True Concurrency?

(not quite yet)

Peter Sewell

Computer Laboratory
University of Cambridge

Susmit Sarkar, Scott Owens, Tom Ridge, Magnus Myreen (U.Cambridge)
Francesco Zappa Nardelli, Jade Alglave, Thomas Braibant, Luc Maranget (INRIA)
Samin Ishtiaq (MSR)

Concurrency Theory Workshop, Jan. 2009

Relaxed Memory

Usually assume concurrent threads operate on a single shared memory (in language semantics, algorithms, program logics, model checking,...).

But for low-level programming of real multiprocessors, that's just not true:

- local write buffers
- pipelines with speculative execution and shadow register files
- hierarchies of cache
- ...

Also not true for high-level languages (C,C++,Java):

- concurrency-oblivious compiler optimisations

Relaxed Memory — a Simple Example

Shared x and y, initially 0.

P_0			P_1		
x	←	1	y	←	1
r_0	←	y	r_1	←	x

Experimentally:

On an 8-core POWER5: we see $r_0=r_1=0$ 196/2000 times

On an Intel Core 2 Duo: similar

Relaxed Memory — a Simple Example

Shared x and y, initially 0.

P_0			P_1		
x	←	1	y	←	1
r_0	←	y	r_1	←	x

Experimentally:

On an 8-core POWER5: we see $r_0=r_1=0$ 196/2000 times

On an Intel Core 2 Duo: similar

We can't think about these systems in terms of global time

Actual Processors

- particular h/w designs, e.g. Intel Core 2 Duo E6300
- can run programs on them
- may change quickly from design to design
- internally well-defined and well-understood (but secret)

“Architectures”

- descriptions of what programmers should rely on
  - Power ISA Version 2.05
  - Intel 64 and IA-32 Architectures SDM rev.28
  - AMD64 Architecture Programmer’s Manual rev.3.14
  - ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition
- claimed to cover a range of past (and future?) actual processors
- informal prose documents (+litmus tests)
- tension: reveal enough for programmers, but without constraining future design or exposing to liability

“Architectures”

- descriptions of what programmers should rely on

But:

- the guarantees are typically very subtle, and differ radically
- the informal prose makes it impossible to understand them precisely (informal prose is particularly bad for such very loose specs)
  - can’t run programs on them
  - can’t use as criterion for internal testing
- they may be incomplete (too weak to program above)
- they may be unsound (too strong w.r.t. actual processors)

So how does anything work? Architectures are typically supposed to be ok(?) for well-synchronised(?) programs.

Plan

- Establish semantics for multiprocessors
- Should be sound w.r.t. processors and strong enough for reasoning about racy code
- Use that for metatheory: DRF, algorithm verification,...

Current State

- x86-CC model [POPL09]. But Intel and AMD architecture docs turned out to be unsound and/or too weak — replacement in progress
- preliminary POWER and ARM model [DAMP09].

x86-CC

Pre-IWP	Nov 2006	Intel manuals, rev 22
IWP/Rev 26–28/AMD3.14	Aug 2007 Sep 2007	Intel white paper v1.0 AMD manual, rev 3.14
x86-CC		us

- Memory model and (core) instruction semantics in HOL
- Key issue: interpret “causality” in informal specs
- Testing: single-instruction and memory model
- Metatheory: data-race freedom, niceness, abstract machine

Rev 29	Nov 2008	Intel manuals, rev 29
(x86-TSO)	(in progress)	(us)

POWER and ARM

Draft memory model and instruction semantics in HOL and Coq

The memory model is an axiomatisation of the legal executions (as usual).

Much weaker (and more complex) than x86-CC or x86-TSO.

Everything not forbidden —no matter how bizarre— is permitted

Basic Types

event_structure $=\langle\!\![$ $events$: ($'reg$ event)set;
$intra\_causality\_data$: ($'reg$ event)reln;
$intra\_causality\_control$: ($'reg$ event)reln;
$atomicity$: ($'reg$ event)set set;
$arch$: architecture;
$granule\_size\_exponent$: num$]\!\!\rangle$

type_abbrev view_orders : proc $\rightarrow$ ($'reg$ event)reln

A view order for processor p is a strict linear order over all its events and other processor's memory writes.

execution_witness $=$
$\langle\!\![$ $initial\_state$: ($'reg$ state_constraint);
vo : ($'reg$ view_orders);
$write\_serialization$: ($'reg$ event reln)$]\!\!\rangle$

