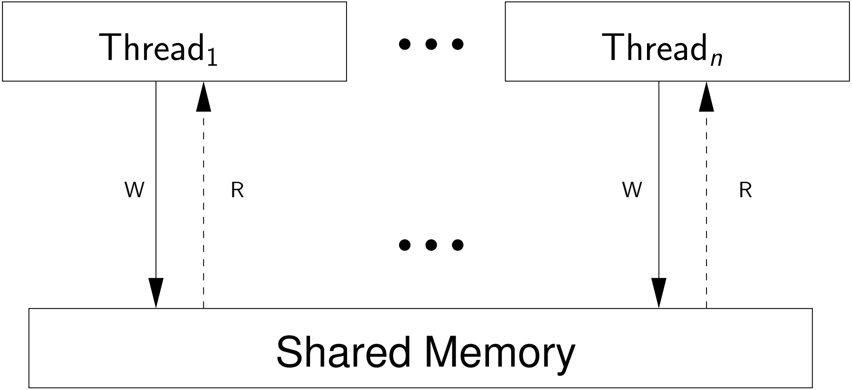
Testing (and defining) Weak Memory Models

Luc Maranget

Inria Paris-Rocquencourt

A simple shared memory computer



Threads execute programs as usual: instructions are executed completely and atomically (memory stores in particular).

The Sequentially Consistent Model (SC)

Definition by L. 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*".

(One may add "*stores take effect immediately*".)

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

A first, one expect shared memory multiprocessors to behave that way, which of course they don't.

Axiomatic or *post-mortem* semantics — Events

The effects of "*operations executed by the processors*" are represented by *events*. We define *memory events* (a):$\mathbf{d}[\ell]v$:

- Unique label typically (a), (b), etc.
- Direction $\mathbf{d}$, 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 (often omitted)

The program order $\xrightarrow{\text{po}}$ ("*order specified by program*") is a linear order amongst the events originating from the same thread.

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

Example of program order

Despite its name, program order is a dynamic notion.

```
/* 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 :

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

Notice: program order (and events) may depend on the values of the reads, *i.e.* on values written by other threads (**if** . . .).
In simple examples, program order is given by program text.

Relating writes and reads

We define a first "communication relation" between events with the same location.

Definition (Read-from $\xrightarrow{\text{rf}}$)

Relates write events to read events that read the stored value (implicit initial writes).

1. Existence and unicity:

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

2. Same location, same value:

$$\mathsf{loc}(w) = \mathsf{loc}(r) \land \mathsf{val}(w) = \mathsf{val}(r).$$

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

LB

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

Observe: `r0`; `r1`;

(Intial values of `x` and `y` are 0.)

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

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

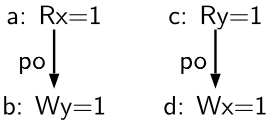
LB

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

Observe: `r0`; `r1`;

(Intial values of `x` and `y` are 0.)

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



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

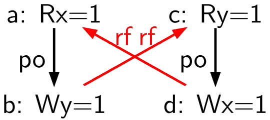
Example of $\xrightarrow{\text{rf}}$

LB

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

Observe: `r0`; `r1`;

(Intial values of `x` and `y` are 0.)

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

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

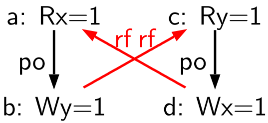
LB

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

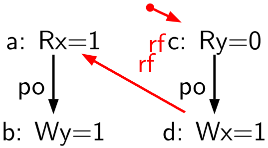
Observe: `r0; r1;`

(Intial values of `x` and `y` are 0.)

`r0=1; r1=1;`



`r0=1; r1=0;`



`r0=0; r1=1;`

`r0=0; r1=0;`

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

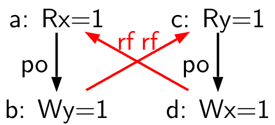
LB

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

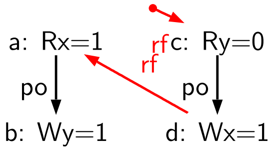
Observe: `r0`; `r1`;

(Intial values of `x` and `y` are 0.)

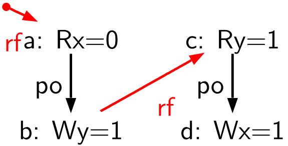
`r0=1; r1=1;`



`r0=1; r1=0;`



`r0=0; r1=1;`



`r0=0; r1=0;`

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

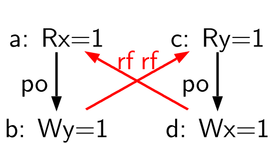
LB

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

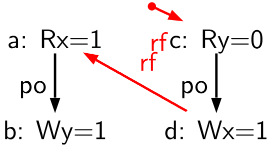
Observe: `r0; r1;`

(Intial values of `x` and `y` are 0.)

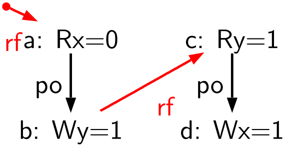
`r0=1; r1=1;`



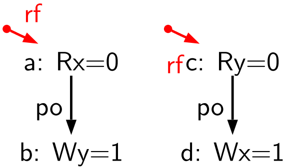
`r0=1; r1=0;`



`r0=0; r1=1;`



`r0=0; r1=0;`



A definition of SC

Definition (SC 1)

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

1. Order $<$ is compatible with program order:

$$e_1 \xrightarrow{\text{po}} e_2 \implies e_1 < e_2.$$

2. A Read r reads from the recent write before r in $<$.

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

A question on SC

Program:

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`

How can we know? Let us enumerate all interleavings.

a, b, c, d |

A question on SC

Program:

R

T_0	T_1
$(a)\,\texttt{x} \leftarrow 1$ $(b)\,\texttt{y} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 2$ $(d)\,\texttt{r0} \leftarrow \texttt{x}$

Observed? `y=2; r0=0`

How can we know? Let us enumerate all interleavings.

a, b, c, d	$\texttt{y}{=}2; \texttt{r0}{=}1;$
a, c, b, d	

A question on SC

Program:

R

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

Observed? `y=2; r0=0`

How can 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	

A question on SC

Program:

R

T_0	T_1
(a) $\texttt{x} \leftarrow 1$	(c) $\texttt{y} \leftarrow 2$
(b) $\texttt{y} \leftarrow 1$	(d) $\texttt{r0} \leftarrow \texttt{x}$
Observed? y=2; r0=0	

How can 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	

A question on SC

Program:

R

T_0	T_1
$(a)\,\texttt{x} \leftarrow 1$ $(b)\,\texttt{y} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 2$ $(d)\,\texttt{r0} \leftarrow \texttt{x}$

Observed? `y=2; r0=0`

How can we know? Let us enumerate all interleavings.

a, b, c, d	$\texttt{y}{=}2; \texttt{r0}{=}1;$
a, c, b, d	$\texttt{y}{=}1; \texttt{r0}{=}1;$
a, c, d, b	$\texttt{y}{=}1; \texttt{r0}{=}1;$
c, d, a, b	$\texttt{y}{=}1; \texttt{r0}{=}0;$
c, a, b, d	

A question on SC

Program:

R

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

Observed? `y=2; r0=0`

How can 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	

A question on SC

Program:

R

T_0	T_1
(a) `x` $\leftarrow$ 1 (b) `y` $\leftarrow$ 1	(c) `y` $\leftarrow$ 2 (d) `r0` $\leftarrow$ `x`
Observed? `y=2; r0=0`	

How can 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;

Conclusion: No SC execution would ever yield the output "`y=2; r0=0;`".

Modern machines are not SC, how can we know?

