Understanding POWER multiprocessors

Susmit Sarkar\textsuperscript{1}  Peter Sewell\textsuperscript{1}
Jade Alglave\textsuperscript{2,3}  Luc Maranget\textsuperscript{3}
Derek Williams\textsuperscript{4}

\textsuperscript{1}University of Cambridge \hspace{1cm} \textsuperscript{2}Oxford University \hspace{1cm} \textsuperscript{3}INRIA \hspace{1cm} \textsuperscript{4}IBM

June 2011
No Sequential Consistency (SC) and not since 1972

But what do we get?

“Relaxed Memory”, differing on different architectures:
- x86, SPARC — Relatively strong, better understood;
- POWER/ARM — Weaker, widely used, not widely understood;
- High-level languages — Different again

Models informed by POWER/ARM features
## Relaxed memory behaviour: Message Passing

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1 )</td>
<td>while ((y == 0))</td>
</tr>
<tr>
<td>( y = 1 )</td>
<td>{};</td>
</tr>
<tr>
<td></td>
<td>( r = x ) (read 0?)</td>
</tr>
</tbody>
</table>

### Test MP: Allowed

- **Thread 0**
  - a: \( W[x]=1 \)
  - b: \( W[y]=1 \)
  - po
  - rf

- **Thread 1**
  - c: \( R[y]=1 \)
  - d: \( R[x]=0 \)
  - po

- **Forbidden on SC, or x86-TSO**
- **Allowed on POWER (\(\sim 1\times 10^6\) in \(2\times 10^9\) on a POWER7)**

Test MP: Allowed
What is going on?

Visible Microarchitectural Effects:

- Out-of-order, and Speculative Execution
- Buffering of Stores and Loads
- Topology of Interconnection
Enforcing order where needed

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>while (y == 0)</td>
</tr>
<tr>
<td>sync()</td>
<td>{}</td>
</tr>
<tr>
<td>y = &amp;x</td>
<td>r = *y (read 0?)</td>
</tr>
</tbody>
</table>

- **sync**: writes in order
  - On the same thread; and
  - When propagating to other threads

- **Dependency**: reads in order
  - Later read not issued until resolved

---

Test MP+sync+addr : Forbidden
POWER model in general: ... How do we find out?

Architecture Manuals:
- Ambiguous prose

  "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

Concrete Implementation:
- Proprietary
- Extremely complex, and too low-level
- Changes across generations

1 Not Derek Williams!
Rigorous Architecture

Do lots of tests (borrow, handwrite, autogenerate) on Power G5, 6, and 7

Discuss with designers/architects

Develop an abstract operational model
  - Matches observed behaviour (intentionally looser in some aspects)
  - Simple enough to understand

Only considering application and common OS code, with no unaligned/mixed-size accesses (no self-modifying code, device memory, or page table changes)
The model structure

Overall structure:

- Write request
- Barrier request
- Write announce
- Barrier ack

- Threads and Storage Subsystem: Abstract state machines
  - Speculative execution in Threads;
  - Topology-independent Storage Subsystem
- Formally: transitions, guarded by preconditions, change state, and synchronize with each other

*Some aspects are thread-only, some storage-only, some both*
The sync is *cumulative*: it keeps (a) and (c) in order for all threads. Flipping the dependency and barrier does *not* recover SC.
Propagate write to another thread
The storage subsystem can propagate a write \( w \) (by thread \( tid \)) that it has seen to another thread \( tid' \), if:

- the write has not yet been propagated to \( tid' \);
- \( w \) is coherence-after any write to the same address that has already been propagated to \( tid' \); and
- all barriers that were propagated to \( tid \) before \( w \) (in \( s.events_propagated_to (tid) \)) have already been propagated to \( tid' \).

