An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors

Kathryn E. Gray\textsuperscript{1}  Gabriel Kerneis\textsuperscript{1+}  Dominic Mulligan\textsuperscript{1}
Christopher Pulte\textsuperscript{1}  Susmit Sarkar\textsuperscript{2}  Peter Sewell\textsuperscript{1}

\textsuperscript{1} University of Cambridge  \textsuperscript{1+} During work  \textsuperscript{2} University of St Andrews
What is an architecture spec?
What is an architecture spec?

Typically prose
What is an architecture spec?

Typically prose
What is an architecture spec?

Typically prose

Sometimes pseudocode
What is an architecture spec?

Typically prose

Sometimes pseudocode

<table>
<thead>
<tr>
<th>Branch</th>
<th>I-form</th>
<th>Branch Conditional</th>
<th>B-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>target_addr</td>
<td>(AA=0, LK=0)</td>
<td>BD:Bl:target_addr</td>
</tr>
<tr>
<td>ba</td>
<td>target_addr</td>
<td>(AA=1, LK=0)</td>
<td>BD:Bl:target_addr</td>
</tr>
<tr>
<td>l</td>
<td>target_addr</td>
<td>(AA=0, LK=1)</td>
<td>BD:Bl:target_addr</td>
</tr>
<tr>
<td>lla</td>
<td>target_addr</td>
<td>(AA=1, LK=1)</td>
<td>BD:Bl:target_addr</td>
</tr>
</tbody>
</table>

Extended Mnemonics:

Example of extended mnemonics for Branch Conditional:

Extended:  Equivalent to:

```plaintext
blt target bc 12,0,target
bne cr2,target bc 4,10,target
bdnz target bc 16,0,target
```

Extended:  Equivalent to:

```plaintext
bc 12,0,target
bne 4,10,target
bdnz 16,0,target
```

Extended:  Equivalent to:

```plaintext
18 LI AA LK
06 3 0 3 1
```

Extended:  Equivalent to:

```plaintext
16 BO BI BD AA LK
06 1 1 1 6 3 0 3 1
```
But ...
But …

- *Not* executable test oracles
  - You can’t test h/w or s/w against prose
But …

- *Not* executable test oracles
  - You can’t test h/w or s/w against prose
- *Not* a clear guide to concurrent behaviour
  - Especially for weakly consistent IBM Power and ARM
But …

• *Not* executable test oracles
  • You can’t test h/w or s/w against prose

• *Not* a clear guide to concurrent behaviour
  • Especially for weakly consistent IBM Power and ARM

• A mass of instruction set detail
Specification as Artefact

We (show how to) make architecture specs that are *real* technical artefacts
Specification as Artefact

We (show how to) make architecture specs that are *real* technical artefacts

- Executable as test oracle
Specification as Artefact

We (show how to) make architecture specs that are real technical artefacts

- Executable as test oracle
- Mathematically precise
Specification as Artefact

We (show how to) make architecture specs that are real technical artefacts

• Executable as test oracle
• Mathematically precise
• Related to vendor pseudocode and intuition
Specification as Artefact

We (show how to) make architecture specs that are real technical artefacts

- Executable as test oracle
- Mathematically precise
- Related to vendor pseudocode and intuition
- Clarify interface between ISA and concurrency
Specification as Artefact

We (show how to) make architecture specs that are real technical artefacts

Specifically IBM POWER
all non-FP non-vector "user" ISA (153 instructions)
and concurrency model
Specification as Artefact

We (show how to) make architecture specs that are *real* technical artefacts

Specifically IBM POWER
all non-FP non-vector "user" ISA (153 instructions)
and concurrency model

Applicable to ARM as well
See Modelling the ARMv8 Architecture, Operationally
Concurrency and ISA, POPL16
Not just an emulator
Not just an emulator

