Background: Basics of our Approach
Microarchitectural Consistency Verification

- Microarch. enforces ISA-level MCM through many small orderings
  - In-order fetch/commit
  - FIFO store buffers
  - Coherence protocol
  - ...
- Difficult to ensure that these orderings *always* enforce the required orderings
- Designs may also be complicated by optimizations (*speculative load reordering*, *early fence retirement*, *OoO execution*), or novel organization (*heterogeneity*)
Does hardware correctly implement ISA MCM?

Microarchitecture


Lds.

L1

L2


Coherence Protocol (SWMR, DVI, etc.)

? = SC/TSO/RISC-V MCM?
Does hardware correctly implement ISA MCM?

SC/TSO/RISC-V MCM? (for the litmus test)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St x ← 1</td>
<td>(i3) Ld r1 ← y</td>
</tr>
<tr>
<td>(i2) St y ← 1</td>
<td>(i4) Ld r2 ← x</td>
</tr>
<tr>
<td>Under TSO: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>
Does hardware correctly implement ISA MCM?

Microarchitecture

Coherence Protocol (SWMR, DVI, etc.)

Instruction level analysis of litmus test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St x ← 1</td>
<td>(i3) Ld r1 ← y</td>
</tr>
<tr>
<td>(i2) St y ← 1</td>
<td>(i4) Ld r2 ← x</td>
</tr>
</tbody>
</table>

Under TSO: Forbid r1=1, r2=0

Observable | Unobservable
-----------|-------------
OK          | OK          
Forbidden   | BUG         
Permitted   | OK          

SC/TSO/RISC-V MCM? (for the litmus test)
Does hardware correctly implement ISA MCM?

Microarchitecture Specification in \textit{\mu}Spec DSL

Axiom "PO\_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \slash \slash \text{ProgramOrder} i1 i2 =>
\quad \text{AddEdge} ((i1, \text{Fetch}), (i2, \text{Fetch}), "PO").

Axiom "Execute\_stage\_is\_in\_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \slash \\slash \text{EdgeExists} ((i1, \text{Fetch}), (i2, \text{Fetch})) =>
\quad \text{AddEdge} ((i1, \text{Execute}), (i2, \text{Execute}), ").

\begin{tabular}{|c|c|}
\hline
\textbf{Core 0} & \textbf{Core 1} \\
\hline
(i1) St [x] \gets 1 & (i3) Ld r1 \gets [y] \\
(i2) St [y] \gets 1 & (i4) Ld r2 \gets [x] \\
\hline
\end{tabular}

Under TSO: Forbid r1=1, r2=0

Litmus Test

Instruction level analysis of litmus test

<table>
<thead>
<tr>
<th></th>
<th>Observable</th>
<th>Unobservable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permitted</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Forbidden</td>
<td>BUG</td>
<td>OK</td>
</tr>
</tbody>
</table>

SC/TSO/RISC-V MCM? (for the litmus test)
Verifying a Single Litmus Test with the Check Suite

Microarchitecture Specification in $\mu$Spec DSL

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ProgramOrder i1 i2 =>
 AddEdge ((i1, Fetch), (i2, Fetch), "PO").

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ /
 EdgeExists ((i1, Fetch), (i2, Fetch)) =>
 AddEdge ((i1, Execute), (i2, Execute), """).

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St x ← 1</td>
<td>(i3) Ld r1 ← y</td>
</tr>
<tr>
<td>(i2) St y ← 1</td>
<td>(i4) Ld r2 ← x</td>
</tr>
</tbody>
</table>

Under TSO: Forbid r1=1, r2=0

Exhaustive enumeration of all possible executions

Microarchitectural happens-before (µhb) graphs
Microarchitectural Consistency Verification with Check

- **Early stage, design-time** verification
- **Key Idea:** Model executions as $\mu$hb graphs
  - **Nodes:** Microarchitectural events or pipeline stages
  - **Edges:** Happens-before relationships between nodes
- **Automatic exhaustive enumeration** of all possible litmus test executions
  - Cyclic Graph $\rightarrow$ Unobservable execution
  - Acyclic Graph $\rightarrow$ Observable execution