**Action:** append \( w \) to \( s.events_propagated_to (tid') \).

**Explanation:** This rule advances the thread \( tid' \) view of the coherence order to \( w \), which is needed before \( tid' \) can read from \( w \), and is also needed before any barrier that has \( w \) in its “Group A” can be propagated to \( tid' \).
Explanation in $\sim 3$ pages of prose

- Microarchitectural intuitions
- No extraneous concrete details

$\sim 2500$ lines of machine-processed math

- In LEM [ITP’11], a simple new semantic metalanguage
- Can extract executable code, and theorem-prover code
Validating the model

- Extract executable code from definition, exhaustively enumerate possible behaviours of tests
- Run many iterations of tests on real hardware (Power G5, 6, 7)

Excerpt of results:

<table>
<thead>
<tr>
<th>Test</th>
<th>Model</th>
<th>POWER 6</th>
<th>POWER 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRC+sync+addr</td>
<td>Forbid</td>
<td>ok 0 / 16G</td>
<td>ok 0 / 110G</td>
</tr>
<tr>
<td>WRC+data+sync</td>
<td>Allow</td>
<td>ok 150k / 12G</td>
<td>ok 56k / 94G</td>
</tr>
<tr>
<td>PPOCA</td>
<td>Allow</td>
<td>unseen 0 / 39G</td>
<td>ok 62k / 141G</td>
</tr>
<tr>
<td>PPOAA</td>
<td>Forbid</td>
<td>ok 0 / 39G</td>
<td>ok 0 / 157G</td>
</tr>
<tr>
<td>LB</td>
<td>Allow</td>
<td>unseen 0 / 31G</td>
<td>unseen 0 / 176G</td>
</tr>
</tbody>
</table>

- Agreed with key IBM Power designers/architects
Validating the model

- Extract executable code from definition, exhaustively enumerate possible behaviours of tests
- Run many iterations of tests on real hardware (Power G5, 6, 7)

Excerpt of results:

<table>
<thead>
<tr>
<th>Test</th>
<th>Model</th>
<th>POWER 6</th>
<th>POWER 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRC+sync+addr</td>
<td>Forbid</td>
<td>ok 0 / 16G</td>
<td>ok 0 / 110G</td>
</tr>
<tr>
<td>WRC+data+sync</td>
<td>Allow</td>
<td>ok 150k / 12G</td>
<td>ok 56k / 94G</td>
</tr>
<tr>
<td>PPOCA</td>
<td>Allow</td>
<td>unseen 0 / 39G</td>
<td>ok 62k / 141G</td>
</tr>
<tr>
<td>PPOAA</td>
<td>Forbid</td>
<td>ok 0 / 39G</td>
<td>ok 0 / 157G</td>
</tr>
<tr>
<td>LB</td>
<td>Allow</td>
<td>unseen 0 / 31G</td>
<td>unseen 0 / 176G</td>
</tr>
</tbody>
</table>

- Agreed with key IBM Power designers/architects
Validating the model

- Extract executable code from definition, exhaustively enumerate possible behaviours of tests
- Run many iterations of tests on real hardware (Power G5, 6, 7)

Excerpt of results:

<table>
<thead>
<tr>
<th>Test</th>
<th>Model</th>
<th>POWER 6</th>
<th>POWER 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRC+sync+addr</td>
<td>Forbid</td>
<td>ok 0 / 16G</td>
<td>ok 0 / 110G</td>
</tr>
<tr>
<td>WRC+data+sync</td>
<td>Allow</td>
<td>ok 150k / 12G</td>
<td>ok 56k / 94G</td>
</tr>
<tr>
<td>PPOCA</td>
<td>Allow</td>
<td>unseen 0 / 39G</td>
<td>ok 62k / 141G</td>
</tr>
<tr>
<td>PPOAA</td>
<td>Forbid</td>
<td>ok 0 / 39G</td>
<td>ok 0 / 157G</td>
</tr>
<tr>
<td>LB</td>
<td>Allow</td>
<td>unseen 0 / 31G</td>
<td>unseen 0 / 176G</td>
</tr>
</tbody>
</table>

Agreed with key IBM Power designers/architects
Summing up

- A mathematically precise, empirically validated, operational model of POWER
- Microarchitectural intuitions, but abstract: no implementation details

**Rigorous Architecture**

- Can reason about low-level code above it (static analysis tools)
- Can build on for software verification (e.g. compiler verification)
- Can use as specification to test implementations

... Lots to be done!