Automated Full-Stack Memory Model Verification with the Check suite

Yatin Manerkar
Princeton University

ARM Cambridge, July 20th, 2018

http://check.cs.princeton.edu/
What are Memory (Consistency) Models?

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].
What are Memory (Consistency) Models?

**Memory Consistency Models (MCMs)**

Specify rules and guarantees about the **ordering and visibility** of accesses to shared memory [Sorin et al., 2011].
What are Memory (Consistency) Models?

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].

C11/C++11
LLVM IR
Java Bytecode
JVM
Cuda
PTX
OpenCL
SPIR
...

x86 CPU
ARM CPU
Power CPU
Nvidia GPU
AMD GPU
...

Shared Virtual Memory
Sequential Consistency (SC) - Interleaving Model

 Defined by [Lamport 1979], execution is the same as if:

(R1) Memory ops of each processor appear in program order

(R2) Memory ops of all processors were executed in some total order

(load reads the value of last store to its address in the total order)

<table>
<thead>
<tr>
<th>Program (mp litmus test) (all addrs initially 0)</th>
<th>Legal Executions</th>
<th>Illegal Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0 x=1 r1=y y=1 r2=x</td>
<td>x=1 r1=y x=1 x=1 r1=y r1=y</td>
<td>r1=1 r2=0</td>
</tr>
<tr>
<td>Core 1 r1=y y=1 r2=x</td>
<td>r1=y r2=x y=1 r2=x y=1 r2=x</td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency (SC) - Interleaving Model

- Defined by [Lamport 1979], execution is the same as if:

  (R1) Memory ops of *each processor* appear in program order

  (R2) Memory ops of *all processors* were executed in some total order

  (load reads the value of last store to its address in the total order)

<table>
<thead>
<tr>
<th>Program (mp litmus test) (all addrs initially 0)</th>
<th>Legal Executions</th>
<th>Illegal Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0 x=1 r1=y y=1 r2=x</td>
<td>x=1 r1=y x=1 r1=y</td>
<td>r1=1 r2=0</td>
</tr>
<tr>
<td>Core 1 r1=y r2=x r1=y r2=x y=1</td>
<td>r1=y x=1 r1=y y=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r2=x y=1 r2=x y=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r1=0 r1=0 r2=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r2=0 r2=0</td>
<td></td>
</tr>
</tbody>
</table>
Hardware Implements Weak Memory Models

- Most processors don’t implement SC
  - x86: Total Store Order (TSO): Relaxes Write->Read ordering
  - ARMv8 and Power relax more orderings

- Compilation to weak memory ISAs must maintain ordering guarantees
  - [Owens et al. TPHOLS 2009], [Batty et al. POPL 2011, POPL 2012], [Wickerson et al. OOPSLA 2015], ...

```
atomic<int> x = 0;
atomic<int> y = 0;

Thread 0
x = 1;
y = 1;

Thread 1
r1 = y;
r2 = x;
```

C11 Forbids: r1 = 1, r2 = 0
Hardware Implements Weak Memory Models

- Most processors don’t implement SC
  - x86: Total Store Order (TSO): Relaxes Write->Read ordering
  - ARMv8 and Power relax more orderings

- Compilation to weak memory ISAs must maintain ordering guarantees
  - [Owens et al. TPHOLS 2009], [Batty et al. POPL 2011, POPL 2012], [Wickerson et al. OOPSLA 2015], ...

```c
atomic<int> x = 0;
atomic<int> y = 0;

Thread 0
x = 1;
y = 1;

Thread 1
r1 = y;
r2 = x;

C11 Forbids: r1 = 1, r2 = 0
```
Hardware Implements Weak Memory Models

- Most processors don’t implement SC
  - x86: Total Store Order (TSO): Relaxes Write->Read ordering
  - ARMv8 and Power relax more orderings

- Compilation to weak memory ISAs must maintain ordering guarantees
  - [Owens et al. TPHOLS 2009], [Batty et al. POPL 2011, POPL 2012], [Wickerson et al. OOPSLA 2015], ...

```c
atomic<int> x = 0;
atomic<int> y = 0;

Thread 0
x = 1;
y = 1;

Thread 1
r1 = y;
r2 = x;

C11 Forbids: r1 = 1, r2 = 0
```

```
Initially, [x] = [y] = 0

Core 0
stl #1, [x]
stl #1, [y]

Core 1
lda r1, [y]
lda r2, [x]

ARMv8 forbids: r1 = 1, r2 = 0
```
Hardware Implements Weak Memory Models