Emulator

PPCMEM2
Not just an emulator

Emulator

PPCMEM2

Written in C etc
A language with many faults
Intermingling of emulation detail & semantics
<table>
<thead>
<tr>
<th>Emulator</th>
<th>PPCMEM2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written in C etc</td>
<td>Written in Lem &amp; Sail</td>
</tr>
<tr>
<td>A language with many faults</td>
<td>Languages for logic, mathematics, and ISAs</td>
</tr>
<tr>
<td>Intermingling of emulation detail &amp; semantics</td>
<td>Only spec detail</td>
</tr>
<tr>
<td></td>
<td>Emulation separated</td>
</tr>
</tbody>
</table>
Not just an emulator

Emulator

PPCMEM2

Running concurrent code:
Consider a lock
Not just an emulator

Emulator

PPCMEM2

T1 Lock
T1 Set critical section
T1 Unlock
T2 Lock
...

repeat

Running concurrent code:
Consider a lock
Not just an emulator

Emulator

T1 Lock
T1 Set critical section
T1 Unlock
T2 Lock
...

repeat

Running concurrent code:
Consider a lock
Beneficiaries

• Compiler writers
• Concurrency primitive implementors
• Security developers
• Hardware developers
In this paper we show how a precise architectural envelope model for a weakly consistent architecture can be used to verify programs, not of more general code.

1.3 Contribution

The most fun-

The most fun-
**Store Word with Update**  
**D-form**

\[
\begin{array}{c|c|c|c|c}
\text{stwu} & \text{RS,D(RA)} \\
\hline
37 & 6 & 11 & 16 & 31 \\
\end{array}
\]

EA \leftarrow (RA) + \text{EXTS(D)}  
\text{MEM(EA, 4)} \leftarrow (RS)_{32:63}  
RA \leftarrow \text{EA}

Let the effective address (EA) be the sum (RA)+ D. 
\((RS)_{32:63}\) are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**  
None

**Store Word Indexed**  
**X-form**

\[
\begin{array}{c|c|c|c|c}
\text{stwx} & \text{RS,RA,RB} \\
\hline
36 & 6 & 11 & 16 & 31 \\
\end{array}
\]

function clause decode (0b100101 :  
(bit[5]) RS :  
(bit[5]) RA :  
(bit[16]) D as instr) =  
Stwu (RS,RA,D)

function clause execute (Stwu (RS, RA, D)) = {  
(b[64]) EA := 0;  
EA := GPR[RA] + \text{EXTS(D)};  
GPR[RA] := EA;  
MEMw(EA,4) := (GPR[RS])[32 .. 63]  
}
Sample Instruction

**Store Word with Update**  
*D-form*

```
stwu RS,D(RA)
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>RS</td>
<td>RA</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)_{32:63}
RA ← EA

Let the effective address (EA) be the sum (RA)+ D. (RS)_{32:63} are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None
Store Word with Update  D-form

stwu RS,D(RA)

<table>
<thead>
<tr>
<th>37</th>
<th>RS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)_{32:63}
RA ← EA

Let the effective address (EA) be the sum (RA)+ D. (RS)_{32:63} are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

union ast member (bit[5], bit[5], bit[16]) Stwu

function clause decode (0b100101:
   (bit[5]) RS :
   (bit[5]) RA :
   (bit[16]) D as instr) =

   Stwu (RS,RA,D)

function clause execute (Stwu (RS, RA, D)) = {
   (bit[64]) EA := 0;
   EA := GPR[RA] + EXTS(D);
   GPR[RA] := EA;
   MEMw(EA,4) := (GPR[RS])[32 .. 63] }
Sample Instruction

**Store Word with Update D-form**

```c
stwu RS,D(RA)     
EA I  (RA) + EXTS(D)
MEM(EA, 4) I  (RS) 32:63
RA I  EA
```

Let the effective address (EA) be the sum (RA) + D.
(RS)_{32:63} are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None

```
union ast member (bit[5], bit[5], bit[16]) Stwu

function clause decode (0b100101 :
    (bit[5]) RS :
    (bit[5]) RA :
    (bit[16]) D as instr) =