Read Values

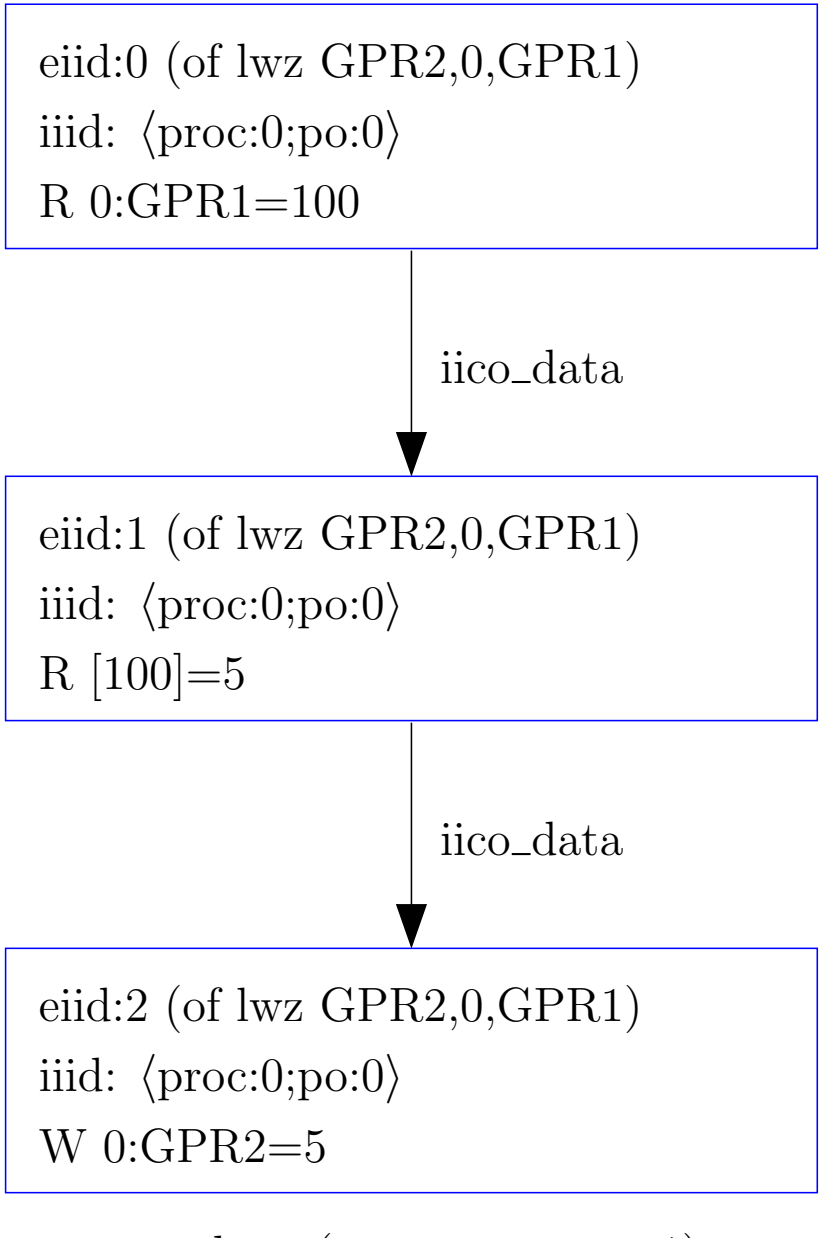
Any read event reads from the most recent (in that processor’s view order) write to that register or memory location, or the initial state.

Coherence

For any memory location, all processors see all writes to that location in the same order.

Instructions vs Events: intra-instruction data and control causality

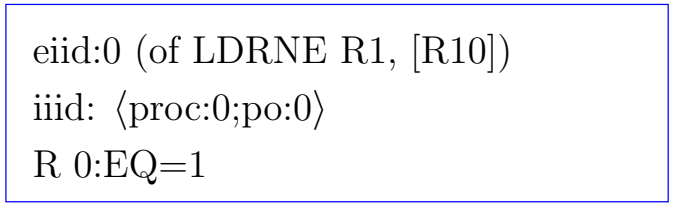
ppc-lwz	proc:0
poi:0	lwz GPR2,0,GPR1
Initial state: 0:GPR1= 100; [100]= 5	