```
void *T0(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  q->x = 1 ;
  q->y = 1 ;
  return NULL ;
}
```

```
void *T1(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  q->y = 2 ;
  int r0 = q->x ;
  q->r0 = r0 ;
  return NULL ;
}
```

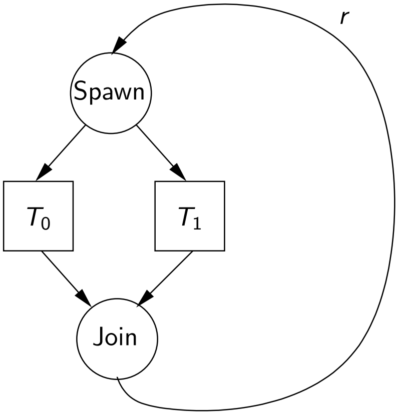
Modern machines are not SC, how can we know?

```
void *T0(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  q->x = 1 ;
  q->y = 1 ;
  return NULL ;
}
```

```
void *T1(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  q->y = 2 ;
  int r0 = q->x ;
  q->r0 = r0 ;
  return NULL ;
}
```

```
 for ( ; ; ) {
// Initialise
   common_t c ; c.x = c.y = 0 ; ctx_t a0,a1 ;
// Run
   a0.id = 0 ; a0.common = &c ; create_thread(&th0,T0,&a0) ;
   a1.id = 1 ; a1.common = &c ; create_thread(&th1,T1,&a1) ;
   join_thread(&th0) ;
   join_thread(&th1) ;
// Collect results
   ... c.y ... c.r0
 }
```

Naive testing, graphically



Let us run test **R** on this machine (`demo/01/naive_r.out`)

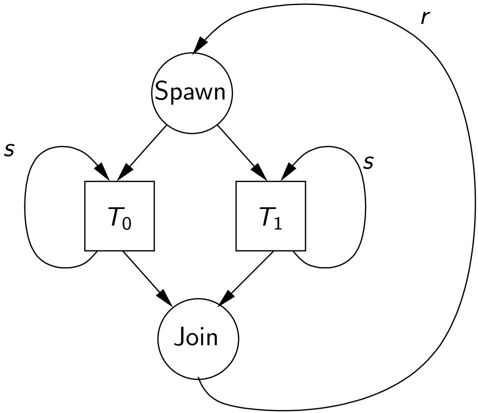
Minimizing the impact of thread creation on cost

We perform `size` tests per thread creation, in arrays.

```
void *T0(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  for
   (int k = 0 ;
    k < q->size ;
    k++) {
      q->x[k] = 1 ;
      q->y[k] = 1 ;
  }
  return NULL ;
}
```

```
void *T1(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  for
   (int k = 0 ;
    k < q->size ;
    k++) {
      q->y[k] = 2 ;
      int r0 = q->x ;
      q->r0[k] = r0 ;
  }
  return NULL ;
}
```

Graphically



Let us run test **R** on this machine (`demo/01/loop_r.out`)

Let us synchronise iterations

```
void *T0(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  for
   (int k = 0 ;
    k < q->size ;
    k++) {
wait_partner(&q->sync[k],k,0) ;
      q->x[k] = 1 ;
      q->y[k] = 1 ;
  }
  return NULL ;
}
```

```
void *T1(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  for
   (int k = 0 ;
    k < q->size ;
    k++) {
wait_partner(&q->sync[k],k,1) ;
      q->y[k] = 2 ;
      int r0 = q->x ;
      q->r0[k] = r0 ;
  }
  return NULL ;
}
```

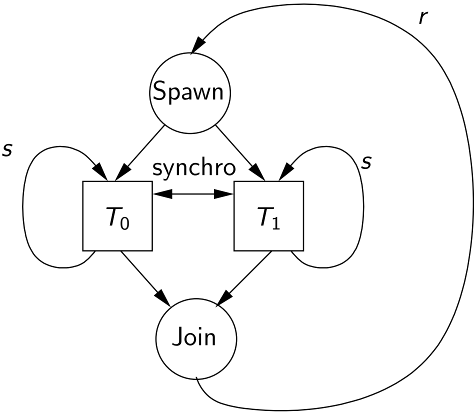
Let us synchronise iterations

```
void *T0(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  for
   (int k = 0 ;
    k < q->size ;
    k++) {
wait_partner(&q->sync[k],k,0) ;
      q->x[k] = 1 ;
      q->y[k] = 1 ;
  }
  return NULL ;
}
```

```
void *T1(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  for
   (int k = 0 ;
    k < q->size ;
    k++) {
wait_partner(&q->sync[k],k,1) ;
      q->y[k] = 2 ;
      int r0 = q->x ;
      q->r0[k] = r0 ;
  }
  return NULL ;
}
```

```
inline static void wait_partner(volatile int *p,int k,int id) {
  if (k % 2 == id) {
    *p = 1 ; __sync_synchronize() ; // Well...
  } else {
    while (*p == 0) ;
  }
}
```

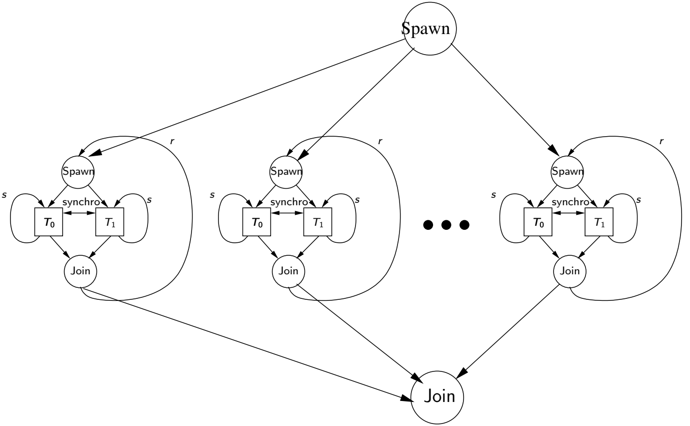
Graphically



Let us run test **R** on this machine (`demo/01/sync_r.out`)

Running n test instances at once

Why not run $n = \lfloor a/2 \rfloor$ instances of **R** on a a-core machine?



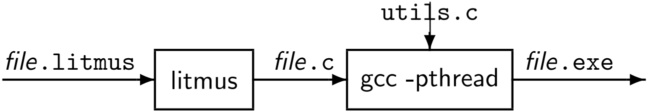
- Much more outcomes on many-core machines per test run
- Important when one pays resources by chunks of say 32 cores.
- Makes noise and favors outcome variability.

The litmus tool

- Tests are written in assembler, for precision.

```
X86 R
{ }
 P0          | P1           ;
 MOV [x],$1 | MOV [y],$2  ;
 MOV [y],$1 | MOV EAX,[x] ;
exists (y=2 /\ 1:EAX=0)
```

- Tests are compiled to C programs (with inline assembler):



Demo in `demo/02`:

```
% litmus -mach ./x86.cfg -o run R.litmus
% cd run
% make
% ./R.exe -v -v
```

Some litmus settings

- Synchronisation
  - Loose synchronisation (we saw it).
  - Exact synchronisation with polling synchronisation barriers and time base.
  - Other, less useful, modes: POSIX thread barrier based and no synchronisation at all.
- Affinity: force threads to run on designated cores, or let the OS scheduler perform its job.
- Prefetching of flushing cache lines.
  - Automatic, depending on test.
  - Random.
  - Complete or none.
- Various scanning order of location arrays.
  - Random (by the means of a shuffled arrray of pointer).
  - Linear with a stride.

A complete testing campaign usually involves trying many settings (for instance, testing all strides from 1 to cache line size).

Building significant tests

Perfect! We know how to run tests on hardware, but

Building significant tests

Perfect! We know how to run tests on hardware, but what tests do we run?

We study *relaxed* memory models, that is *relaxed* w.r.t SC.

Hence, we focus on programs that have non-SC behaviours.

The question is: how do we generate such programs.

Let us study SC in detail first.

Back to our non-SC example

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

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;

We observe if $b < c$ then y=2, if $d < a$ then r0=0.

Let us be a bit more clever

R

T_0	T_1
$(a)\,\texttt{x} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 2$
$(b)\,\texttt{y} \leftarrow 1$	$(d)\,\texttt{r0} \leftarrow \texttt{x}$

Observed? `y=2; r0=0`

Collecting constraints on the scheduling order $<$:

We respect program order, thus

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 $a < b$ and $c < d$.
We observe `r0=0`, thus

Let us be a bit more clever

R	
T_0	T_1
$(a)\,\texttt{x} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 2$
$(b)\,\texttt{y} \leftarrow 1$	$(d)\,\texttt{r0} \leftarrow \texttt{x}$
Observed? `y=2; r0=0`	

Collecting constraints on the scheduling order $<$:

We respect program order, thus $a < b$ and $c < d$.
We observe `r0=0`, thus $d < a$.
We observe `y=2`, thus

Let us be a bit more clever

R

T_0	T_1
$(a)\,\texttt{x} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 2$
$(b)\,\texttt{y} \leftarrow 1$	$(d)\,\texttt{r0} \leftarrow \texttt{x}$

Observed? y=2; r0=0

Collecting constraints on the scheduling order $<$:

We respect program order, thus $a < b$ and $c < d$.
We observe r0=0, thus $d < a$.
We observe y=2, thus $b < c$.

Hence we have a cycle in $<$, which prevents it from being an order!

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

Conclusion: No SC execution would ever yield the output "y=2; r0=0;".

Systematic approach

For a particular (partial) execution candidate (that is, for a set of events and a $\xrightarrow{\text{po}}$ relation) we assume two additional relations:

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

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

- **Coherence ($\xrightarrow{\mathbf{co}}$):** 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 order.

Notice: To me, the very existence of $\xrightarrow{\text{co}}$ stems from the existence of a shared, coherent, memory — Given location ℓ, there is exactly one memory cell whose location is ℓ.

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

2+2W

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

Observe: `x; y;`

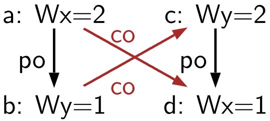
`x=1; y=2;`	`x=1; y=1;`
`x=2; y=2;`	`x=2; y=1;`

Example of $\xrightarrow{co}$

2+2W

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

Observe: `x; y;`

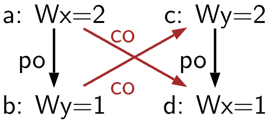
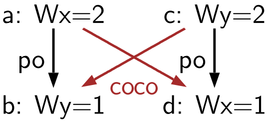
`x=1; y=2;` 	`x=1; y=1;`
`x=2; y=2;`	`x=2; y=1;`

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

2+2W

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

Observe: `x; y;`

`x=1; y=2;`	`x=1; y=1;`
	
`x=2; y=2;`	`x=2; y=1;`

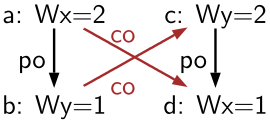
Example of $\xrightarrow{\text{co}}$

2+2W

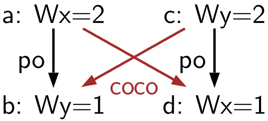
T_0	T_1
(a) x $\leftarrow$ 1 (b) y $\leftarrow$ 2	(c) y $\leftarrow$ 1 (d) x $\leftarrow$ 2

Observe: x; y;

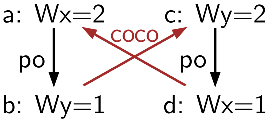
x=1; y=2;



x=1; y=1;



x=2; y=2;



x=2; y=1;

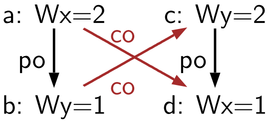
Example of $\xrightarrow{\text{co}}$

2+2W

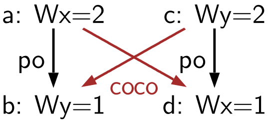
T_0	T_1
(a) `x` $\leftarrow 1$	(c) `y` $\leftarrow 1$
(b) `y` $\leftarrow 2$	(d) `x` $\leftarrow 2$

Observe: `x; y;`

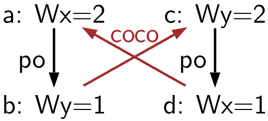
`x=1; y=2;`



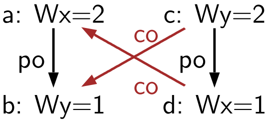
`x=1; y=1;`



`x=2; y=2;`



`x=2; y=1;`

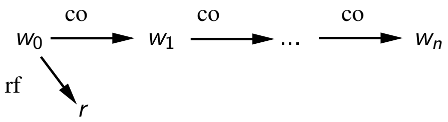


One more relation: $\xrightarrow{\text{fr}}$

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



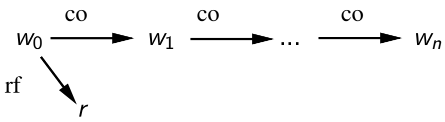
We have

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

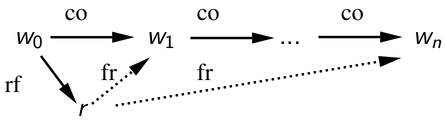
The new relation $\xrightarrow{\text{fr}}$ (from read) relates reads to "younger writes" (younger w.r.t. $\xrightarrow{\text{co}}$).

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

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



We have



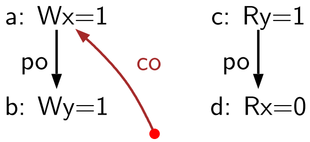
Playing with $\xrightarrow{\text{fr}}$

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

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`



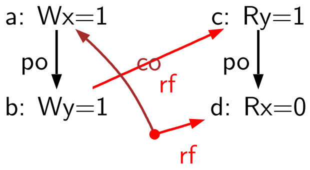
Playing with $\xrightarrow{\text{fr}}$

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

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`



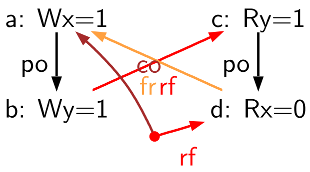
Playing with $\xrightarrow{\text{fr}}$

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

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`



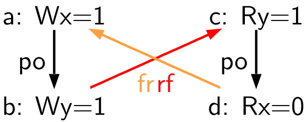
Playing with $\xrightarrow{\text{fr}}$

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

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`



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

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

MP

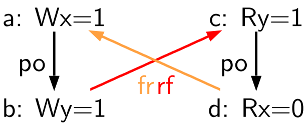
T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `r1` $\leftarrow$ `x`

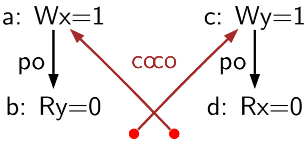
Observed? `r0=1; r1=0`

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`





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

