Weak Memory Models: A Tutorial

Jade Alglave

University College London

February 3rd, 2014

Sequential Consistency

A comfortable model for concurrent programming would be Sequential Consistency (SC), as defined by Leslie Lamport in 1979:

> *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.*

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`
`r1=?; r2=?;`	

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$
$(a)(c)(d)(b)$	
$(c)(a)(b)(d)$	
$(c)(a)(d)(b)$	

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

`r1=?; r2=?;`

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$
$(a)(c)(d)(b)$	
$(c)(a)(b)(d)$	
$(c)(a)(d)(b)$	

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

`r1=0; r2=?;`

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

`r1=0; r2=?;`

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$
$(a)(c)(d)(b)$	
$(c)(a)(b)(d)$	
$(c)(a)(d)(b)$	

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

`r1=0; r2=1;`

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$

Experiment

On an Intel Core 2 Duo:

```
{x=0; y=0;}

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

exists (0:EAX=0 /\ 1:EAX=0)
```

Certain instructions appear to be reordered w.r.t. the program order.

Let us check that on my machine.

Weak memory models

For performance reasons, modern architectures provide several features that are weakenings of SC:

> *For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs.*

How can we make sure that we write correct programs?

- We need to understand precisely what memory models guarantee to write correct concurrent programs.
- This problem spreads to high level languages and is potentially much worse, due to compiler optimisations.

Surely there are specs?

Documentation is (at least) ambiguous, since written in natural language.

Surely there are specs?

"all that horrible horribly incomprehensible and confusing [. . .] text that no-one can parse or reason with — not even the people who wrote it"

Anonymous Processor Architect, 2011

Describing executions

Style of modelling

Memory models roughly fall into two classes:

- Operational
- Axiomatic

Building an execution

rlns

P_0	P_1
$(a)\ \mathtt{x} \leftarrow 2$	$(b)\ \mathtt{x} \leftarrow 1$
	$(c)\ \mathtt{r1} \leftarrow \mathtt{x}$

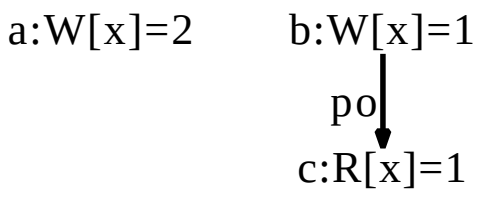
Allowed: `1:r1=1; x=2;`

Building an execution : Events $\mathbb{E}$ and program order po

rlns

P_0	P_1
$(a)\,\mathtt{x} \leftarrow 2$	$(b)\,\mathtt{x} \leftarrow 1$
	$(c)\,\mathtt{r1} \leftarrow \mathtt{x}$

Allowed: `1:r1=1; x=2;`



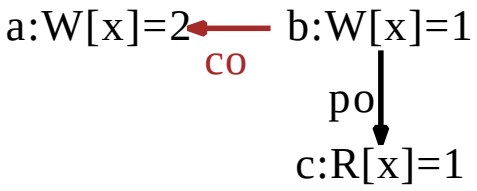
We write $E \triangleq (\mathbb{E}, \mathrm{po})$ for such a structure.

Building an execution : Coherence co

rlns

P_0	P_1
(a) x $\leftarrow$ 2	(b) x $\leftarrow$ 1
	(c) r1 $\leftarrow$ x

Allowed: 1:r1=1; x=2;



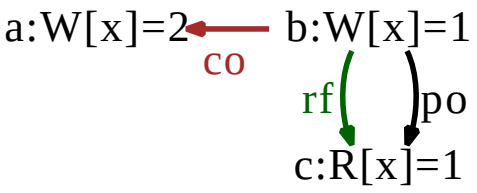
The coherence co orders totally all the write events to the same memory location.

Building an execution : Read-from rf

rlns

P_0	P_1
(a) x $\leftarrow$ 2	(b) x $\leftarrow$ 1 (c) r1 $\leftarrow$ x

Allowed: 1:r1=1; x=2;



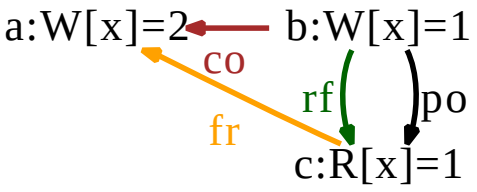
The read-from map rf links a write and any read that reads from it.