Instructions vs Events: intra-instruction data and control causality

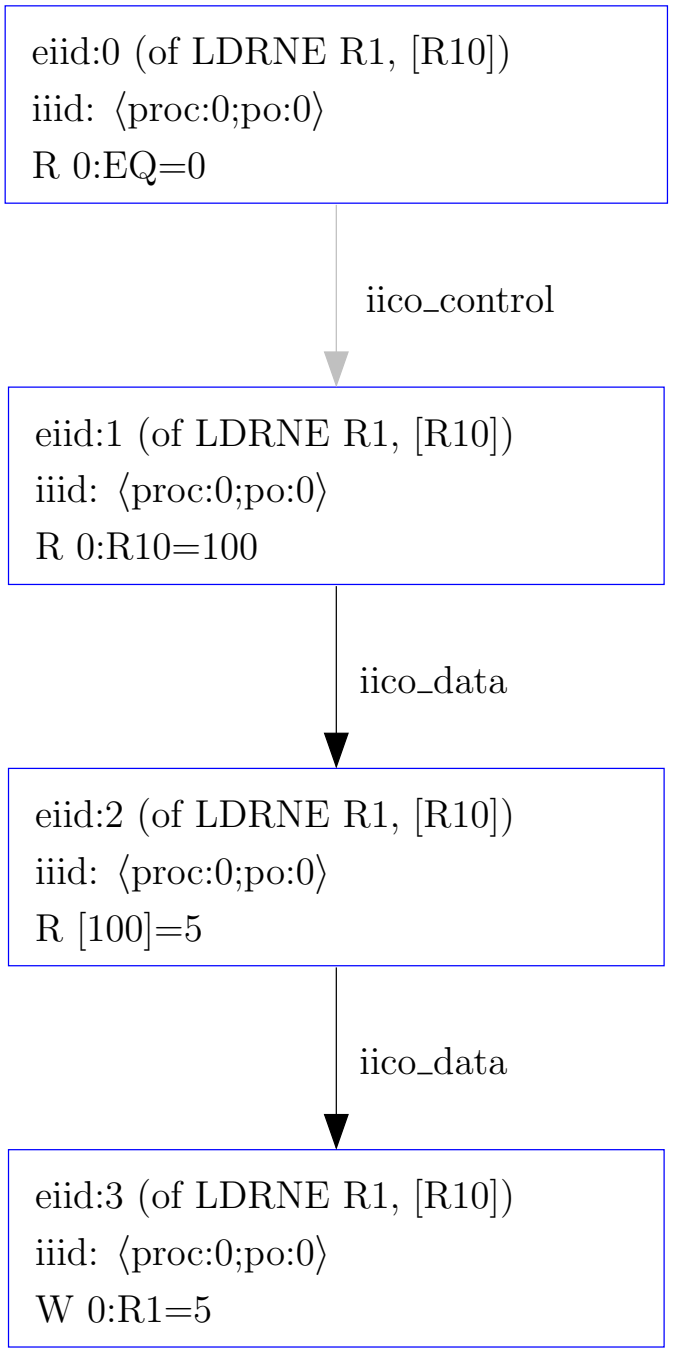
ARM-ldrne	proc:0
poi:0	LDRNE R1, [R10]

Either reads non-zero flag:



ARM-ldrne: (event structure 2)

