A Memory Model for RISC-V

Sizhuo Zhang, Muralidaran Vijayaraghavan, Arvind

RISC-V Workshop, November 29, 2016
Why not SC/TSO?

- They both have simple specifications, both axiomatically and operationally
- But simple implementations have low performance
  - Strict ordering requirements for memory instructions
  - To improve performance, one must
    - speculatively execute memory instructions
    - monitor coherence invalidation traffic, and
    - keep checkpoints and rollback on invalidation
Why not POWER/ARM?

- Their operational models expose too much microarchitectural details
  - Branch speculation, OOO execution, etc is exposed in the memory model specification!

- Their axiomatic models are too complex with no well-understood relation to microarchitecture
  - One cannot say with confidence if a particular microarchitectural implementation obeys the model
Properties for a new memory model

- Simple specification without microarchitectural details like Branch speculation
- But well established correspondence to microarchitecture implementations
- Inclusion of sufficient fences to force SC-like behavior when necessary
Our proposal for RISC-V memory model: WMM

Simple operational specification like SC and TSO

- Instantaneous Memory
- Processor
- Processor

SC:
- Stores update memory instantly
- Load reads memory instantly
Our proposal for RISC-V memory model: WMM

Simple operational specification like SC and TSO

TSO:
- Stores are dequeued in order
- When stores are dequeued from store buffer, it updates memory instantly
- Load reads the youngest store from store buffer, or (if not present) memory instantly
Our proposal for RISC-V memory model: WMM

Simple operational specification like SC and TSO

WMM:
- Stores are dequeued in order only for same address
- When stores are dequeued from store buffer, it updates memory instantly and enters every other invalidation buffer instantly
- Load reads the youngest store from store buffer, or (if not present) oldest entry in invalidation buffer, or (if not present) memory instantly
- Oldest invalidation buffer entry can be thrown out any time
Fences in WMM

- Reconcile Fence: Clears Invalidation buffer
- Commit Fence: Flushes Store buffer, i.e. wait till store buffer is empty before executing
Axiomatic Definition of WMM

Preserved program order axiom

\[ X <_{po} Y \land \text{order}(X, Y) \Rightarrow X <_{mo} Y \]

<table>
<thead>
<tr>
<th>order((X, Y))</th>
<th>(Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld (b)</td>
<td>St (b) (v')</td>
</tr>
<tr>
<td>Ld (a)</td>
<td>(a=b)</td>
</tr>
<tr>
<td>St (a) (v)</td>
<td>False</td>
</tr>
<tr>
<td>Reconcile</td>
<td>True</td>
</tr>
<tr>
<td>Commit</td>
<td>False</td>
</tr>
</tbody>
</table>

Load value axiom

\[ Ld \ a \ v \Rightarrow v = \max_{mo}\{v'\mid St \ a \ v' <_{po} Ld \ a \ \vee \ St \ a \ v' <_{mo} Ld \ a\} \]

St-St Fence: Commit
Ld-Ld Fence: Reconcile

St-Ld Fence: Commit + Reconcile
Ld-St Fence: Not needed
Implementing WMM – Processor side

A typical OOO implementation obeys WMM

- A load can execute as early as it wants without being squashed as long as it doesn’t overtake a reconcile or load or store to same address
  - Local checks, no monitoring of coherence invalidations
  - Load address speculation allowed – squashed only if predicted address is wrong

- All instructions are committed in order
  - Stores visible to other threads/cores only after commit
    - Stores cannot overtake loads
  - Prevents “out-of-thin-air” generation of values
Out-of-thin-air issue

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld R1 = x</td>
<td>Ld R2 = y</td>
</tr>
<tr>
<td>St y = R1</td>
<td>St x = 42</td>
</tr>
</tbody>
</table>

Initially \( x = y = R1 = R2 = 0 \)

Finally \( x = y = R1 = R2 = 42 \)

- No processor can produce values out of thin air
  - But incomplete set of axioms seemingly allows this
- Insisting on in-order commits and advertising stores only after commit to other threads/processors takes care of this issue
Implementing WMM – Memory side

- Typical cache-coherent memory with writeback caches attached to typical OOO processors implements WMM.
- If L1 is write-through, it still implements WMM unless the core is SMT.
- SMT cores with L1 write-through caches implement a “non-multicopy-atomic” memory, which is not covered by WMM.

Don’t do it.
## Mapping C++11 to WMM

<table>
<thead>
<tr>
<th>C++11</th>
<th>WMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-atomic Load</td>
<td>Load</td>
</tr>
<tr>
<td>Load Relaxed</td>
<td>Load</td>
</tr>
<tr>
<td>Load Consume</td>
<td>Load; Reconcile</td>
</tr>
<tr>
<td>Load Acquire</td>
<td>Load; Reconcile</td>
</tr>
<tr>
<td>Load SC</td>
<td>Commit; Reconcile; Load; Reconcile</td>
</tr>
<tr>
<td>Non-atomic Store</td>
<td>Store</td>
</tr>
<tr>
<td>Store Relaxed</td>
<td>Store</td>
</tr>
<tr>
<td>Store Release</td>
<td>Commit; Store</td>
</tr>
<tr>
<td>Store SC</td>
<td>Commit; Store</td>
</tr>
</tbody>
</table>

Using operational specification of WMM makes it straightforward.
Conclusion

WMM is a memory model with simple specification and high performant implementation

- Blends well with RISC-V philosophy and should be used as the memory model for RISC-V

Thank you!

szzhang@mit.edu
vmurali@csail.mit.edu
Backup