Building an execution : From-read map fr

rlns

P_0	P_1
(a) x $\leftarrow$ 2	(b) x $\leftarrow$ 1 (c) r1 $\leftarrow$ x

Allowed: 1:r1=1; x=2;



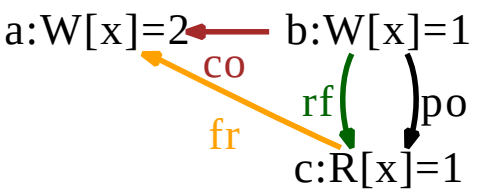
We derive the from-read map fr from co and rf.

Building an execution : Execution witness $X \triangleq (\mathrm{co}, \mathrm{rf})$

rlns

P_0	P_1
$(a)\ \mathtt{x} \leftarrow 2$	$(b)\ \mathtt{x} \leftarrow 1$
	$(c)\ \mathtt{r1} \leftarrow \mathtt{x}$

Allowed: `1:r1=1; x=2;`



We define an execution witness as $X \triangleq (\mathrm{co}, \mathrm{rf})$.

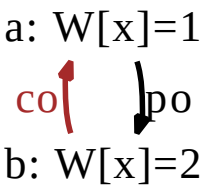
Describing architectures

Four axioms

- Uniproc
- No thin air
- Causality
- Propagation

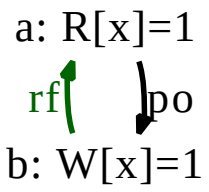
Uniproc (Coherence)

All the models I have studied preserve SC per location.



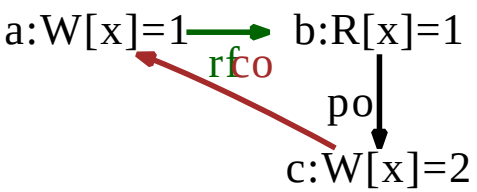
Uniproc (Coherence)

All the models I have studied preserve SC per location.



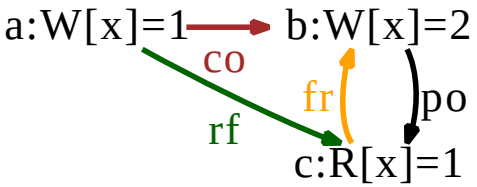
Uniproc (Coherence)

All the models I have studied preserve SC per location.



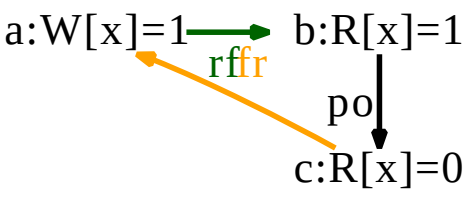
Uniproc (Coherence)

All the models I have studied preserve SC per location.



Uniproc (Coherence)

All the models I have studied preserve SC per location.



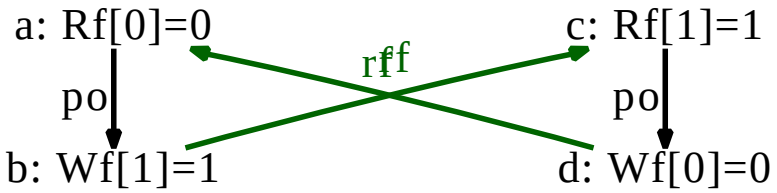
Uniproc (Coherence)

All the models I have studied preserve SC per location.

This ensures that non-relational analyses are sound on weak memory.

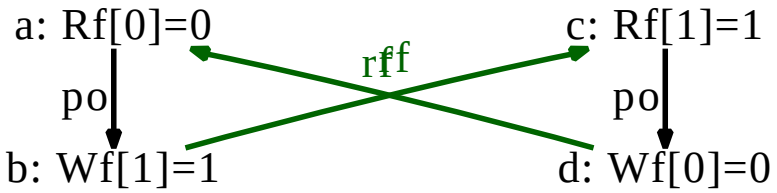
No thin air

All the models I have studied define a *happens-before* relation:



No thin air

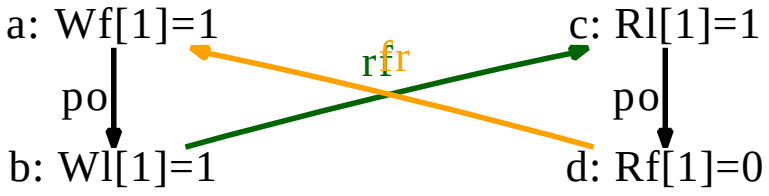
All the models I have studied define a *happens-before* relation:



which should be acyclic

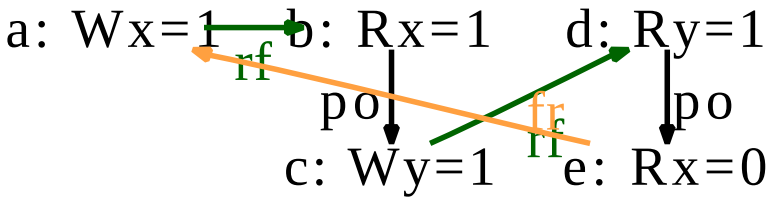
Causality (mp)

This happens-before relation determines which message passing idioms work as intended:



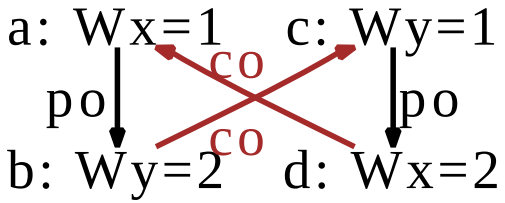
Causality (wrc)

This happens-before relation determines which write-to-read causality idioms work as intended:



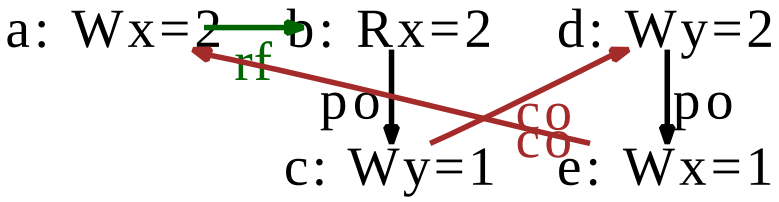
Propagation (2+2w)

Fences constrain the order in which writes to different locations propagate:



Propagation (w+rw+2w)

Fences constrain the order in which writes to different locations propagate:



A real-world excerpt

PostgreSQL developers' discussions

Yes, WaitLatch is vulnerable to weak-memory-ordering bugs - Iceweasel

File Edit View History Bookmarks Tools Help

http://www.postgresql.org/message-id/24241.1312739269@sss.pgh.pa.us

Yes, WaitLatch is vulnerable ... | Re: Latches with weak mem...

Yes, WaitLatch is vulnerable to weak-memory-ordering bugs

From: Tom Lane <tgl(at)sss(dot)pgh(dot)pa(dot)us>
To: pgsql-hackers(at)postgreSQL(dot)org
Subject: Yes, WaitLatch is vulnerable to weak-memory-ordering bugs
Date: 2011-08-07 17:47:49
Message-ID: 24241.1312739269@sss.pgh.pa.us (view raw or flat)
Thread: 2011-08-07 17:47:49 from Tom Lane <tgl(at)sss(dot)pgh(dot)pa(dot)us>
Lists: pgsql-hackers

```
I suspected $SUBJECT from the beginning, and I've now put in enough work
to be able to prove it.  The attached test program reliably fails within
a few minutes of being started, when run with 8 worker processes on an
8-core PPC machine.  It's a pretty simple "token passing ring" protocol,
and at some point one of the processes sees its latch set without seeing
its flag set, so it goes back to sleep and the token stops getting passed.
```

Done

Synchronisation in PostgreSQL

```
1  void worker(int i)
2   { while(!latch[i]);
3     for (;;)
4     { assert (!latch[i] || flag[i]);
5       latch[i] = 0;
6       if (flag[i])
7       { flag[i] = 0;
8         flag[(i+1)%WORKERS] = 1;
9         latch[(i+1)%WORKERS] = 1;
10      }
11      while(!latch[i]);
12    }
13  }
```

Each element of the array `latch` is a shared boolean variable dedicated to interprocess communication.

A process waits to have its latch set then should have work to do, namely passing around a token *via* the array `flag` (line 8).

Once the process is done, it sets the latch of the process the token was passed to (line 9).