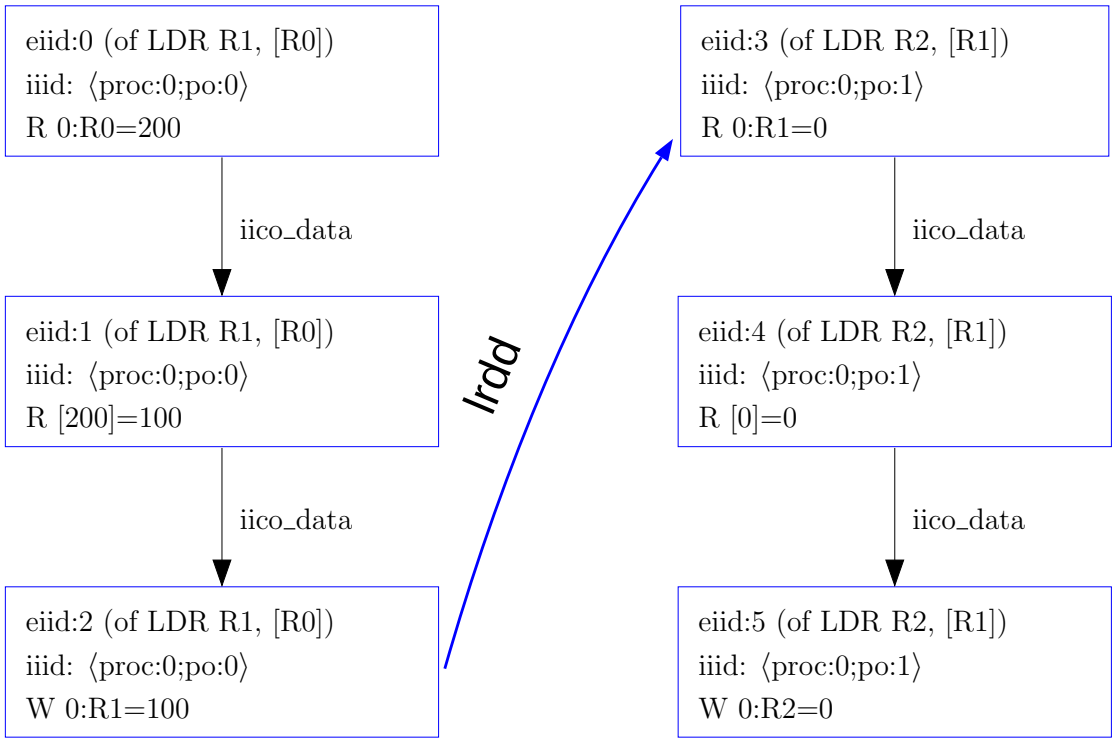
or reads a zero flag value, with an intra-instruction-causality-control relation to the read of R10:



ARM-ldrne: (event structure 1)

Local Register Data Dependency

ARM-depend-A	proc:0
poi:0	LDR R1, [R0]
poi:1	LDR R2, [R1]



ARM-depend-A: (event structure 1)

address_or_data_dependency_load_load

Dependencies from a memory load to a memory load, in program order, by the same processor (though perhaps to different addresses):

local register data dependency and intra-instruction data causality, transitively (not intra-instruction control).

For example, here the LDRNE instruction is conditional on the EQ flag.

ARM-nodep	proc:0
poi:0	LDR R1, [R4]
poi:1	CMP R1, #1
poi:2	LDRNE R2, [R5]

but even if the value read from that flag is 0, there is no address_or_data_dependency_load_load from the first memory load to the memory load of the LDRNE.

address_or_data_dependency_load_load

However, here there is, even though it's a "false" dependency:

ARM-depend-B	proc:0
poi:0	LDR R1, [R0]
poi:1	AND R1, R1, #0
poi:2	LDR R2, [R3,R1]

address_or_data_or_control_dependency_load_store

Dependencies from a memory load to a memory store are similar, except that here *both* intra-instruction relations are included:

Here the execution (or not) of the memory store depends on the value returned by the memory load.

ARM-depend-C	proc:0
poi:0	LDR R1, [R0]
poi:1	CMP R1, #55
poi:2	STRNE R2, [R3]

Preserved Program Order

$$\begin{aligned}
&\text{preserved_program_order}\ E\ p = \\
&\{(e_1, e_2) \mid (e_1, e_2) \in E.\mathit{intra_causality_data} \wedge \\
&\qquad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p)\} \cup \\
&\{(e_1, e_2) \mid (e_1, e_2) \in E.\mathit{intra_causality_control} \wedge \\
&\qquad (\text{proc}\ e_1 = p) \wedge (\text{proc}\ e_2 = p)\} \cup \\
&\text{address_or_data_dependency_load_load}\ E\ p\ \cup \\
&\text{address_or_data_or_control_dependency_load_store}\ E\ p\ \cup \\
&\text{preserved_program_order_mem_loc}\ E\ p
\end{aligned}$$

(the last condition orders all p's memory accesses to the same location)

No global order on register accesses(!)

Barriers

Power: sync, lwsync, eieio

ARM: DMB

A Glimpse of the Informal Specs

The Power specification says the ordering provided by a barrier is “cumulative”, i.e.

> “it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.
>
> - A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
> - B includes all applicable storage accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.”

Here “performed” is defined informally as follows:

> "A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2.
>
> A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). [...] The preceding definitions apply regardless of whether P1 and P2 are the same entity."