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

MP

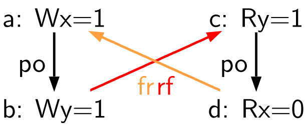
T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `r1` $\leftarrow$ `x`

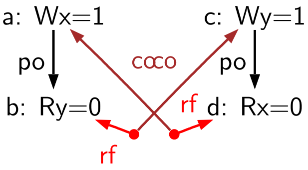
Observed? `r0=1`; `r1=0`

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`





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

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

MP

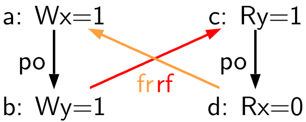
T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `r1` $\leftarrow$ `x`

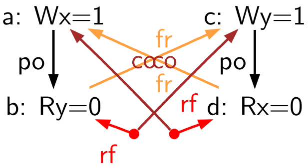
Observed? `r0=1; r1=0`

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`





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

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

MP

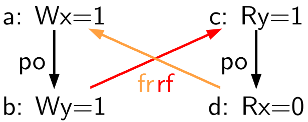
T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `r0` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `r1` $\leftarrow$ `x`

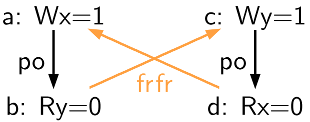
Observed? `r0=1; r1=0`

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`





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:

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.

Introducing herd a memory model simulator

A model `sc.cat`:

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

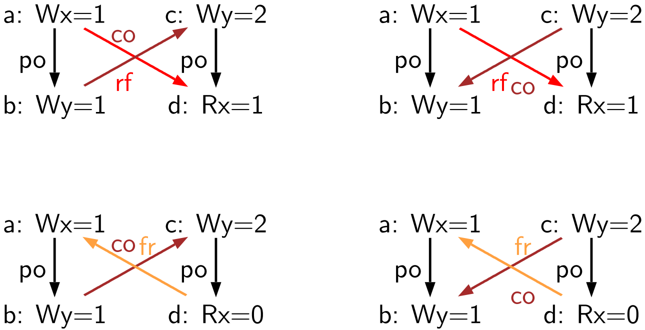
Running **R** on SC (demo in `demo/02`):

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

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

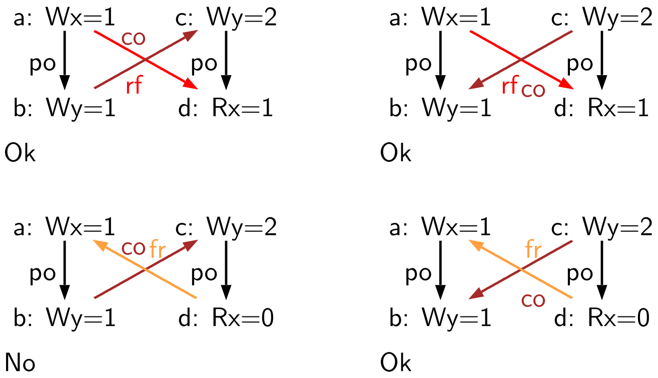
Herd structure

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



Herd structure

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



- Apply model checks to each candidate execution.

Violations of SC

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

However, the cycle does not describe:

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

We now aim at:

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

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

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

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

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

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

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

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

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

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

Given a cycle, we consider all $\xrightarrow{\text{com}}$ and $\xrightarrow{\widehat{\text{com}}}$ steps are external, that is source and target events are from pairwise distinct thread.

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

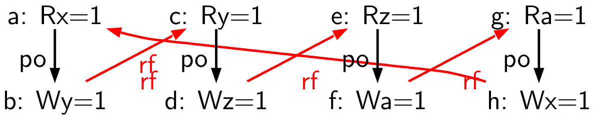
- Either $e_1 \xrightarrow{\text{po}} e_2$ and we consider this $\xrightarrow{\text{po}}$ step in the cycle, in place of the $\xrightarrow{\widehat{\text{com}}}$ step.
- Or $e_2 \xrightarrow{\text{po}} e_1$ and we have a very simple cycle $e_2 \xrightarrow{\text{po}} e_1 \xrightarrow{\widehat{\text{com}}} e_2$. Such cycles are "*violations of coherence*" (more on them later).

Notice: The same reasoning applies individual $\xrightarrow{\text{com}}$ steps in composite $\xrightarrow{\widehat{\text{com}}}$.

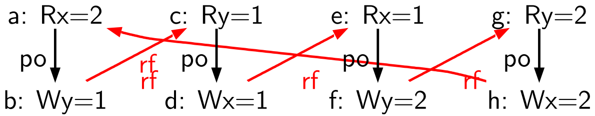
Simplifying cycles — Locations

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

- One interpretation (four locations):

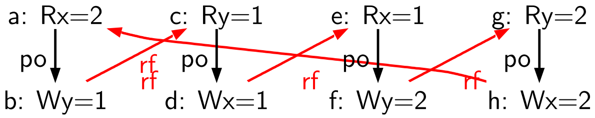


- Another interpretation (two locations):



The second interpretation is not "minimal"

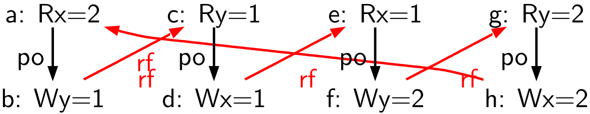
Reminding the interpretation with two locations:



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

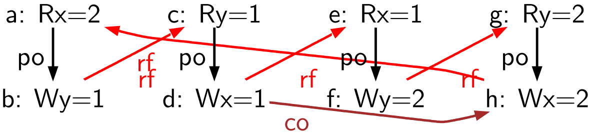
The second interpretation is not “minimal”

Reminding the interpretation with two locations:



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

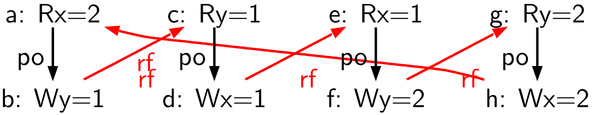
Let us choose: Wx1 $\xrightarrow{\text{co}}$ Wx2:



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

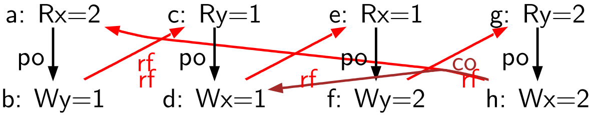
The second interpretation is not "minimal"

Reminding the interpretation with two locations:



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

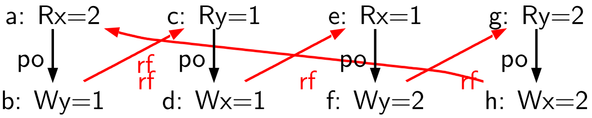
Let us choose: Wx2 $\xrightarrow{\text{co}}$ Wx1:



We have a smaller cycle: $h \xrightarrow{\text{co}} d \xrightarrow{\text{rf}} e \xrightarrow{\text{po}} f \xrightarrow{\text{rf}} g \xrightarrow{\text{po}} h$.

The second interpretation is not “minimal”

Reminding the interpretation with two locations:

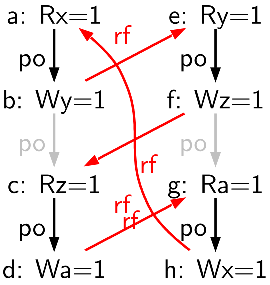


But, coherence $\xrightarrow{\text{co}}$ totally orders write events to a given location.

Generally: do not repeat locations in cycles.

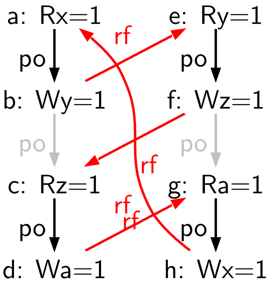
Simplifying cycles — Threads

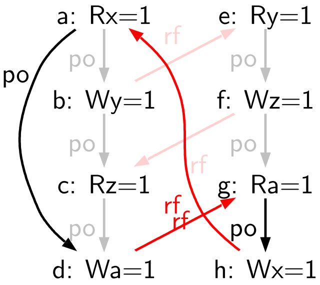
Cycle: $W \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} R \xrightarrow{\text{fr}} W \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} R \xrightarrow{\text{fr}}$



Simplifying cycles — Threads

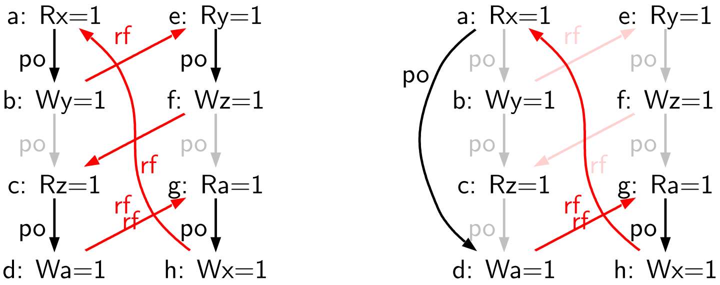
Cycle: $W \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} R \xrightarrow{\text{fr}} W \xrightarrow{\text{po}} W \xrightarrow{\text{rf}} R \xrightarrow{\text{po}} R \xrightarrow{\text{fr}}$





Simplifying cycles — Threads

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



Generally: one passage per thread

. . . Simplifying cycles

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, with at least four 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.

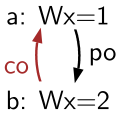
Notice: For a more formal presentation see D. Shasha and M. Snir Toplas 88 article, which introduced critical cycles.

Violations of coherence

There are five such cycles, which can occur as the following executions:
$\xrightarrow{\text{po}}$ contradicts

Violations of coherence

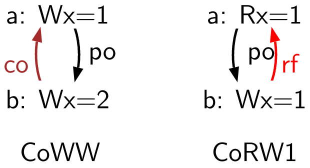
There are five such cycles, which can occur as the following executions:
$\xrightarrow{\text{po}}$ contradicts $\xrightarrow{\text{co}}$,



CoWW

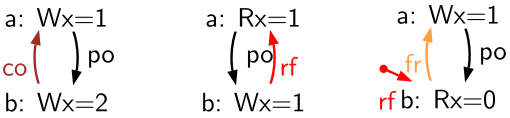
Violations of coherence

There are five such cycles, which can occur as the following executions:
$\xrightarrow{\text{po}}$ contradicts $\xrightarrow{\text{co}}$, $\xrightarrow{\text{rf}}$,



Violations of coherence

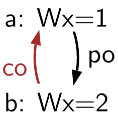