<table>
<thead>
<tr>
<th>0 acyclic (Unobservable)</th>
<th>$\geq 1$ acyclic (Observable)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Permitted</strong></td>
<td>OK</td>
</tr>
<tr>
<td><strong>Forbidden</strong></td>
<td>BUG</td>
</tr>
<tr>
<td>OK</td>
<td>OK (Stricter than necessary)</td>
</tr>
</tbody>
</table>
Microarchitectural Consistency Verification with Check

- Early stage, design-time verification
- Key Idea: Model executions as \( \mu hb \) graphs
  - Nodes: Microarchitectural events or pipeline stages
  - Edges: Happens-before relationships between nodes
- **Automatic exhaustive enumeration** of all possible litmus test executions
  - Cyclic Graph \( \rightarrow \) Unobservable execution
  - Acyclic Graph \( \rightarrow \) Observable execution

<table>
<thead>
<tr>
<th></th>
<th>( \geq 1 ) acyclic (Observable)</th>
<th>0 acyclic (Unobservable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permitted</td>
<td>OK</td>
<td>OK (Stricter than necessary)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>BUG</td>
<td>OK</td>
</tr>
</tbody>
</table>
Microarchitectural Consistency Verification with Check

- **Early stage, design-time** verification
- **Key Idea:** Model executions as μhb graphs
  - **Nodes:** Microarchitectural events or pipeline stages
  - **Edges:** Happens-before relationships between nodes

- **Automatic exhaustive enumeration** of all possible litmus test executions
  - Cyclic Graph → Unobservable execution
  - Acyclic Graph → Observable execution

<table>
<thead>
<tr>
<th></th>
<th>≥1 acyclic (Observable)</th>
<th>0 acyclic (Unobservable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permitted</td>
<td>OK</td>
<td>OK (Stricter than necessary)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>BUG</td>
<td>OK</td>
</tr>
</tbody>
</table>
Litmus test-based verification

- Litmus tests: small parallel programs (4-8 instrs)
  - Used to highlight memory model differences/features
  - Typically there is one non-SC outcome of interest (e.g. r1 = 1, r2 = 0 for mp)

- Different litmus tests associated with different ISA models
  - ISA memory model often characterized by their Permitted and Forbidden non-SC litmus test outcomes
  - e.g. TSO litmus test suite, Power litmus test suite, ARM litmus test suite

- Why litmus test-based verification?
  - Focus verification on the scenarios most likely to exhibit bugs, but...
  - ...litmus test-based verification is incomplete (i.e. won’t catch all bugs)
  - PipeProof [Manerkar et al. MICRO 2018] solves this problem! (all-program verification)
## Some Example Litmus Tests

### mp (Message Passing)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [y]</td>
</tr>
<tr>
<td>i2: Store [y] ← 1</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

**SC Forbids:** r1=1, r2=0

### co-mp (mp with one addr)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

**SC Forbids:** r1=2, r2=1, Mem[x] = 2

### sb (Store Buffering)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: Store [y] ← 1</td>
</tr>
<tr>
<td>i2: r1 = Load [y]</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

**SC Forbids:** r1=0, r2=0

### iriw (Independent Reads, Independent Writes)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i2: Store [y] ← 1</td>
<td>i3: r1 = Load [x]</td>
<td>i5: r1 = Load [y]</td>
</tr>
<tr>
<td>i4: r2 = Load [y]</td>
<td>i6: r2 = Load [x]</td>
<td>i4: r2 = Load [y]</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

**SC Forbids:** r1=1, r2=0, r3=1, r4=0
Other things to note

- Check can handle heterogeneous parallelism (not covered today)
- Check can handle microarch. optimizations like speculative execution
- Different flows into and out of tools over the years
  - Originally: uspec DSL => custom solver (written in Gallina)
    - runtimes of seconds/minutes for a single test
  - More recently:
    - input specifications in Alloy (for CheckMate tool)
    - μspec compiled into Z3 formula (in progress)
- **Solver’s search for a satisfying assignment == search for acyclic graph**