Synchronisation in PostgreSQL

```
1  void worker(int i)
2   { while(!latch[i]);
3     for (;;)
4     { assert (!latch[i] || flag[i]);
5       latch[i] = 0;
6       if (flag[i])
7       { flag[i] = 0;
8         flag[(i+1)%WORKERS] = 1;
9         latch[(i+1)%WORKERS] = 1;
10      }
11      while(!latch[i]);
12    }
13  }
```

Starvation seemingly cannot occur: when a process is woken up, it has work to do.

Yet, the developers observed that the wait in line 11 would time out, *i.e.* starvation of the ring of processes.

The processor can delay the write in line 8 until after the latch had been set in line 9.

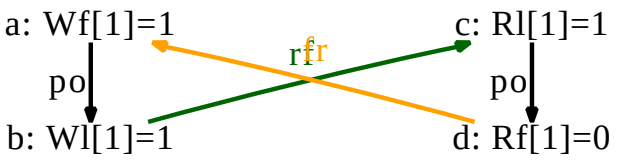
Message passing idiom in PostgreSQL

This corresponds to the message passing idiom

pgsql (mp)

Worker 0	Worker 1
(8) `f[1]=1;`	(2) `while(!l[1]);`
(9) `l[1]=1;`	(6) `if(f[1])`

Observed: `l[1]=1; f[1]=0`



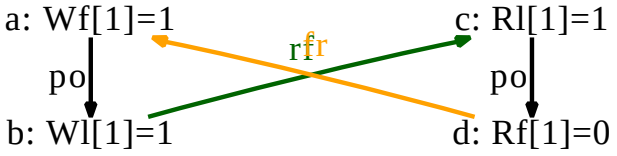
Message passing idiom in PostgreSQL

This corresponds to the message passing idiom which requires synchronisation to behave as on SC

pgsql (mp)

Worker 0	Worker 1
(8) `f[1]=1;`	(2) `while(!l[1]);`
`lwsync`	`dependency`
(9) `l[1]=1;`	(6) `if(f[1])`

Forbidden: `l[1]=1; f[1]=0`



Verification

Porte ouverte à deux battants

We propose two ways of verifying concurrent software running on weak memory:

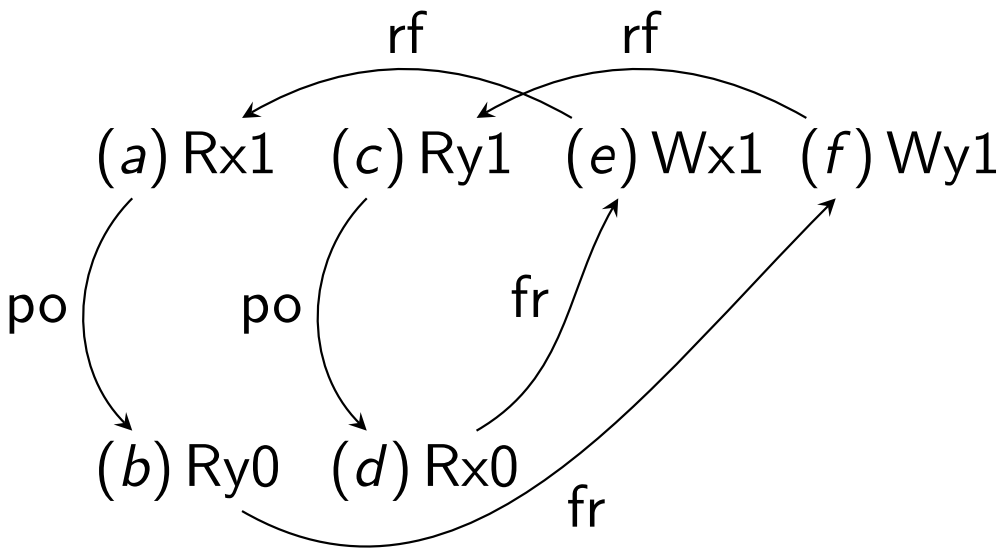
- we instrument the program to embed the weak memory semantics inside it, then feed the transformed program to an SC verification tool;
- we explicitly build partial order models representing the possible executions of the program on weak memory.