There are five such cycles, which can occur as the following executions: $\xrightarrow{\text{po}}$ contradicts $\xrightarrow{\text{co}}$, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{fr}}$,



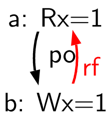
CoWW CoRW1 CoWR

Violations of coherence

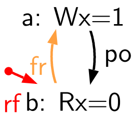
There are five such cycles, which can occur as the following executions:
$\xrightarrow{\text{po}}$ contradicts $\xrightarrow{\text{co}}$, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{fr}}$, "$\xrightarrow{\text{co}}$; $\xrightarrow{\text{rf}}$",



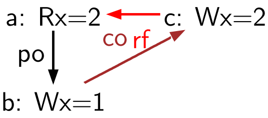
CoWW



CoRW1



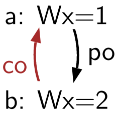
CoWR



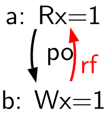
CoRW2

Violations of coherence

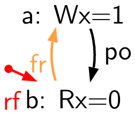
There are five such cycles, which can occur as the following executions:
$\xrightarrow{\text{po}}$ contradicts $\xrightarrow{\text{co}}$, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{fr}}$, "$\xrightarrow{\text{co}};\xrightarrow{\text{rf}}$", "$\xrightarrow{\text{fr}};\xrightarrow{\text{rf}}$".



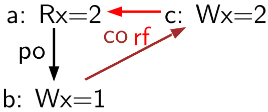
CoWW



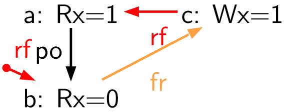
CoRW1



CoWR



CoRW2



CoRR

Building a test from a cycle

The conditions on locations and threads allows generating one test (program + final condition) from one cycle.
Consider for instance:

$$\xrightarrow{\text{fre}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}$$

- Set events (with directions, from communication relations)

Building a test from a cycle

The conditions on locations and threads allows generating one test (program + final condition) from one cycle.
Consider for instance:

$$\xrightarrow{\text{fre}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}$$

- Set events (with directions, from communication relations)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{W}[?]?$$

Building a test from a cycle

The conditions on locations and threads allows generating one test (program + final condition) from one cycle.
Consider for instance:

$$\xrightarrow{\text{fre}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}$$

- Set events (with directions, from communication relations)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{W}[?]?$$

- Set locations (one per $\widehat{\xrightarrow{\text{com}}}$)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[x]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[x]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{R}[x]?$$

Building a test from a cycle

The conditions on locations and threads allows generating one test (program + final condition) from one cycle.
Consider for instance:

$$\xrightarrow{\text{fre}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}$$

- Set events (with directions, from communication relations)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{W}[?]?$$

- Set locations (one per $\widehat{\xrightarrow{\text{com}}}$)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[x]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[x]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[y]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[y]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{R}[x]?$$

- Set values for writes (initial value 0, then follow cycle: 1, 2, etc.)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[x]1 \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[x]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[y]1 \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[y]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{R}[x]?$$

Building a test from a cycle

The conditions on locations and threads allows generating one test (program + final condition) from one cycle.
Consider for instance:

$$\xrightarrow{\text{fre}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}\xrightarrow{\text{rfe}}\xrightarrow{\text{po}}$$

- Set events (with directions, from communication relations)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[?]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[?]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{W}[?]?$$

- Set locations (one per $\xrightarrow{\widehat{\text{com}}}$)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[x]? \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[x]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[y]? \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[y]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{R}[x]?$$

- Set values for writes (initial value 0, then follow cycle: 1, 2, etc.)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[x]1 \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[x]? \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[y]1 \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[y]? \xrightarrow{\text{po}} (e)\text{:}\mathbf{R}[x]?$$

- Set values for reads (consider $\xrightarrow{\text{rfe}} R$ or $R \xrightarrow{\text{fre}}$)

$$\xrightarrow{\text{fre}} (a)\text{:}\mathbf{W}[x]1 \xrightarrow{\text{rfe}} (b)\text{:}\mathbf{R}[x]1 \xrightarrow{\text{po}} (c)\text{:}\mathbf{W}[y]1 \xrightarrow{\text{rfe}} (d)\text{:}\mathbf{R}[y]1 \xrightarrow{\text{po}} (e)\text{:}\mathbf{R}[x]0$$

Building a test from a cycle, continued

$$\xrightarrow{\text{fre}} (a){:}\mathbf{W}[x]1 \xrightarrow{\text{rfe}} (b){:}\mathbf{R}[x]1 \xrightarrow{\text{po}} (c){:}\mathbf{W}[y]1 \xrightarrow{\text{rfe}} (d){:}\mathbf{R}[y]1 \xrightarrow{\text{po}} (e){:}\mathbf{R}[x]0$$

- Build program from events (change thread at every $\xrightarrow{\text{com}}$).

WRC

T_0	T_1	T_2
(a) `x` $\leftarrow$ `1`	(b) `r0` $\leftarrow$ `x` (c) `y` $\leftarrow$ `1`	(d) `r1` $\leftarrow$ `y` (e) `r2` $\leftarrow$ `x`

- Build condition from coherence sequences (here nothing) and from values read.

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

Building another test, non-trivial coherence

Test **R**

$$\xrightarrow{\text{fre}} (a){:}\mathbf{W}[x]1 \xrightarrow{\text{po}} (b){:}\mathbf{W}[y]1 \xrightarrow{\text{coe}} (c){:}\mathbf{W}[y]2 \xrightarrow{\text{po}} (x){:}\mathbf{R}[x]0$$

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`

Here, coherence order is $0 \xrightarrow{\text{co}} 1 \xrightarrow{\text{co}} 2$, it suffices to read the final value.

Longer coherence orders command other techniques, for instance adding an observer thread.

A first tool: diyone

Generating **WRC**:

```
% diyone -arch X86 Rfe Pod** Rfe Pod** Fre
X86 A
 P0          | P1           | P2           ;
 MOV [x],$1  | MOV EAX,[x]  | MOV EAX,[y]  ;
             | MOV [y],$1   | MOV EBX,[x]  ;
exists (1:EAX=1 /\ 2:EAX=1 /\ 2:EBX=0)
```

Doing the same for ARM is as simple as:

```
% diyone -arch ARM Rfe Pod** Rfe Pod** Fre
ARM A
{ %x0=x; %x1=x; %y1=y; %y2=y; %x2=x; }
 P0            | P1            | P2            ;
 MOV R0,#1     | LDR R0,[%x1]  | LDR R0,[%y2]  ;
 STR R0,[%x0]  | MOV R1,#1     | LDR R1,[%x2]  ;
               | STR R1,[%y1]  |               ;
exists (1:R0=1 /\ 2:R0=1 /\ 2:R1=0)
```

Tool diyone, generating **R**

```
% diyone -arch X86 PodWW Wse PodWR Fre
X86 A
"PodWW Wse PodWR Fre"
{ }
 P0         | P1           ;
 MOV [x],$1 | MOV [y],$2   ;
 MOV [y],$1 | MOV EAX,[x]  ;
exists (y=2 /\ 1:EAX=0)
```

Notice: We wrote `PodWW`, `PodWR`. The vocabulary of *Candidate Relaxations* is quite rich:

- Internal communications `Rfi`, `Fri`, `Wsi`.
- $\xrightarrow{\text{po}}$ edges with identical target and source locations: `PosRR`, etc.
- Dependencies (`DpAddrdW`, etc.). fences (`MFencedWR`, etc.)

Application, testing non-SC executions for two threads

Cycle $\rightarrow$ execution $\rightarrow$ program $+$ final condition.

All (critical) cycles for two threads: 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	$\xrightarrow{\text{po}}\xrightarrow{\text{fr}}\xrightarrow{\text{po}}\xrightarrow{\text{fr}}$

Save of coherence violation, any non-SC execution on two threads includes one of the above six cycles.

Hence, testing the six tests built from the six cycles gives reasonable coverage of possible SC violation on two threads. (**Notice:** coherence violations neglected).

Generating two-threads SC violations

The tool diy generates cycles (and tests) from a vocabulary of CR. 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/03`.

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

Demo 03 continued, running the tests

Compiling:

```
% litmus -mach ./x86.cfg src/@all -o run
% make -C run -j 4
```

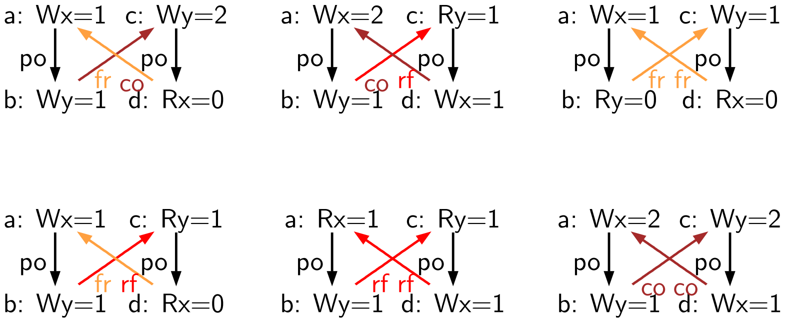
Running:

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

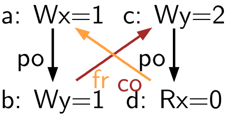
Analysis:

```
% 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
```

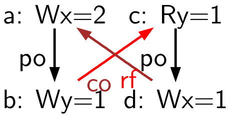
Results for running the six tests on this machine



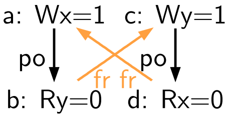
Results for running the six tests on this machine



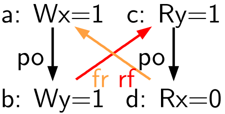
R: Ok



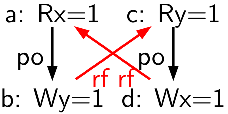
S: No



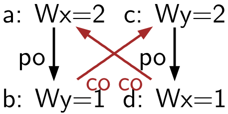
SB: Ok



MP: No

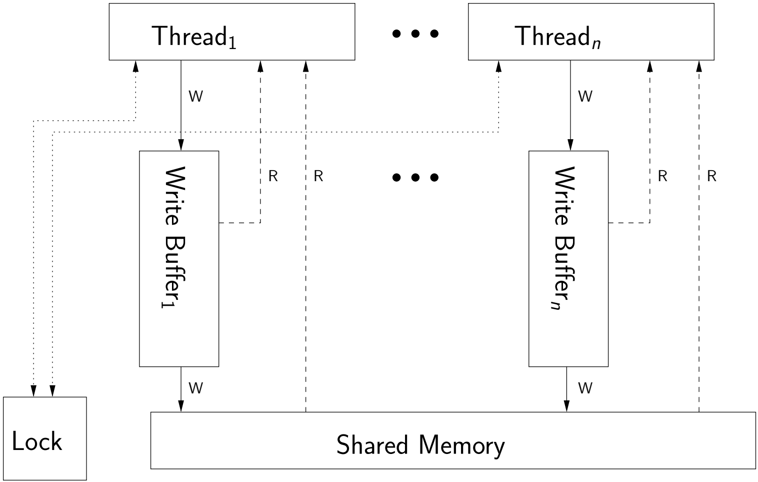


LB: No



2+2W: No

TSO — The Model of X86 machines



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

Axiomatic TSO

- 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 = RM(po) | WW(po)   # WR(po) omitted
```

  - Local reads do not create order:

```
let com-hb = rfe | fr | co # rfi omitted
```

- TSO "*happens-before*" (HB) check:

```
acyclic (ppo | com-hb | mfence) as hb
```

Axiomatic TSO

- 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 = RM(po) | WW(po)   # WR(po) omitted
    ```

  - Local reads do not create order:

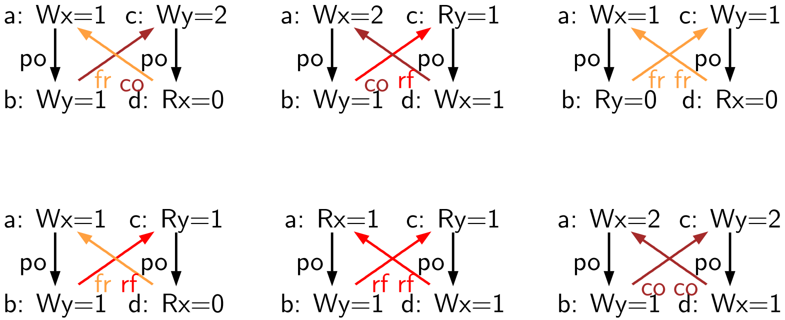
    ```
    let com-hb = rfe | fr | co # rfi omitted
    ```

- TSO "*happens-before*" (HB) check:

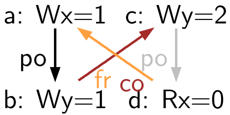
```
acyclic (ppo | com-hb | mfence) as hb
```

Notice: Relations are between the points in time where a load binds its value and where a written value reaches memory.

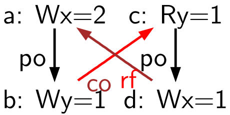
Results for running the six test on the model

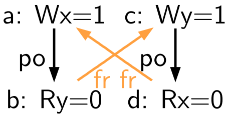


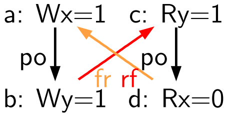
Results for running the six test on the model

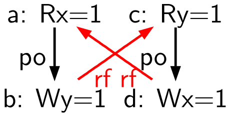


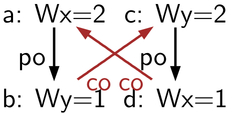
R: Ok



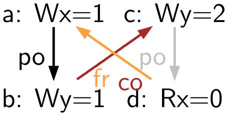




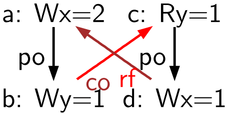




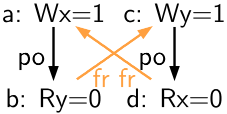
Results for running the six test on the model

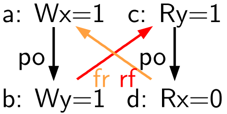


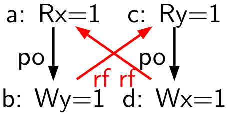
R: Ok

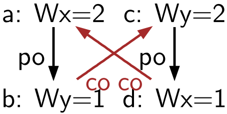


S: No

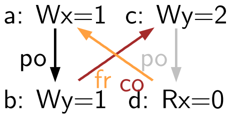




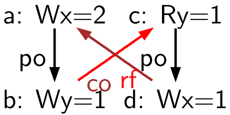




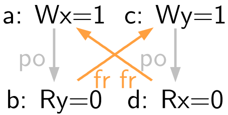
Results for running the six test on the model