    Stwu (RS,RA,D)

function clause execute (Stwu (RS, RA, D)) = {
    (bit[64]) EA := 0;
    EA := GPR[RA] + EXTS(D);
    GPR[RA] := EA;
    MEMw(EA,4) := (GPR[RS])[32 .. 63]
    }
```
Sail:
for specifying concurrent ISAs

- C-like/ISA Pseudo-code like imperative language with
  - Built-in understanding of registers and memory
  - Type inference, including vector-size checking
- Formal interpreter
- Executable for sequential or concurrent exploration
  - Analyses instructions for register/memory footprint
Do that of \[3\]) with an architectural model for all of the programs, not of more general code. That maintains a tree of in-flight and committed instructions, expressing the programmer-visible aspects of out-of-order and speculative computation; this is tied to that description, we took an XML version extended to the instruction semantics.

The most fundamental question we address is what the interface be (\(\text{CC} \mapsto \text{ISA}\) should be between the concurrency model and ISA semantics, which we summarise here and detail below.

### ISA Description Tied to Vendor Documentation

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.

The vendor specification for the POWER architecture is provided as a PDF document \[20\] produced from Framemaker sources. To keep our ISA model closely aligned with ad hoc variations and patching the results, so producing a precise Sail definition required dealing with these issues with the interface. Lem (Sail AST) is provided as a PDF document \[20\] produced from Framemaker sources.
In this paper we show how a precise architectural envelope model for a weakly consistent architecture can be used to verify the intended concurrency model in the process; this permits instruction descriptions to be expressed in that familiar imperative style while simultaneously simplifying assumptions. In particular, it includes only similar to POWER, though not identical).

We have also discovered errata in a number of multiprocessor pages of instruction description, and it is important that the concurrency model, we introduce a new instruction set, and discuss what is required instead, with a series of some aspects of out-of-order and speculative execution, and of the non-multi-copy atomic storage subsystem, and TSO multiprocessor, and SC multiprocessor, and TSO multiprocessor.

The most fun-
ISA + Concurrency Challenges

- No single program point
- No per-thread register state
- Register shadowing effects
- and more
### No Per-thread register state

<table>
<thead>
<tr>
<th></th>
<th>MP+sync+rs</th>
<th>POWER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td>stw r7,0(r1) # x=1</td>
<td>lwz r5,0(r2) # r5=y</td>
</tr>
<tr>
<td></td>
<td>sync # sync</td>
<td>mr r6,r5 # r6=r5</td>
</tr>
<tr>
<td></td>
<td>stw r8,0(r2) # y=1</td>
<td>lwz r5,0(r1) # r5=x</td>
</tr>
</tbody>
</table>

**Initial state:**
- 0:r1=x, 0:r2=y, 0:r7=1,
- 0:r8=1, 1:r1=x, 1:r2=y, x=0

**Allowed:**
- 1:r6=1, 1:r5=0
In a way that achieves all the desirable properties of \cite{3} with an architectural model for all of the POWER implementations of POWER and ARM architectures, together with related research on axiomatic models \cite{10}. This work (together with that of production or pre-silicon hardware, on pre- and post-silicon (ARM concurrency is broadly a tiny fragment of the POWER instruction set, and simplifying assumptions. In particular, it includes only that maintains a tree of in-flight and committed instructions. Then there is a model for each hardware thread against several generations of POWER implementations that looks like it is written in a sequential imperative language (of abstract micro-operations such as assign-ments to registers, arithmetic operations, etc.). They are aware of deals with these issues with the interface semantics (

Extended Concurrency Model

That model has been experimentally validated so producing a precise Sail definition required dealing with ad hoc variations and patching the results. But for weakly consistent processors, some aspects of out-of-order and speculative execution, some components of cache coherence, and of the non-multi-copy atomic storage subsystem, state-update model for instructions. We explain this, permitting instruction descriptions to be expressed precisely defined and expressive type system, using type inference to check pseudocode consistency while keeping instructions simply as updating a global register and memory state. The same holds for a sequentially consistent processor, where actions happen in the order they are issued, but the memory state may be visible to a thread only after it has been committed to.