Most processors don’t implement SC

- x86: Total Store Order (TSO): Relaxes Write→Read ordering
- ARMv8 and Power relax more orderings

Compilation to weak memory ISAs must maintain ordering guarantees

- [Owens et al. TPHOLS 2009], [Batty et al. POPL 2011, POPL 2012], [Wickerson et al. OOPSLA 2015], …

```
atomic<int> x = 0;
atomic<int> y = 0;

Thread 0
x = 1;
y = 1;
```

```
Thread 1
r1 = y;
r2 = x;
```

C11 Forbids: r1 = 1, r2 = 0

---

Initially, [x] = [y] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>r1 = y;</td>
</tr>
<tr>
<td>y = 1;</td>
<td>r2 = x;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>stl #1, [x]</td>
<td>lda r1, [y]</td>
</tr>
<tr>
<td>stl #1, [y]</td>
<td>lda r2, [x]</td>
</tr>
</tbody>
</table>

Is the ARMv8 hardware correctly implementing the ARMv8 MCM?
MCM Verification is a Full-Stack Problem!

- Each layer has responsibilities for ensuring correct MCM operation
- Need MCM checking tools at all layers of the computing stack!

[Batty et al. POPL 2011, POPL 2012]
[Alglave et al. TOPLAS 2014]
[Wickerson et al. OOPSLA 2015]

...
MCM Verification is a Full-Stack Problem!

- Each layer has responsibilities for ensuring correct MCM operation
- Need MCM checking tools at all layers of the computing stack!


...
MCM Verification is a Full-Stack Problem!

- Each layer has responsibilities for ensuring correct MCM operation
- Need MCM checking tools at all layers of the computing stack!

[Batty et al. POPL 2011, POPL 2012]
[Alglave et al. TOPLAS 2014]
[Wickerson et al. OOPSLA 2015]

...
MCM Verification is a Full-Stack Problem!

- Each layer has responsibilities for ensuring correct MCM operation
- Need MCM checking tools at all layers of the computing stack!

[Batty et al. POPL 2011, POPL 2012]
[Alglave et al. TOPLAS 2014]
[Wickerson et al. OOPSLA 2015]

...
Suite of tools at various levels of computing stack

Automated Full-Stack MCM checking across litmus test suites
Check Suite: Full-Stack Automated MCM Analysis

- Suite of tools at various levels of computing stack
- **Automated Full-Stack MCM checking** across litmus test suites

PipeCheck & CCICheck
[Lustig et al. MICRO 2014]

TriCheck
[Trippel et al. ASPLOS 2017]

COATCheck
[Lustig et al. ASPLOS 2016]

RTLCheck
[Manerkar et al. MICRO 2017]
Check Suite: Full-Stack Automated MCM Analysis

- Suite of tools at various levels of computing stack
- **Automated Full-Stack MCM checking** across litmus test suites

- **TriCheck**
  [Trippel et al. ASPLOS 2017]

- **COATCheck**
  [Lustig et al. ASPLOS 2016]

- **PipeCheck & CCICheck**
  [Lustig et al. MICRO 2014]
  [Manerkar et al. MICRO 2015]

- **RTLCheck**
  [Manerkar et al. MICRO 2017]

---

Does RTL like Verilog correctly implement microarchitecture?
Check Suite: Full-Stack Automated MCM Analysis

- Suite of tools at various levels of computing stack
- **Automated Full-Stack MCM checking** across litmus test suites
Check Suite: Full-Stack Automated MCM Analysis

- Suite of tools at various levels of computing stack
- **Automated Full-Stack MCM checking** across litmus test suites

So far, tools have found bugs in:
- Widely-used gem5 Research simulator
- Cache coherence paper (TSO-CC)
- IBM XL C++ compiler (fixed in v13.1.5)
- In-design commercial processors
- RISC-V draft ISA specification
- Compiler mapping proofs
- C11 memory model
- Open-source processor RTL
Modelling Microarchitecture: Going below the ISA

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/decode/execute...
  - Orderings enforced by memory hierarchy
  - ...and many more
Modelling Microarchitecture: Going below the ISA

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/decode/execute...
  - Orderings enforced by memory hierarchy
  - ...and many more

Pipeline stages may be FIFO to ensure in-order execution
Modelling Microarchitecture: Going below the ISA

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/decode/execute...
  - Orderings enforced by memory hierarchy
  - ...and many more

Do **individual** orderings correctly work **together** to satisfy consistency model?
Microarchitectural Consistency Checking

Microarchitecture in μspec DSL

Axiom “Decode_is_FIFO”:

... EdgeExists ((i1, Decode), (i2, Decode))
    => AddEdge ((i1, Execute), (i2, Execute)).