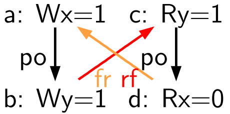
R: Ok

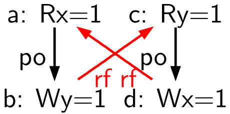


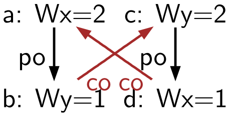
S: No



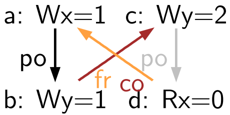
SB: Ok



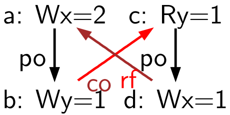




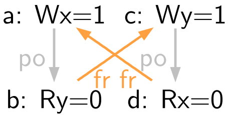
Results for running the six test on the model



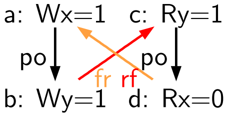
R: Ok



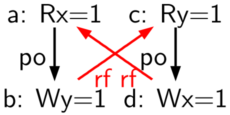
S: No



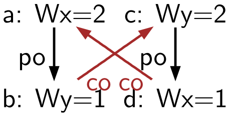
SB: Ok



MP: No



LB: No



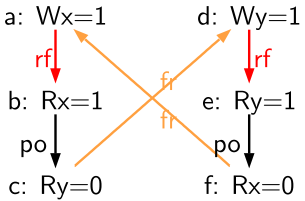
2+2W: No

Internal $\xrightarrow{\text{rf}}$ ($\xrightarrow{\text{rfi}}$) are not in HB

SB+rfi-pos

T_0	T_1
(a) x ← 1	(d) y ← 1
(b) r0 ← x	(e) r2 ← y
(c) r1 ← y	(f) r3 ← x

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

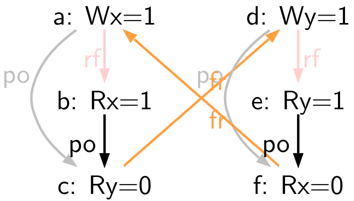


Internal $\xrightarrow{\text{rf}}$ ($\xrightarrow{\text{rfi}}$) are not in HB

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;`

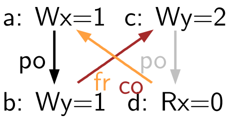


SB+rfi-pos: Ok

Restoring SC

R	
T_0	T_1
$(a)\,\texttt{x} \leftarrow 1$ $(b)\,\texttt{y} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 2$ $(d)\,\texttt{r0} \leftarrow \texttt{x}$

Observed? `y=2; r0=0`

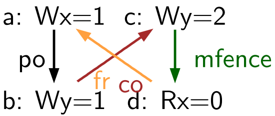


Restoring SC with `mfence`

R+po+mfence

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

Observed? `y=2; r0=0`

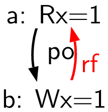


We are not done yet...

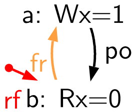
Our TSO model:

```
let ppo = RM(po) | WW(po)   # WR(po) omitted
let com-hb = rfe | fr | co # rfi omitted
acyclic (ppo | com-hb)
show ppo | com-hb as hb
```

Allows two violations of coherence:



CoRW1



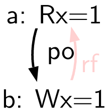
CoWR

We are not done yet...

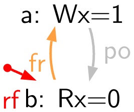
Our TSO model:

```
let ppo = RM(po) | WW(po)   # WR(po) omitted
let com-hb = rfe | fr | co # rfi omitted
acyclic (ppo | com-hb)
show ppo | com-hb as hb
```

Allows two violations of coherence:



CoRW1



CoWR

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

Those behaviours *must* be rejected by our TSO model.

Correct axiomatic TSO

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.

```
irreflexive (po-loc; com+) 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.

A word on UNIPROC

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

Definition (Uniproc 1)

Program order $\xrightarrow{\text{po}}$ does not contradict communication $\xrightarrow{\text{com}}^{+}$.

There is another definition "SC per location"'. (Jason F. Cantin, Mikko H. Lipasti, James E. Smith ACM Symposium on Parallel Algorithms and Architectures 2004).

Definition (Uniproc 2)

Relation $\xrightarrow{\text{po-loc}} \cup \xrightarrow{\text{com}}$ is acyclic.

Definitions are equivalent.
It suffices to show that the existence of a cycle in $\xrightarrow{\text{po-loc}} \cup \xrightarrow{\text{com}}$ implies the existence of a coherence violation (*i.e.* a cycle $e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\widehat{\text{com}}} e_1$).

Consequence of $\xrightarrow{\text{co}}$ ordering writes

Lemma (Identical locations)

Let e_1, e_2 be 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 $w_2 \xrightarrow{\text{co}}\xrightarrow{\text{rf}} r_1$.

Consequence of $\xrightarrow{\text{co}}$ ordering writes

Lemma (Identical locations)

Let e_1, e_2 be 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.

Equivalence of the two UNIPROC definitions

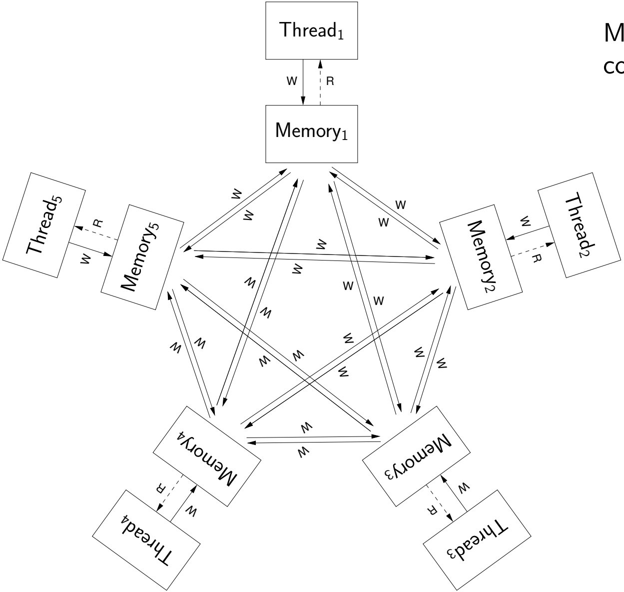
Proof is easy from "Identical locations" lemma.

Consider a cycle in $\xrightarrow{\text{pos}} \cup \xrightarrow{\text{com}}$.

- If there exists a $e_1 \xrightarrow{\text{po}} e_2$ step s.t. $e_2 \xrightarrow{\widehat{\text{com}}} e_1$, then we are done.
- Otherwise, for each $e_1 \xrightarrow{\text{po}} e_2$ 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, $e_1 \xrightarrow{\widehat{\text{com}}} e_2$. 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.

A relaxed shared memory computer



More or less visible to user code:

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

Situation of (our) ARM/Power models

- **Architecture public reference** Informal, cannot clearly explain how fences restore SC for instance.
- **Simple, global-time model:** (CAV'10) too relaxed. It remains useful as it supports simple reasoning on SC-violations (CAV'11).
- **Operational model:** (PLDI'11) more precise, 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:** (...) more precise (proved to be more relaxed than PLDI'11, experimentally equivalent). A more simple axiomatic model.

Joint work with (in order of appearance) Jade Alglave, Susmit Sarkar, Peter Sewell, Derek Williams, Kayvan Memarian, Scott Owens, Mark Batty, Sela Mador-Haim, Rajeev Alur, Milo M. K. Martin and Michael Tautschnig.

Some issues for ARM/Power

- No simple preserved-program-order. More precisely, $\xrightarrow{\text{ppo}}$ 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.

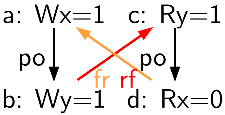
An experiment on ARM/Power

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 ← x
r2 ← 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 ← x
y ← 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:

r1 ← x
if r1=0 then
  y ← 1

```
  lwz r1,0(r8)
  cmpwi r1,0
  bne L1

  li r2,1
  stw r2,0(r9)
L1:
```

Dependencies (Power)

Address dependency:

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

Data dependency:

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

Control dependency: (+isync)