Independent Reads of Independent Writes

iriw

P_0	P_1	P_2	P_3
(a) r1 $\leftarrow$ x (b) r2 $\leftarrow$ y	(c) r3 $\leftarrow$ y (d) r4 $\leftarrow$ x	(e) x $\leftarrow$ 1	(f) y $\leftarrow$ 2

r1=1; r2=0; r3=2; r4=0;

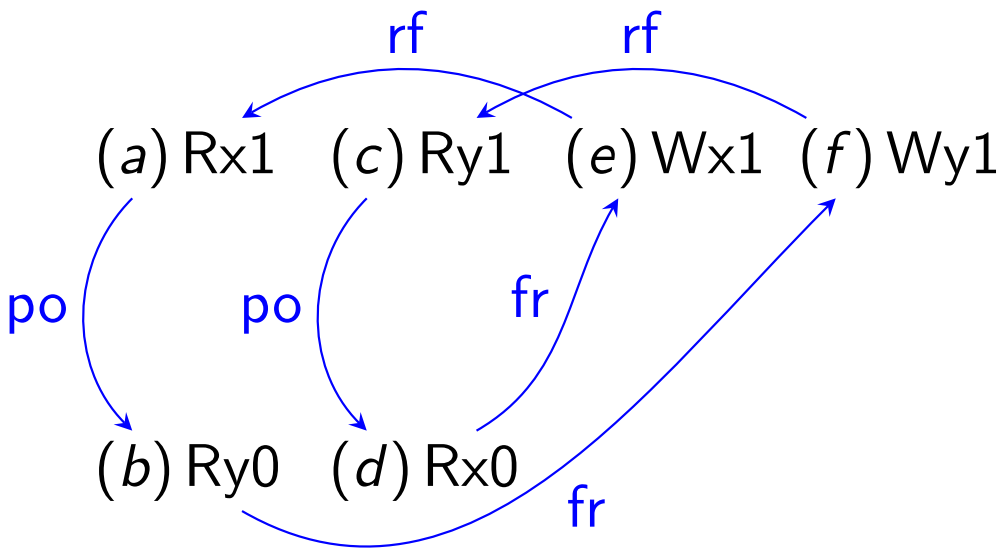


iriw on SC

iriw

P_0	P_1	P_2	P_3
(a) r1 $\leftarrow$ x (b) r2 $\leftarrow$ y	(c) r3 $\leftarrow$ y (d) r4 $\leftarrow$ x	(e) x $\leftarrow$ 1	(f) y $\leftarrow$ 2

r1=1; r2=0; r3=2; r4=0;

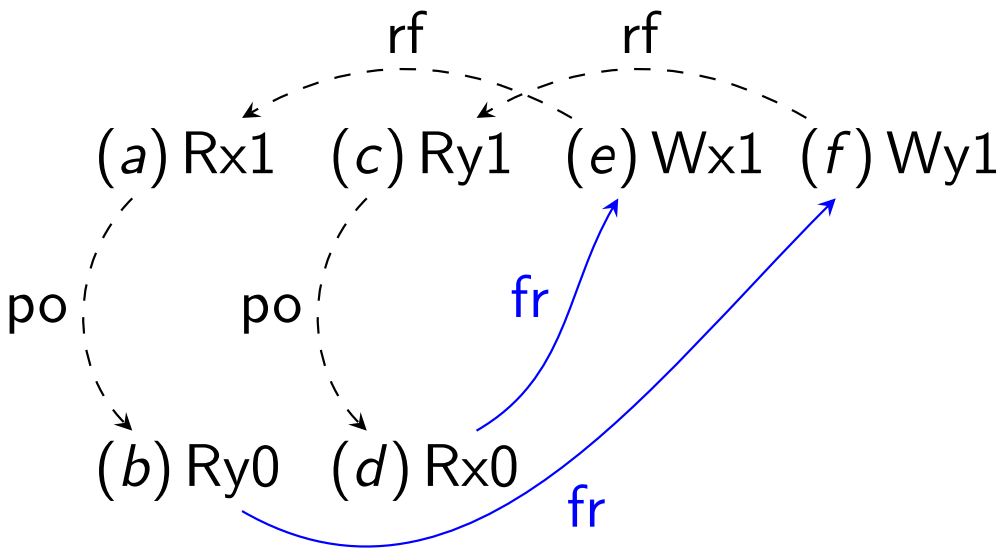


iriw on Power

iriw

P_0	P_1	P_2	P_3
(a) r1 $\leftarrow$ x (b) r2 $\leftarrow$ y	(c) r3 $\leftarrow$ y (d) r4 $\leftarrow$ x	(e) x $\leftarrow$ 1	(f) y $\leftarrow$ 2