Axiom "PO_Fetch":

... SameCore i1 i2 \ ProgramOrder i1 i2 =>
    AddEdge ((i1, Fetch), (i2, Fetch)).

Litmus Test

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

Under SC: Forbid r1=1, r2=0
Microarchitectural Consistency Checking

Microarchitecture in µspec DSL

Axiom “Decode_is_FIFO”:
... EdgeExists ((i1, Decode), (i2, Decode))
    => AddEdge ((i1, Execute), (i2, Execute)).

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

Each axiom specifies an ordering that µarch should respect
Microarchitectural Consistency Checking

Microarchitecture in µspec DSL

Axiom “Decode_is_FIFO”:
... \text{EdgeExists} ((i1, \text{Decode}), (i2, \text{Decode}))
  \Rightarrow \text{AddEdge} ((i1, \text{Execute}), (i2, \text{Execute})).

Axiom "PO_Fetch":
... \text{SameCore} i1 i2 \backslash / \text{ProgramOrder} i1 i2 \Rightarrow
  \text{AddEdge} ((i1, \text{Fetch}), (i2, \text{Fetch})).

Litmus Test

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

Under SC: Forbid r1=1, r2=0
Microarchitectural Consistency Checking

Microarchitecture in µspec DSL

**Axiom “Decode_is_FIFO”:**
... EdgeExists ((i1, Decode), (i2, Decode))
  => AddEdge ((i1, Execute), (i2, Execute)).

**Axiom "PO_Fetch":**
... SameCore i1 i2 \ ProgramOrder i1 i2 =>
  AddEdge ((i1, Fetch), (i2, Fetch)).
Microarchitectural Consistency Checking

Microarchitecture in µspec DSL

**Axiom “Decode_is_FIFO”:**

... EdgeExists ((i1, Decode), (i2, Decode))
   => AddEdge ((i1, Execute), (i2, Execute)).

**Axiom "PO_Fetch":**

... SameCore i1 i2 \ ProgramOrder i1 i2 =>
   AddEdge ((i1, Fetch), (i2, Fetch)).

Litmus Test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
</tbody>
</table>

Microarch. verification checks that combination of axioms satisfies MCM
PipeCheck: Executions as μhb Graphs [Lustig et al. MICRO 2014]

Core 0

Core 1

<p>| Litmus Test mp |</p>
<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
PipeCheck: Executions as µhb Graphs [Lustig et al. MICRO 2014]

Litmus Test $mp$

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

Under TSO: Forbid $r1=1$, $r2=0$
PipeCheck: Executions as $\mu$hb Graphs [Lustig et al. MICRO 2014]
PipeCheck: Executions as µhb Graphs [Lustig et al. MICRO 2014]

Litmus Test mp

<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>
</tbody>
</table>

Under TSO: Forbid r1=1, r2=0
PipeCheck: Executions as µhb Graphs [Lustig et al. MICRO 2014]

Core 0

(i1)
- Fetch
- Dec.
- Exec.
- Mem.
- WB
- SB
- MemHier
- MemHier

(i2)

Core 1

(i1)
- F
- D
- X
- M
- W
- SB
- MemHier
- Compl.

(i2)

Litmus Test mp

<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
PipeCheck: Executions as µhb Graphs [Lustig et al. MICRO 2014]

Core 0

(i1)
Fetch
Dec.
Exec.
Mem.
WB

SB
Mem Hier.

Mem Hier.

(i2)

Fetch
Dec.
Exec.
Mem.
WB

SB
Mem Hier.