```
r1 ← x                 lwz r1,0(r8)
if r1=0 then           cmpwi r1,0
   (isync)             bne L1
   y ← 1               (isync)
                       li r2,1
                       stw r2,0(r9)
                     L1:
```

Generating tests (ARM), yet another tool: diycross

Generating tests with diycross (demo in `demo/04`):

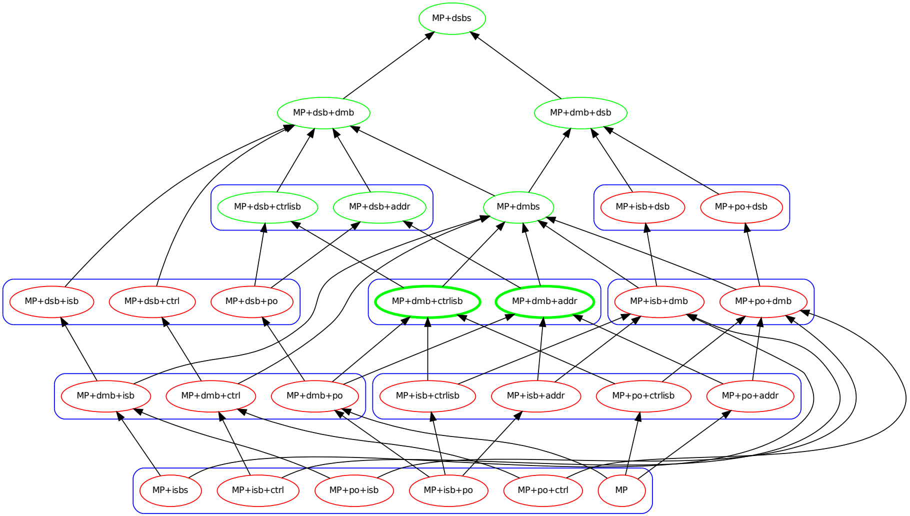
```
% diycross -arch ARM\
 PodWW,DMBdWW,DSBdWW,ISBdWW\
 Rfe\
 PodRR,DpCtrldR,DpCtrlIsbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR\
 Fre
Generator produced 28 tests
```

- One generates **MP** as `diyone PodWW Rfe PodRR Fre`
- `diycross` $r^1_1, \ldots, r^1_{N_1} \; \cdots \; r^M, \ldots, r^M_{N_M}$, 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 CR list arguments.

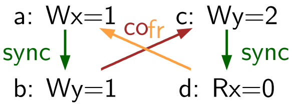
This generates some variations in the **MP** family.

We then compile and run, and…

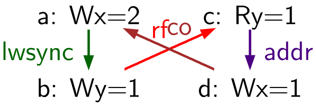
Optimal fencing/dependencies for MP



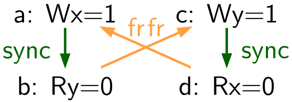
Optimal fencing for the 6 two-threads tests (Power)



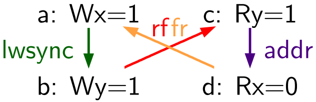
R+syncs



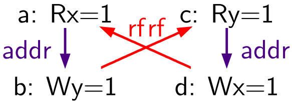
S+lwsync+addr



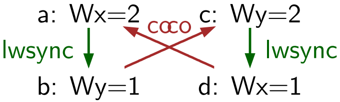
SB+syncs



MP+lwsync+addr



LB+addrs



2+2W+lwsyncs

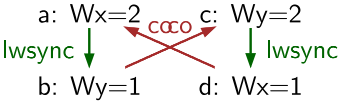
Some observations

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

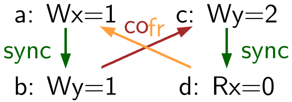
$$\mathit{addr} < \texttt{lwsync} < \texttt{sync}$$

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+syncs**.



2+2W+lwsyncs



R+syncs

No

No

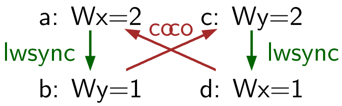
Some observations

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

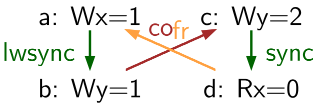
$$\mathit{addr} < \texttt{lwsync} < \texttt{sync}$$

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

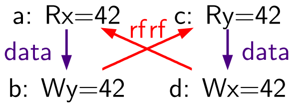
Ok

Dependencies are enough

LB+datas

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

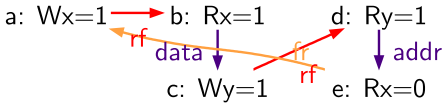
Dependencies from reads not always enough!

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

WRC

T_0	T_1	T_2
(a) `x` $\leftarrow$ 1	(b) `r0` $\leftarrow$ `x` (c) `y` $\leftarrow$ 1	(d) `r1` $\leftarrow$ `y` (e) `r2` $\leftarrow$ `x`

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



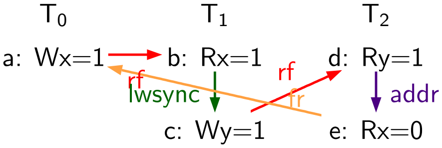
WRC+data+addr

Behaviour observed on Power 6 and 7 (not on ARM, but documentation allows it).

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

Restoring SC for **WRC**

Use a lightweight fence on 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).

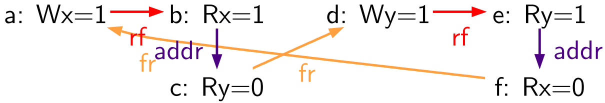
Another case of unsufficient dependencies

Consider test **IRIW+addrs**:

IRIW

T_0	T_1	T_2	T_3
(a) x ← 1	(b) r0 ← x (c) r1 ← y	(d) y ← 1	(e) r2 ← y (f) r3 ← x

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

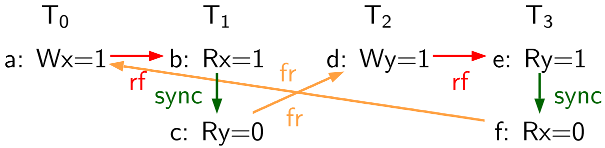


IRIW+addrs

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.
- 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

UNIPROC

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

HB

```
let fence = strong | light
let hb = ppo | fence | rfe
acyclic hb
```

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

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

PROPAGATION Strong fences wait for all communications.

```
let propstrong = com*; propbase*; strong; hb*
let prop = WW(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 = RW(ic) | RR(ii)
```

Can be limited to dependencies...

How good is our model?

Is it sound?

- A proof: any behaviour allowed is also allowed by the operational model of PLDI'11.
- Experiments
  - Soundness w.r.t. hardware (ARM being a bit problematic because of acknowledged read-after-read hazard).
  - Experimental equivalence with our previous models, saved from current debate on some subtle semantical point for `lwsync`.

In any case:

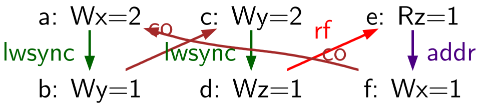
- Simulation is fast ($\times$1000 w.r.t. PLDI'11) ($\times$10 w.r.t. CAV'12).
- The existence of four checks UNIPROC, HB OBSERVATION and PROPAGATION stand on firm bases.
- The semantics of strong fences also does.
- The model and simulator (*i.e.* herd) are flexible, one easily change a few relations (*e.g.* $\xrightarrow{\text{ppo}}$, or the semantics of weak fences).

Subtle point

Z6.1

T_0	T_1	T_2
(a) `x` $\leftarrow$ 2	(c) `y` $\leftarrow$ 2	(d) `r0` $\leftarrow$ `z`
(d) `y` $\leftarrow$ 1	(e) `z` $\leftarrow$ 1	(f) `x` $\leftarrow$ 1

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

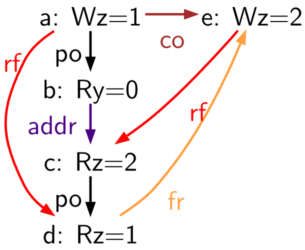


Z61+lwsync+lwsync+addr

Unobserved and forbidden by model. May be allowed. . .

A test of coherence violation

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



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