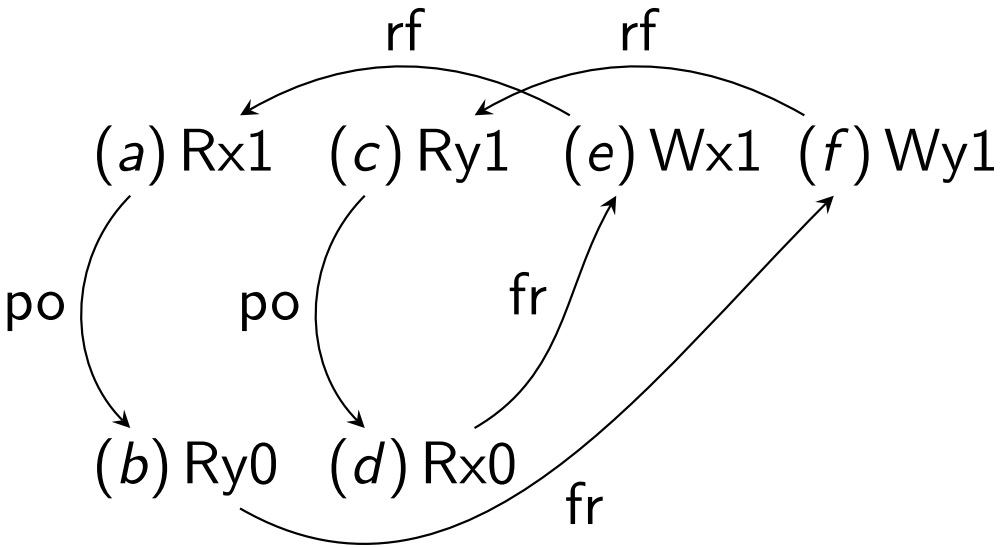
r1=1; r2=0; r3=2; r4=0;



Validity of an execution

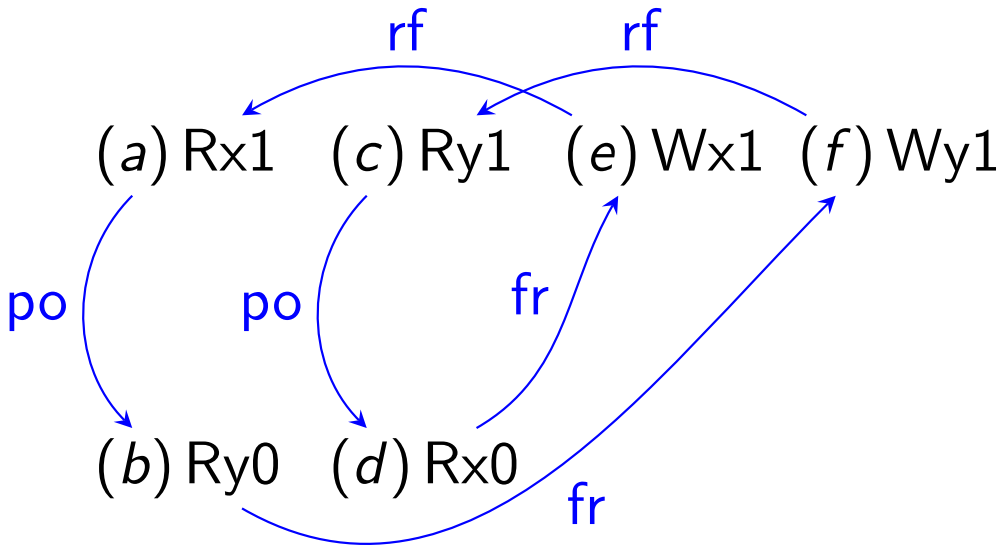
- An execution is valid on an architecture if it does not show certain cycles.
- So we assign a clock to each event
- Then see if we can order these clocks *w.r.t.* less-than over $\mathbb{N}$