Core 1

(i3)
Fetch
Dec.
Exec.
Mem.
WB

SB
MemHier

(i4)

Fetch
Dec.
Exec.
Mem.
WB

(SB

Core 0

(i1) St [x] ← 1
(i2) St [y] ← 1

Core 1

(i3) Ld r1 ← [y]
(i4) Ld r2 ← [x]

Litmus Test mp

Under TSO: Forbid r1=1, r2=0
PipeCheck: Executions as μhb Graphs [Lustig et al. MICRO 2014]
PipeCheck: Executions as µhb Graphs [Lustig et al. MICRO 2014]
PipeCheck: Microarchitectural Correctness

- Cycle in µhb graph => event has to happen before itself (impossible)
- **Cyclic** graph → **unobservable** on µarch
- **Acyclic** graph → **observable** on µarch
- Exhaustively enumerate and check all possible execs of litmus test on µarch
  - Implemented using fast SMT solvers
  - Compare against ISA-level outcome from herd [Alglave et al. TOPLAS 2014]

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

Under SC: Forbid r1=1, r2=0
PipeCheck: Microarchitectural Correctness

- Cycle in μhb graph => event has to happen before itself (impossible)
- **Cyclic** graph → **unobservable** on μarch
- **Acyclic** graph → **observable** on μarch
- Exhaustively enumerate and check all possible execs of litmus test on μarch
  - Implemented using fast SMT solvers
  - Compare against ISA-level outcome from herd [Alglave et al. TOPLAS 2014]

### Litmus Test mp ISA-Level Outcome

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

Under SC: Forbid r1=1, r2=0

<table>
<thead>
<tr>
<th>ISA-Level Outcome</th>
<th>Observable (≥ 1 Graph Acyclic)</th>
<th>Not Observable (All Graphs Cyclic)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allowed</strong></td>
<td>OK</td>
<td>OK (stricter than necessary)</td>
</tr>
<tr>
<td><strong>Forbidden</strong></td>
<td><strong>Consistency violation!</strong></td>
<td>OK</td>
</tr>
</tbody>
</table>
PipeCheck: Microarchitectural Correctness

- Cycle in μhb graph => event has to happen before itself (impossible)
- **Cyclic** graph → **unobservable** on μarch
- **Acyclic** graph → **observable** on μarch
- Exhaustively enumerate and check all possible execs of litmus test on μarch
  - Implemented using fast SMT solvers
  - Compare against ISA-level outcome from **herd** [Alglave et al. TOPLAS 2014]

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISA-Level Outcome</th>
<th>Observable (≥ 1 Graph Acyclic)</th>
<th>Not Observable (All Graphs Cyclic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed</td>
<td>OK</td>
<td>OK (stricter than necessary)</td>
</tr>
<tr>
<td>Forbidden</td>
<td><strong>Consistency violation!</strong></td>
<td>OK</td>
</tr>
</tbody>
</table>
PipeCheck: Microarchitectural Correctness

- Cycle in μhb graph \(\Rightarrow\) event has to happen before itself (impossible)
- **Cyclic** graph \(\rightarrow\) **unobservable** on \(\mu\)arch
- **Acyclic** graph \(\rightarrow\) **observable** on \(\mu\)arch
- Exhaustively enumerate and check all possible execs of litmus test on \(\mu\)arch
  - Implemented using fast SMT solvers
  - Compare against ISA-level outcome from herd [Alglave et al. TOPLAS 2014]

<table>
<thead>
<tr>
<th>ISA-Level Outcome</th>
<th>Observable (≥ 1 Graph Acyclic)</th>
<th>Not Observable (All Graphs Cyclic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed</td>
<td>OK</td>
<td>OK (stricter than necessary)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>Consistency violation!</td>
<td>OK</td>
</tr>
</tbody>
</table>

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

Under SC: Forbid r1=1, r2=0
PipeCheck: Microarchitectural Correctness

- Cycle in μhb graph => event has to happen before itself (impossible)
- **Cyclic** graph → **unobservable** on μarch
- **Acyclic** graph → **observable** on μarch
- Exhaustively enumerate and check all possible execs of litmus test on μarch
  - Implemented using fast SMT solvers
  - Compare against ISA-level outcome from herd [Alglave et al. TOPLAS 2014]

### ISA-Level Outcome | Observable (≥ 1 Graph Acyclic) | Not Observable (All Graphs Cyclic)
--- | --- | ---
Allowed | OK | OK (stricter than necessary)
Forbidden | Consistency violation! | OK

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>
Abstracted memory hierarchy prevents verification of complex coherence issues!
CCICheck: Coherence vs Consistency

- Memory hierarchy is a collection of caches
  - Coherence protocols ensure that all caches agree on the value of any variable

- CCICheck [Manerkar et al. MICRO 2015] shows that consistency verification often cannot simply treat memory hierarchy abstractly
  - Nominated for Best Paper at MICRO 2015
CCICheck: Coherence vs Consistency

- Memory hierarchy is a collection of caches
  - Coherence protocols ensure that all caches agree on the value of any variable