The most fundamental question we address is what the interface between the concurrency model and ISA semantics should be (§3). Sail has a pre-ff effects and causal semantics, and discussions of instruction behaviour are traditionally expressed in a combination of prose and of pseudocode. For a single-threaded processor one can regard the ISA model Litmus frontend

Concurrency model

Storage semantics

Thread semantics

System semantics

Harness

Text UI

Web UI

OCaml, CSS, JS

evaluations
That previous model makes many major simplifying assumptions. In particular, it includes only a tiny fragment of the POWER instruction set, and effectively it is given only an ad hoc semantics and only that is produced in a way that achieves all the desirable properties.

Unfortunately, that previous concurrency model of Sarkar et al. to this larger and more complicated problem. It has also discovered errata in a number of multiprocessor implementations of POWER and ARM architectures, together with related research on axiomatic models [10].

This work (together with related research on axiomatic models [10]) has also been used by Linux-kernel software developers [9]. This work (together with related research on axiomatic models [10]) has also been used by Linux-kernel software developers [9].

For the results to be useful, they have to be directly usable by software developers who interact with the hardware. For this purpose, we need a tool for verifying software that makes assumptions about the hardware, such as data-race-free code. That tool has been experimentally validated with that of production or pre-silicon hardware, on both pre- and post-silicon (ARM concurrency is broadly similar to POWER, though not identical).

The most fun-
In this paper we show how a precise architectural endor-

Concurrency model

- Storage semantics
  - Lem
- System semantics
  - Lem
- Thread semantics
  - Lem

Maintains tree of in-flight instructions
In this paper we show how a precise architectural environment for simple litmus-test programs, not of more general code. The Litmus tool of Alglave and Maranget with ad hoc variations and patching the results. That model has been experimentally validated by Linux-kernel software developers. This work (together with related research on axiomatic models) has also discovered errata in a number of multiprocessor architectures. Together these form an abstract machine that maintains a tree of in-flight and committed instructions.

The vendor specification for the POWER architecture is provided as a PDF document produced from Framemaker sources. To keep our ISA model closely tied to that description, we took an XML version extracted and analyses the instruction descriptions from it, producing Sail definitions of the decoding and instruction semantics.

Concise defined and expressive type system, using type inference to check pseudocode consistency while keeping the concurrency model, we introduce a new instruction description language (IDL), Sail (Sail AST). Sail has a precise, invariant semantics.

The most fun-
In this paper we show how a precise architectural envelope model for a weakly consistent architecture can fines the architectural behaviour for simple litmus-test executions at an assembly level; and it does not handle mixed-size instruction instances, expressing the programmer-visible aspects of out-of-order and speculative computation; this extends the concurrency model in the process; and it has been validated mathematically by using it in a tiny fragment of the POWER instruction set, and simplifying assumptions. In particular, it includes only implementations of POWER and ARM architectures, has also discovered errata in a number of multiprocessor together with related research on axiomatic models [10] by Linux-kernel software developers [9]. This work (to-compile to POWER [8, 4]; and the tool has been used verifying the intended concurrency model in the process); extensively with a senior IBM architect (clarifying tool of Alglave and Maranget [5], using the Litmus diiy tool of Alglave and Maranget [5], using the Litmus hand-written litmus tests and on tests produced by the (G5, 5, 6, 7, and 8), comparing the model behaviour against several generations of POWER implementations with a state and transitions.

