MCM Background and Basics for our Approach
Consistency Verification Layers

Our Approach
• Formal specifications -> Happens-before graphs
• Litmus tests for events of interest
• Check Happens-Before Graphs via Efficient SMT solvers
  • Cyclic => A->B->C->A... Can’t happen
  • Acyclic => Scenario is observable
• Fast enough for broad testing
Two Possible Levels of Analysis

- **At the ISA/HLL level** a litmus test outcome can be:
  - Permitted
  - Forbidden

- **Our approach:** At the **hardware level** a litmus test outcome can be:
  - Observable
  - Unobservable

<table>
<thead>
<tr>
<th>What instruction level analysis tells us</th>
<th>What our analysis tells us</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observable</td>
<td>Observable</td>
</tr>
<tr>
<td>Unobservable</td>
<td>Unobservable</td>
</tr>
<tr>
<td>Permitted</td>
<td>OK</td>
</tr>
<tr>
<td>Forbidden</td>
<td>BUG</td>
</tr>
</tbody>
</table>

OK (Stricter than necessary)
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 \textit{always} enforce the required orderings

- Designs may also be complicated by optimizations (\textit{speculative load reordering, early fence retirement, OoO execution}), or novel organization (heterogeneity)
What is the Definition of Correct?

- Preserved Program Order (PPO) for x86-TSO:

✅: Must appear to have executed in order
Prior work: ISA-level happens-before graphs

Recall: Cycle = Outcome Forbidden
No Cycle = Outcome Permitted

Initially: \([x]= [y]=0\)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st ([x])</td>
<td>(i4) st ([y])</td>
</tr>
<tr>
<td>(i2) ld ([x])</td>
<td></td>
</tr>
<tr>
<td>(i3) ld ([y])</td>
<td></td>
</tr>
</tbody>
</table>

Can \(r1=1\), \(r2=0\), \(r3=1\), and \(r4=0\)?

Draw PPO edges:
- x86 TSO requires St-St, Ld-St, Ld-Ld
Prior work: ISA-level happens-before graphs

Recall: Cycle = Outcome Forbidden
No Cycle = Outcome Permitted

Initially: \([x] = [y] = 0\)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st [x] (\leftarrow) 1</td>
<td>(i4) st [y] (\leftarrow) 1</td>
</tr>
<tr>
<td>(i2) ld [x] (\rightarrow) r1</td>
<td>(i5) ld [y] (\rightarrow) r3</td>
</tr>
<tr>
<td>(i3) ld [y] (\rightarrow) r2</td>
<td>(i6) ld [x] (\rightarrow) r4</td>
</tr>
</tbody>
</table>

Can \(r1=1\), \(r2=0\), \(r3=1\), and \(r4=0\)?

Draw “reads-from” edges:
(e.g., if \(r1=1\), (i2) “read from” (i1))
Prior work: ISA-level happens-before graphs

Recall: Cycle = Outcome Forbidden
No Cycle = Outcome Permitted

Initially: \([x] = [y] = 0\)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st ([x]) (\leftarrow 1)</td>
<td>(i4) st ([y]) (\leftarrow 1)</td>
</tr>
<tr>
<td>(i2) ld ([x]) (\rightarrow r1)</td>
<td>(i5) ld ([y]) (\rightarrow r3)</td>
</tr>
<tr>
<td>(i3) ld ([y]) (\rightarrow r2)</td>
<td>(i6) ld ([x]) (\rightarrow r4)</td>
</tr>
</tbody>
</table>

Can \(r = 1, r2 = 0, r3 = 1, \text{ and } r4 = 0\)?

Draw “from-reads” edges:
(e.g., if \(r2 = 0\), \(i3\) must have read its value before \((i4)\) overwrote it)
Prior work: ISA-level happens-before graphs

Recall: Cycle = Outcome Forbidden
No Cycle = Outcome Permitted

Initially: \([x]=[y]=0\)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st [x] (\leftarrow 1)</td>
<td>(i4) st [y] (\leftarrow 1)</td>
</tr>
<tr>
<td>(i2) ld [x] (\rightarrow r1)</td>
<td>(i5) ld [y] (\rightarrow r3)</td>
</tr>
<tr>
<td>(i3) ld [y] (\rightarrow r2)</td>
<td>(i6) ld [x] (\rightarrow r4)</td>
</tr>
</tbody>
</table>

Can \(r1=1\), \(r2=0\), \(r3=1\), and \(r4=0\)?

