e) `r2` ← `y[0]` `t` ← `r2^r2` (f) `r3` ← `x[t]`

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

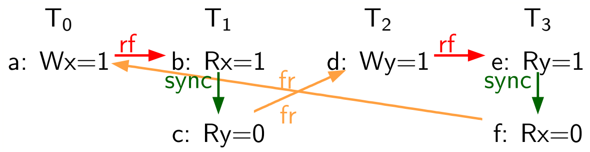


IRIW+addrs

Behaviour observed on Power (not on ARM, but documentation allows it).
Stores are not "multi-copy atomic": T_0 and T_1 have a private buffer/cache/memory, T_2 and T_3 also have one.

Restoring SC for **IRIW**

Use a full fence on T_1 and T_2:



IRIW+syncs

Propagation: Full fences order all communications.

Relation summary

Communication relations:

- Read-from: $w \xrightarrow{\text{rf}} r$, with $\text{loc}(w) = \text{loc}(r), \text{val}(w) = \text{val}(r)$.
- Coherence: $w \xrightarrow{\text{co}} w'$, with $\text{loc}(w) = \text{loc}(w') = x$. Total order for given x: hence "*coherence orders*".
- We deduce from-read: $r \xrightarrow{\text{fr}} w$, *i.e* $w' \xrightarrow{\text{rf}} r$ and $w' \xrightarrow{\text{co}} w$.
- We distinguish internal (same proc, $\xrightarrow{\text{rfi}}$, $\xrightarrow{\text{coi}}$, $\xrightarrow{\text{fri}}$) and external (different procs, $\xrightarrow{\text{rfe}}$, $\xrightarrow{\text{coe}}$, $\xrightarrow{\text{fre}}$) communications.

"Execution" relations

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

A model in four checks (TOPLAS'14)

UNIPROC

```
acyclic poloc | com as uniproc
```

NO-THIN-AIR

```
let fence = strong | light 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 **WRC+data+addr** and **IRIW+addrs** are forbidden (but **SB+rfi-addrs**, cf. slide 57, is still allowed).

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

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

ARMv8 model: `aarch64.cat` from herd distribution.

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

let lob =          # localy 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 forumulation for transitive closure

irreflexive ob as external
```


A few details

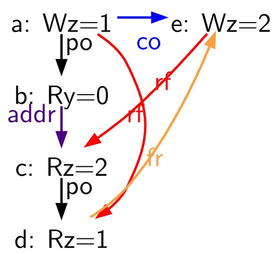
Armv8 features load-acquire instructions — two of them, Acquire (LDAR) and AcquirePC (LDAPR), events `A` and `Q`; and store-release instructions — STLR, events `L`.

```
(* 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.* $\xrightarrow{po}$ contradicts $\xrightarrow{fr};\xrightarrow{rf}$). **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 CAT memory-order specifications result in sets of events `RLX`, `ACQ`, ..., `SC`. Those events can be reads or writes (sets `R` and `W`) but also fences (set `F`).

Significant differences, w.r.t. hardware models

- No real preserved-program-order, as `po` is part of happens-before `hb`. Defining dependencies is impossible for the sake of compiler optimisations.
- As a result, general communications cannot be part of `hb`. If so we define SC!
- C favors atomic accesses overs fences. This resulted in (initial) weak semantics of SC fences.
- In case of data-race: undefined behaviour:

```
let conflict = ((W * _) | (_ * W)) & loc & ext
let dr = conflict \ (hb | hb^-1 | A * A)
 # A = atomic access

flag ~empty dr as DataRace
```

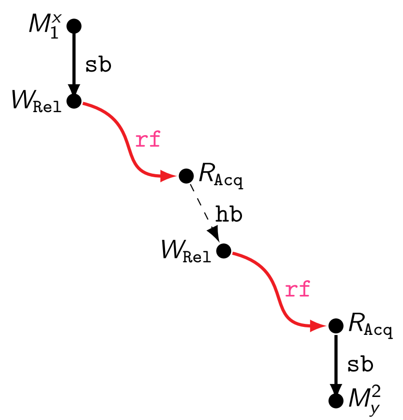
- The C11 model have evolved since first release, some points (essentially NO-THIN-AIR) still debated.

We present "Repaired C11"

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

(Repaired) C11 model. happens-before

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




```
(* rf from write release
   to read acquire *)
let sw = [REL];rf;[ACQ]

(* hb is a sequence of
   (alternating)
   sw and sb steps *)
let hb = (sb | sw)+
```

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 *relase-sequence*, `rs`:

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

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

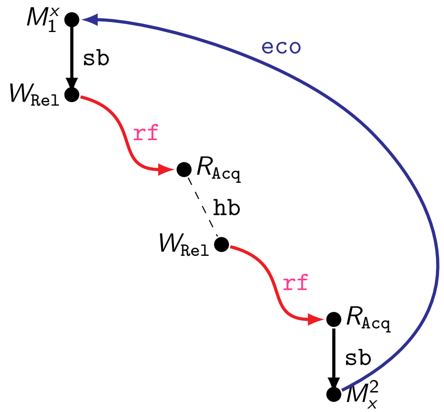
Then, full synchronise-with:

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

let hb = (sb | sw)+
```


RC11, "coherence" check

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

Interestingly, "`coherence`" above regroups both UNIPROC (`sb` included in `hb`) and generalised 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.

```
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) ;
```

```
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 correcteness (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.* $\xrightarrow{\text{ppo}}$, or the semantics of weak fences).

Some valuable readings

"A diy "Seven" tutorial" Jade Alglave, Luc Maranget. Sofware and documentation, http://diy.inria.fr/doc/index.html.

"Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory" Jade Alglave, Luc Maranget, Michael Tautschnig: ACM Trans. Program. Lang. Syst. 36(2): 7:1-7:74 (2014)

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