On **iriw**



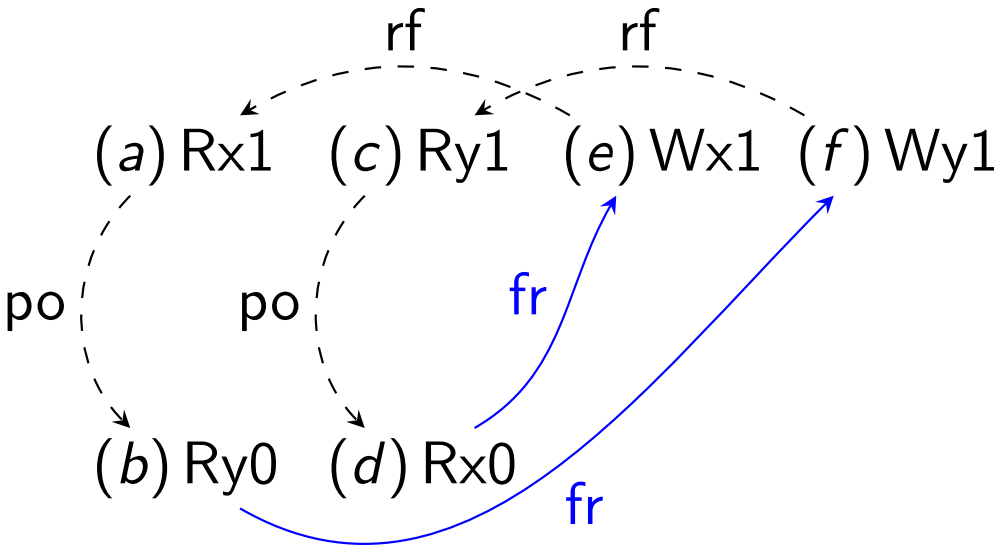
$$
\begin{array}{l}
(\text{po } P_0) \;\; c_{ab} \qquad (\text{po } P_1) \;\; c_{cd} \\
(\text{rf } x) \;\; s_{ea} \wedge s_{i_0 d} \qquad (\text{rf } y) \;\; s_{fc} \wedge s_{i_1 b} \\
(\text{ws } x) \;\; c_{i_0 e} \qquad (\text{ws } y) \;\; c_{i_1 f} \\
(\text{fr } x) \;\; (s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de} \qquad (\text{fr } y) \;\; (s_{i_1 b} \wedge c_{i_1 f}) \Rightarrow c_{bf} \\
(\text{grf } x) \;\; (s_{ea} \Rightarrow c_{ea}) \qquad (\text{grf } y) \;\; (s_{fc} \Rightarrow c_{fc})
\end{array}
\tag{1}
$$

iriw on SC



$$
\begin{array}{ll}
(\text{po } P_0) \;\; c_{ab} & (\text{po } P_1) \;\; c_{cd} \\
(\text{rf } x) \;\; s_{ea} \wedge s_{i_0 d} & (\text{rf } y) \;\; s_{fc} \wedge s_{i_1 b} \\
(\text{ws } x) \;\; c_{i_0 e} & (\text{ws } y) \;\; c_{i_1 f} \\
(\text{fr } x) \;\; (s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de} & (\text{fr } y) \;\; (s_{i_1 b} \wedge c_{i_1 f}) \Rightarrow c_{bf} \\
(\text{grf } x) \;\; (s_{ea} \Rightarrow c_{ea}) & (\text{grf } y) \;\; (s_{fc} \Rightarrow c_{fc})
\end{array}
\tag{2}
$$

iriw on Power



$$
\begin{array}{ll}
(\text{po } P_0) \;\; c_{ab} & (\text{po } P_1) \;\; c_{cd} \\
(\text{rf } x) \;\; s_{ea} \wedge s_{i_0 d} & (\text{rf } y) \;\; s_{fc} \wedge s_{i_1 b} \\
(\text{ws } x) \;\; c_{i_0 e} & (\text{ws } y) \;\; c_{i_1 f} \\
(\text{fr } x) \;\; (s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de} & (\text{fr } y) \;\; (s_{i_1 b} \wedge c_{i_1 f}) \Rightarrow c_{bf} \\
(\text{grf } x) \;\; (s_{ea} \Rightarrow c_{ea}) & (\text{grf } y) \;\; (s_{fc} \Rightarrow c_{fc})
\end{array}
\tag{3}
$$

Tools

Testing hardware, simulating models:

http://diy.inria.fr

Verifying software:

www.cprover.org/wmm

Thanks!