We have a cycle! But this program outcome is explicitly allowed by TSO...
Why do we need \( \mu \text{hb} \) graphs? – Reason 1

Recall: Cycle = Outcome Forbidden
No Cycle = Outcome Permitted

TSO allows forwarding from a store buffer, so these edges shouldn’t count in our cycle checking:

- **Instruction-level techniques** label edges to solve this problem
- By modeling microarchitectural details (e.g., store buffers) we treat all edges equally
Why do we need μhb graphs? – Reason 2

Furthermore, **instruction level analysis** assumes that the processor correctly implements the memory model.
Our approach: μuhb graphs

Initially: \([x]=[y]=0\)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st [x] ( \leftarrow 1 )</td>
<td>(i4) st [y] ( \leftarrow 1 )</td>
</tr>
<tr>
<td>(i2) ld [x] ( \rightarrow r1 )</td>
<td>(i5) ld [y] ( \rightarrow r3 )</td>
</tr>
<tr>
<td>(i3) ld [y] ( \rightarrow r2 )</td>
<td>(i6) ld [x] ( \rightarrow r4 )</td>
</tr>
</tbody>
</table>

Can \( r1=1, r2=0, r3=1, \) and \( r4=0? \)

No cycle in our uhb graph!
Therefore, observable (which TSO allows).
Litmus test verification

- Litmus tests – small parallel programs
  - Used to highlight memory model differences/features
  - Typically there is one non-SC outcome of interest
    - Outcome = values returned by litmus tests reads

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

- Why litmus test verification?
  - Higher performance when evaluating complex designs
  - Enables us to have a fast, iterative design process
Litmus test verification (for TSO)

Many litmus tests have been developed over the years; they have names e.g., MP, SB
Initial conditions are all 0 unless otherwise stated
This tutorial: we use a sprinkling of established tests

<table>
<thead>
<tr>
<th>MP executions</th>
<th></th>
<th>MP litmus test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message Passing (MP)</strong></td>
<td></td>
<td><strong>Message Passing (MP)</strong></td>
</tr>
<tr>
<td>P0</td>
<td>P1</td>
<td>P0</td>
</tr>
<tr>
<td>W x ← 1</td>
<td>R y → 0</td>
<td>W x ← 1</td>
</tr>
<tr>
<td>W y ← 1</td>
<td>R x → 0</td>
<td>W y ← 1</td>
</tr>
<tr>
<td>SC Outcome Permitted</td>
<td>SC Outcome Permitted</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SB executions</th>
<th></th>
<th>SB litmus test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Store Buffering (SB)</strong></td>
<td></td>
<td><strong>Store Buffering (SB)</strong></td>
</tr>
<tr>
<td>P0</td>
<td>P1</td>
<td>P0</td>
</tr>
<tr>
<td>W x ← 1</td>
<td>W y ← 1</td>
<td>W x ← 1</td>
</tr>
<tr>
<td>R y → 0</td>
<td>R x → 0</td>
<td>R y → 0</td>
</tr>
<tr>
<td>Non-SC Outcome Permitted</td>
<td>SC Outcome Permitted</td>
<td></td>
</tr>
</tbody>
</table>

Non-SC Outcome Forbidden

SC Outcome Permitted

Many litmus tests have been developed over the years; they have names e.g., MP, SB
Initial conditions are all 0 unless otherwise stated
This tutorial: we use a sprinkling of established tests