Not This (though maybe something like it)

$$
\begin{aligned}
&\text{check\_sync\_power\_2\_05 } E\ vos = \\
&\quad \forall es \in (E.events).(es.action = \textsc{Barrier Sync}) \implies \\
&\quad \textbf{let } group\_A = \{e \mid ((e, es) \in \text{ po } E \vee (e, es) \in vos(\text{proc } es)) \wedge \text{mem\_access } e\} \textbf{ in} \\
&\quad \textbf{let } group\_B\_base = \{e \mid (es, e) \in \text{ po } E \wedge \text{mem\_access } e\} \textbf{ in} \\
&\quad \textbf{let } group\_B\_ind\ B_0 = \\
&\qquad \{e \mid \text{mem\_access } e \wedge \\
&\qquad\quad (\neg(\text{proc } e = \text{proc } es)) \wedge \\
&\qquad\quad \exists er.\, \text{mem\_load } er \wedge (er, e) \in vos(\text{proc } er) \wedge (\text{proc } er = \text{proc } e) \wedge \\
&\qquad\qquad \exists ew.\, \text{mem\_store } ew \wedge ew \in B_0 \wedge (ew, er) \in vos(\text{proc } er) \wedge (\text{loc } er = \text{loc } ew) \wedge \\
&\qquad\qquad\quad (\neg(\exists ew'.(ew, ew') \in vos(\text{proc } er) \wedge (ew', er) \in vos(\text{proc } er) \wedge \\
&\qquad\qquad\qquad (\text{loc } ew' = \text{loc } er) \wedge \text{mem\_store } ew'))\} \textbf{ in} \\
&\quad \textbf{let } group\_B = \text{FIX } group\_B\_ind\ group\_B\_base \textbf{ in} \\
&\quad \forall p \in (\text{procs } E).\forall ea \in group\_A.\forall eb \in group\_B.(ea \in \text{viewed\_events } E\ p \wedge eb \in \text{viewed\_events } E\ p) \implies \\
&\qquad \textbf{if } (p = es.iiid.\,\text{proc}) \textbf{ then } ((ea, es) \in vos\ p \wedge (es, eb) \in vos\ p) \textbf{ else } (ea, eb) \in vos\ p
\end{aligned}
$$

Reservations

Power: `lwarx`/`stwcx`

ARM: `LDREX`/`STREX`

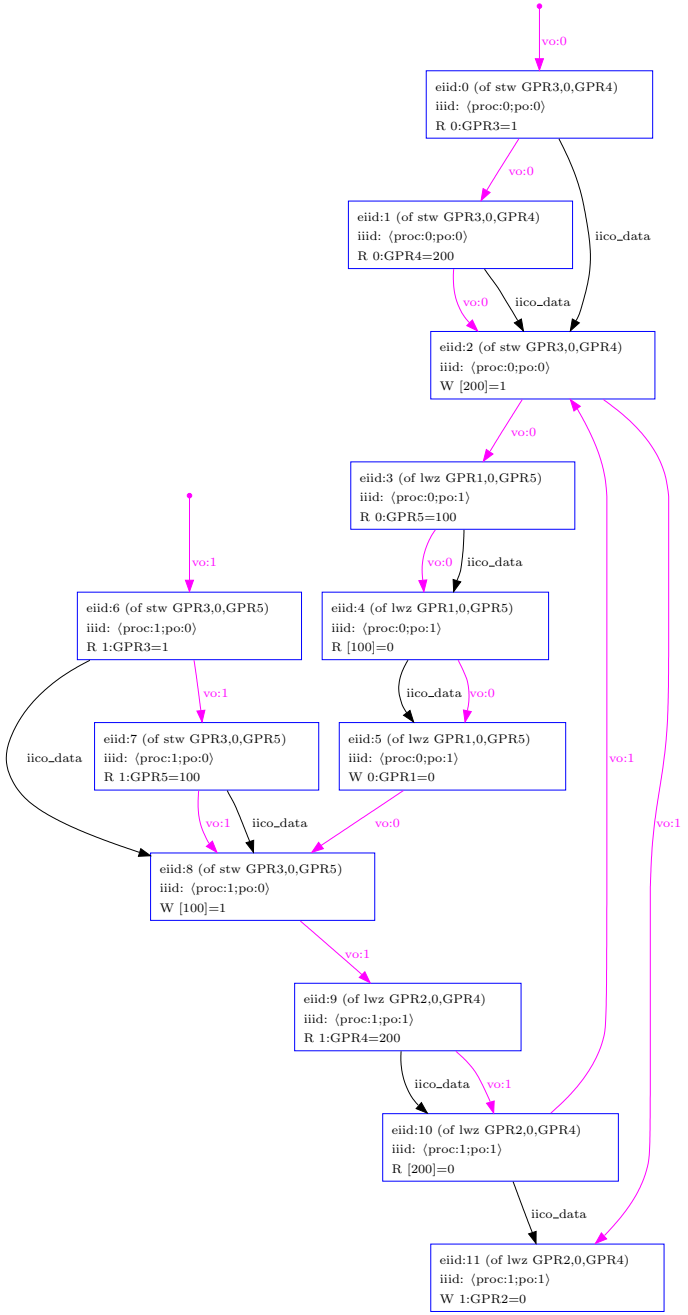
more plumbing...