architecture. Together these form an abstract machine abstracts from pipeline and local store queue microar-
spects of out-of-order and speculative computation; this

In this paper we show how a precise architectural en-
velope model for a weakly consistent architecture can fines the architectural behaviour for simple litmus-test executions at an assembly level; and it does not handle mixed-size instruction instances, expressing the programmer-visible aspects of out-of-order and speculative computation; this extends the concurrency model in the process; and it has been validated mathematically by using it in a tiny fragment of the POWER instruction set, and simplifying assumptions. In particular, it includes only implementations of POWER and ARM architectures, has also discovered errata in a number of multiprocessor together with related research on axiomatic models [10] by Linux-kernel software developers [9]. This work (to-compile to POWER [8, 4]; and the tool has been used verifying the intended concurrency model in the process); extensively with a senior IBM architect (clarifying tool of Alglave and Maranget [5], using the Litmus diiy tool of Alglave and Maranget [5], using the Litmus hand-written litmus tests and on tests produced by the (G5, 5, 6, 7, and 8), comparing the model behaviour against several generations of POWER implementations with a state and transitions.
In this paper we show how a precise architectural end-ff effects the architectural behaviour for simple litmus-test memory and register accesses. E at an assembly level; and it does not handle mixed-size a tiny fragment of the POWER instruction set, and similar to POWER, though not identical).

Implementations of POWER and ARM architectures, has also discovered errata in a number of multiprocessor together with related research on axiomatic models [10]) by Linux-kernel software developers [9]. This work (to-compiled to POWER [8, 4]; and the tool has been used proof that C/C++11 concurrency [7] can be correctly it has been validated mathematically by using it in a specifying the intended concurrency model in the process); extensive discussion with a senior IBM architect (clar-diagonol tool of Alglave and Maranget [5], using the Litmus test harness [6]. It has been validated intensionally by [11, 12, 13, 14, 15, 16, 17, 18, 19], but none that we concurrent POWER examples. There has been a great and discuss what is required instead, with a series of are aware of deals with these issues with the interface some aspects of out-of-order and speculative execution, some additions to registers, arithmetic operations, etc.). They be (as in x86 and Sparc) requires only the addi-parallel consistency (SC) multiprocessor, and TSO multiprocessor memory state. The same holds for a sequentially con-
POWER Memory Model

PPC MP+sync+ctrl
"SyncdWW Rfe DpCtrlldR Fre"
Cycle=Rfe DpCtrlldR Fre SyncdWW
{
0:r2=x; 0:r4=y;
1:r2=y; 1:r4=x;
}
P0 | P1
li r1,1  | lwz r1,0(r2);
stw r1,0(r2) | cmpw r1,r1;
sync | beq LC00;
li r3,1  | LC00;
stw r3,0(r4) | lwz r3,0(r4);
exists
(1:r1=1) /
1:r3=0)
Storage subsystem state:
- writes seen = \{W\/x=8=0, W\/y=8=0, W\/x=4=1\}
- coherence = \{W\/x=8\rightarrow W\/x=4=1\}
- writes_past_coherence_point = \{W\/x=8=0, W\/y=8=0\}

Events propagated to:
- Thread 0: [W\/y=8=0 [0-7], W\/x=8=0 [0-7], W\/x=4=1 [0-3], Sync ]
- Thread 1: [W\/y=8=0 [0-7], W\/x=8=0 [0-7]]

2. Propagate write to thread: W\/x=4=1 [0-3] to Thread 1

Unacknowledged Sync requests = \{Sync\}

Thread 0 state:
- unacknowledged Syncs = \{Sync\}
- Old instructions
  - ioid: 4 loc: 0x00000000000050000 addi RT=1 RA=0 SI=1
  - ioid: 5 loc: 0x00000000000050004 stw RS=1 RA=2 D=0
- New instructions

Thread 1 state:
- Unacknowledged Syncs = \{}
- Initial fetch address: 0x00000000000051000 (from notional predecessor ioid 2)

Enabled transitions:
- 0: (0:8) Register read: GPR4 = y from initialstate of 0-63:Y
- 1: (1:2) Fetch from address 0x00000000000051000 lwz RT=1 RA=2 D=0
- 2: Propagate write to thread: W\/x=4=1 [0-3] to Thread 1
Validation

Sequential single instruction

  6983 tests of fixed-point user-mode instruction

Concurrent litmus tests

  2175 tests run exhaustively
    including those from prior concurrency models
Conclusions

Combined ISA and concurrency model for IBM POWER

Developed w.r.t existing h/w & in consultation with architects

Usable as reference model for future h/w & s/w

Usable for verification

Relevant for ARM and future models

http://www.cl.cam.ac.uk/~pes20/ppcmem2/