<table>
<thead>
<tr>
<th>MP executions</th>
<th></th>
<th>MP litmus test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message Passing (MP)</strong></td>
<td></td>
<td><strong>Message Passing (MP)</strong></td>
</tr>
<tr>
<td>P0</td>
<td>P1</td>
<td>P0</td>
</tr>
<tr>
<td>W x ← 1</td>
<td>R y → 1</td>
<td>W x ← 1</td>
</tr>
<tr>
<td>W y ← 1</td>
<td>R x → 0</td>
<td>W y ← 1</td>
</tr>
<tr>
<td>SC Outcome Permitted</td>
<td>SC Outcome Permitted</td>
<td></td>
</tr>
</tbody>
</table>

Non-SC Outcome Forbidden

SC Outcome Permitted

<table>
<thead>
<tr>
<th>SB executions</th>
<th></th>
<th>SB litmus test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Store Buffering (SB)</strong></td>
<td></td>
<td><strong>Store Buffering (SB)</strong></td>
</tr>
<tr>
<td>P0</td>
<td>P1</td>
<td>P0</td>
</tr>
<tr>
<td>W x ← 1</td>
<td>W y ← 1</td>
<td>W x ← 1</td>
</tr>
<tr>
<td>R y → 1</td>
<td>R x → 1</td>
<td>R y → 1</td>
</tr>
<tr>
<td>SC Outcome Permitted</td>
<td>SC Outcome Permitted</td>
<td></td>
</tr>
</tbody>
</table>

Non-SC Outcome Forbidden

SC Outcome Permitted
Check litmus test format (*.test)

Alternative
0 0 0 0 (Write (VA 0 0) (PA 0 0) (Data 1))
1 0 0 0 (Read (VA 1 0) (PA 1 0) (Data 0))
2 1 0 0 (Write (VA 1 0) (PA 1 0) (Data 1))
3 1 0 0 (Read (VA 0 0) (PA 0 0) (Data 0))

Relationship po 0 0 -> 1 0
Relationship po 2 0 -> 3 0
Permitted

- Relationship po ID1 0 -> ID2 0 is required for every po relation in the litmus test
  - Check uses this in the ProgramOrder predicate
  - It must be transitively enumerated
  - Dependency relationships are also possible – see Quick Start Guide for more details

NOTE: In this tutorial, we will be providing litmus tests for you for the most part
Check litmus test format (*.test)

Alternative
0 0 0 0 (Write (VA 0 0) (PA 0 0) (Data 1))
1 0 0 0 (Read (VA 1 0) (PA 1 0) (Data 0))
2 1 0 0 (Write (VA 1 0) (PA 1 0) (Data 1))
3 1 0 0 (Read (VA 0 0) (PA 0 0) (Data 0))

Relationship po 0 0 -> 1 0
Relationship po 2 0 -> 3 0
Permitted

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:W x</td>
<td></td>
<td>2:W y 1</td>
</tr>
<tr>
<td>1:R y</td>
<td>0</td>
<td>3:R x 0</td>
</tr>
</tbody>
</table>

Store Buffering (SB)

Non-SC Outcome Permitted

NOTE: In this tutorial, we will be providing litmus tests for you for the most part.
Check litmus test format (*.test)

Alternative
0 0 0 0 (Write (VA 0 0) (PA 0 0) (Data 1))
1 0 0 0 (Read (VA 1 0) (PA 1 0) (Data 0))
2 1 0 0 (Write (VA 1 0) (PA 1 0) (Data 1))
3 1 0 0 (Read (VA 0 0) (PA 0 0) (Data 0))

Relationship po 0 0 -> 1 0
Relationship po 2 0 -> 3 0
Permitted

<table>
<thead>
<tr>
<th>Store Buffering (SB)</th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:W x ← 1</td>
<td></td>
<td>2:W y ← 1</td>
</tr>
<tr>
<td>1:R y → 0</td>
<td></td>
<td>3:R x → 0</td>
</tr>
</tbody>
</table>

Non-SC Outcome Permitted

NOTE: In this tutorial, we will be providing litmus tests for you for the most part.