Valid Executions

$$
\begin{aligned}
&\text{valid\_execution } E\ X = \\
&\quad \text{view\_orders\_well\_formed } E\ X.vo \wedge \\
&\quad \text{read\_most\_recent\_value } E\ X.initial\_state\ X.vo \wedge \\
&\quad X.write\_serialization \in \text{write\_serialization\_candidates } E \wedge \\
&\quad (\forall p \in (\text{procs } E). \\
&\qquad \text{preserved\_coherence\_order } E\ p \subseteq X.write\_serialization \wedge \\
&\qquad X.write\_serialization \subseteq X.vo\ p \wedge \\
&\qquad \text{preserved\_program\_order } E\ p \subseteq X.vo\ p \wedge \\
&\qquad \text{(*no intervening writes in local register data dependency*)} \\
&\qquad \text{local\_register\_data\_dependency } E\ p \subseteq X.vo\ p \wedge \\
&\qquad (\forall (e_1, e_2) \in (\text{local\_register\_data\_dependency } E\ p). \\
&\qquad\quad \neg(\exists e_3.(e_1, e_3) \in X.vo\ p \wedge (e_3, e_2) \in X.vo\ p \wedge \\
&\qquad\qquad\quad (\text{loc } e_3 = \text{loc } e_1) \wedge \text{store } e_3))) \wedge \\
&\quad \text{(*no intervening writes before a reg read from initial state*)} \\
&\quad (\forall e \in (E.events).(\text{reg\_load } e \wedge \\
&\qquad (\neg(\exists e_0.(e_0, e) \in \text{po\_iico\_both } E \wedge \text{reg\_store } e_0 \wedge \\
&\qquad\quad (\text{loc } e_0 = \text{loc } e)))) \implies \\
&\qquad (\neg(\exists e_0.(e_0, e) \in X.vo(\text{proc } e) \wedge \text{reg\_store } e_0 \wedge \\
&\qquad\quad (\text{loc } e_0 = \text{loc } e))) \wedge \\
&\quad \text{(*no intervening writes after a reg write to the final state*)} \\
&\quad (\forall e \in (E.events).(\text{reg\_store } e \wedge \\
&\qquad (\neg(\exists e_1.(e, e_1) \in \text{po\_iico\_both } E \wedge \text{reg\_store } e_1 \wedge \\
&\qquad\quad (\text{loc } e_1 = \text{loc } e)))) \implies \\
&\qquad (\neg(\exists e_1.(e, e_1) \in X.vo(\text{proc } e) \wedge \text{reg\_store } e_1 \wedge \\
&\qquad\quad (\text{loc } e_1 = \text{loc } e))) \wedge
\end{aligned}
$$

Example Valid Execution

ppc3.1	proc:0	proc:1
poi:0	stw GPR3,0,GPR4	stw GPR3,0,GPR5
poi:1	lwz GPR1,0,GPR5	lwz GPR2,0,GPR4
Initial state: 0:GPR3= 1; 0:GPR4= 200; 0:GPR5= 100; 1:GPR3= 1; 1:GPR4= 200; 1:GPR5= 100 (elsewhere 0)		
Allowed: 0:GPR1=0 ∧ 1:GPR2=0		



ppc3.1: (event structure 1)

Other Things

Litmus tool.

Memevents tool.

Instruction Semantics — for reasonable subsets. Monadicised.

Metatheory — single-processor conjecture (in progress)

Conclusion

Towards precise and accurate semantics for real multiprocessors

(necessary foundation for any verification of low-level code)

Broader Moral: don’t believe any memory model, unless (at least!):

- it’s formalised (preferably mechanised)
- it’s mechanically run on many litmus examples
- it’s extensively tested against actual processors
- there’s (preferably mechanised) metatheory
- there’s extensive programming above the model