- Instruction ID
- Core ID
- Location ID
- Write Values
- Read Values
Check litmus test format (*.test)

Alternative
0 0 0 0 (Write (VA 0 0) (PA 0 0) (Data 1))
1 0 0 0 (Read (VA 1 0) (PA 1 0) (Data 0))
2 1 0 0 (Write (VA 1 0) (PA 1 0) (Data 1))
3 1 0 0 (Read (VA 0 0) (PA 0 0) (Data 0))

Relationship po 0 0 -> 1 0
Relationship po 2 0 -> 3 0
Permitted

<table>
<thead>
<tr>
<th>Store Buffering (SB)</th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: W x ← 1</td>
<td>2: W y ← 1</td>
<td></td>
</tr>
<tr>
<td>1: R y → 0</td>
<td>3: R x → 0</td>
<td></td>
</tr>
</tbody>
</table>

Non-SC Outcome Permitted

---

**Note:** In this tutorial, we will be providing litmus tests for you for the most part.
Check litmus test format (*.test)

Alternative
0 0 0 0 (Write (VA 0 0) (PA 0 0) (Data 1))
1 0 0 0 (Read (VA 1 0) (PA 1 0) (Data 0))
2 1 0 0 (Write (VA 1 0) (PA 1 0) (Data 1))
3 1 0 0 (Read (VA 0 0) (PA 0 0) (Data 0))

Relationship po 0 0 -> 1 0
Relationship po 2 0 -> 3 0
Permitted

<table>
<thead>
<tr>
<th>Store Buffering (SB)</th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:W x ← 1</td>
<td>2:W y ← 1</td>
<td></td>
</tr>
<tr>
<td>1:R y → 0</td>
<td>3:R x → 0</td>
<td></td>
</tr>
</tbody>
</table>

Non-SC Outcome Permitted

Note this is PO NOT PPO

- Instruction ID
- Core ID
- Location ID
- Write Values
- Read Values

NOTE: In this tutorial, we will be providing litmus tests for you for the most part.
Does hardware correctly implement memory model?

Microarchitecture

Coherence Protocol (SWMR, DVI, etc.)

Instruction level analysis

Permitted AND Observable

Forbidden AND Unobservable

Litmus Test

Core 0 | Core 1
---|---
(i1) St x ← 1 & (i3) Ld r1 ← y
(i2) St y ← 1 & (i4) Ld r2 ← x

Under TSO: Forbid r1=1, r2=0

Our analysis

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

Permitted AND Observable
OR
Permitted AND Unobservable
OR
Forbidden AND Unobservable
Check Inputs: Microarchitecture Spec + Litmus Tests

Microarchitecture Specification in \(\mu\text{Spec} \ \text{DSL}\)

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

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

Refer to Quick Start Guide for more information on the \(\mu\text{Spec} \ \text{DSL}\) and how to write axioms.

<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 \(r1=1, r2=0\)
Check Inputs: Microarchitecture Spec + Litmus Tests

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), "PPO").

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St $x \leftarrow 1$</td>
<td>(i3) Ld r1 $\leftarrow y$</td>
</tr>
<tr>
<td>(i2) St $y \leftarrow 1$</td>
<td>(i4) Ld r2 $\leftarrow x$</td>
</tr>
<tr>
<td>Under TSO: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Exhaustive enumeration of all possible executions

Microarchitectural happens-before ($\mu$hb) graphs
Microarchitectural Consistency Verification with Check

- **Key Idea:** Model executions as $\mu$hb graphs
  - **Nodes:** Microarchitectural events or pipeline stages
  - **Edges:** *Local* happens-before relationships between nodes

- **Automatic** model checking of all possible executions through exhaustive enumeration
  - Multiple graphs generated for a single litmus test
  - Cyclic Graph $\rightarrow$ Outcome Not Observable
  - Acyclic Graph $\rightarrow$ Outcome Observable

<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>

![Diagram](https://via.placeholder.com/150)
Other things to note

- Check can handle heterogeneous parallelism
  - For this tutorial, we stick to verifying homogenous parallel configurations
- Check can handle microarchitectural optimizations such as speculative execution
  - More on this (with examples) later
- Our solver is fast – on the order of seconds/minutes