- CCICheck [Manerkar et al. MICRO 2015] shows that consistency verification often cannot simply treat memory hierarchy abstractly
  - Nominated for Best Paper at MICRO 2015
Coherence Protocol Example

- If P1 updates the value of x to 200, the stale value of x in other processors must be **invalidated**
- If P3 wants to subsequently read/write x, it must request the new value
- **SWMR** = Single-Writer Multiple Readers, **DVI** = Data Value Invariant

![Diagram showing processors and caches](attachment:diagram.png)
Coherence Protocol Example

- If P1 updates the value of x to 200, the stale value of x in other processors must be **invalidated**.
- If P3 wants to subsequently read/write x, it must request the new value.
- **SWMR** = Single-Writer Multiple Readers, **DVI** = Data Value Invariant.

![Diagram of Coherence Protocol Example]

- **Processors**
  - P1: St x = 200
  - P2
  - P3

- **Caches**
  - x = 100
  - x = 100
  - x = 100
Coherence Protocol Example

- If P1 updates the value of x to 200, the stale value of x in other processors must be **invalidated**

- If P3 wants to subsequently read/write x, it must request the new value

- **SWMR** = Single-Writer Multiple Readers, **DVI** = Data Value Invariant
Coherence Protocol Example

- If P1 updates the value of x to 200, the stale value of x in other processors must be invalidated.

- If P3 wants to subsequently read/write x, it must request the new value.

- SWMR = Single-Writer Multiple Readers, DVI = Data Value Invariant.
If P1 updates the value of x to 200, the stale value of x in other processors must be **invalidated**.

If P3 wants to subsequently read/write x, it must request the new value.

**SWMR** = Single-Writer Multiple Readers, **DVI** = Data Value Invariant.
Coherence Protocol Example

- If P1 updates the value of x to 200, the stale value of x in other processors must be **invalidated**
- If P3 wants to subsequently read/write x, it must request the new value
- **SWMR** = Single-Writer Multiple Readers, **DVI** = Data Value Invariant

![Diagram of Coherence Protocol Example](image)
Motivating Example – “Peekaboo” [Sorin et al. Primer 2011]

- Three optimizations: correct individually, but not in combination
Motivating Example – “Peekaboo” [Sorin et al. Primer 2011]

- Three optimizations: correct individually, but not in combination
  1. Prefetching
Motivating Example – “Peekaboo” [Sorin et al. Primer 2011]

- Three optimizations: correct individually, but not in combination
  1. Prefetching
  2. Invalidation before use
     • Invalidation can arrive before data
     • Acknowledge Inv early rather than wait for data to arrive
     • But repeated inv before use → livelock [Kubiatowicz et al. ASPLOS 1992]
Motivating Example – “Peekaboo” [Sorin et al. Primer 2011]

- Three optimizations: correct individually, but not in combination
  1. Prefetching
  2. Invalidation before use
     - Invalidation can arrive before data
     - Acknowledge Inv early rather than wait for data to arrive
     - But repeated inv before use → livelock [Kubiatowicz et al. ASPLOS 1992]
  3. **Livelock avoidance**: allow destination core to perform one operation on data when it arrives, **even if already invalidated** [Sorin et al. Primer 2011]
     - Does **not** break coherence
     - Sometimes **intentionally** returns stale data
Motivating Example – “Peekaboo”

- Consider `mp` with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1)</td>
<td>[x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2)</td>
<td>[y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance

Core 0
- x: Shared
- y: Modified
- [x] ← 1
- [y] ← 1

Core 1
- x: Invalid
- y: Invalid
- r1 ← [y]
- r2 ← [x]
Motivating Example – “Peekaboo”

Consider `mp` with the livelock-avoidance mechanism:

**Core 0**

- `x`: Shared
- `y`: Modified

<table>
<thead>
<tr>
<th>Action</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) <code>[x] ← 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i2) <code>[y] ← 1</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Core 1**

- `x`: Invalid
- `y`: Invalid

<table>
<thead>
<tr>
<th>Action</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r1 ← [y]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>r2 ← [x]</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Under SC: Forbid `r1=1, r2=0`

**Optimizations:**

1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1)</td>
<td>[x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2)</td>
<td>[y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under</td>
<td>SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

**Optimizations:**
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider `mp` with the livelock-avoidance mechanism:

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

### Optimizations:

1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1)</td>
<td>[x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2)</td>
<td>[y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance

Core 0

- x: Modified
- y: Modified
- [x] ← 1
- [y] ← 1

Core 1

- x: Invalid
- y: Invalid
- r1 ← [y]
- r2 ← [x]
Motivating Example – “Peekaboo”

- Consider `mp` with the livelock-avoidance mechanism:

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

Under SC: Forbid \(r1=1, r2=0\)

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
Motivating Example – “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Optimizations:
1. Prefetching
2. Invalidation-before-use
3. Livelock avoidance
The Coherence-Consistency Interface (CCI)

- CCI = coherence protocol guarantees to microarch. + orderings microarch. expects from coherence protocol

- SWMR, DVI, No Stale Data + Expected Coherence = Consistency
The Coherence-Consistency Interface (CCI)

- **CCI** = coherence protocol guarantees to microarch. + orderings microarch. expects from coherence protocol

![Diagram showing SWMR, DVI, No Stale Data + Expected Coherence = Consistency]
The Coherence-Consistency Interface (CCI)

- CCI = coherence protocol guarantees to microarch. + orderings microarch. expects from coherence protocol

- SWMR, DVI, No Stale Data + Expected Coherence = Consistency
The Coherence-Consistency Interface (CCI)

- $\text{CCI} = \text{coherence protocol guarantees to microarch.} + \text{orderings microarch. expects from coherence protocol}$

- SWMR, DVI, No Stale Data + Expected Coherence = Consistency
The Coherence-Consistency Interface (CCI)

- CCI = coherence protocol guarantees to microarch. + orderings microarch. expects from coherence protocol

SWMR, DVI, No Stale Data + Expected Coherence = Consistency
The Coherence-Consistency Interface (CCI)

- $\text{CCI} = \text{coherence protocol guarantees to microarch.} + \text{orderings microarch. expects from coherence protocol}

$\text{SWMR, DVI, No Liveloop} + \text{Expected Coherence} = \text{Consistency}$
The Coherence-Consistency Interface (CCI)

- \( CCI = \) coherence protocol guarantees to microarch. + orderings microarch. expects from coherence protocol

SWMR, DVI, No Livlock + Expected Coherence = Consistency
The Coherence-Consistency Interface (CCI)

- **CCI** = coherence protocol guarantees to microarch. + orderings microarch. expects from coherence protocol

- SWMR, DVI, No Liveloack

- Expected Coherence

- CCI Mismatch

- Consistency Violation!
ViCL: Value in Cache Lifetime

- Need a way to model cache occupancy and coherence events for:
  - Coherence protocol optimizations (eg: Peekaboo)
  - Partial incoherence and lazy coherence (GPUs, etc)

- A ViCL is a 4-tuple:
  \[(\text{cache\_id}, \text{address}, \text{data\_value}, \text{generation\_id})\]

- \text{cache\_id} and \text{generation\_id} uniquely identify each cache line

- A ViCL 4-tuple maps on to the period of time over which the cache line serves the data value for the address
ViCLs in µhb Graphs

- ViCLs start at a ViCL Create event and end at a ViCL Expire event
  - Correspond to nodes in µhb graphs
  - Axioms over these nodes and edges enforce coherence and data movement orderings

- Use pipeline model from PipeCheck, but add ViCL nodes and edges

---

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St</td>
<td>x ← 1</td>
</tr>
<tr>
<td>(i2) St</td>
<td>x ← 2</td>
</tr>
<tr>
<td>(i3) Ld</td>
<td>r1 ← x</td>
</tr>
<tr>
<td>(i4) Ld</td>
<td>r2 ← x</td>
</tr>
</tbody>
</table>

Litmus Test `co-mp`
ViCLs in $\mu$hb Graphs

- ViCLs start at a **ViCL Create** event and end at a **ViCL Expire** event
  - Correspond to nodes in $\mu$hb graphs
  - Axioms over these nodes and edges enforce coherence and data movement orderings

- Use pipeline model from PipeCheck, but add ViCL nodes and edges

---

**Litmus Test co-mp**

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St</td>
<td>$x \leftarrow 1$</td>
<td></td>
</tr>
<tr>
<td>(i2) St</td>
<td>$x \leftarrow 2$</td>
<td></td>
</tr>
<tr>
<td>(i3) Ld r1</td>
<td>$\leftarrow x$</td>
<td></td>
</tr>
<tr>
<td>(i4) Ld r2</td>
<td>$\leftarrow x$</td>
<td></td>
</tr>
</tbody>
</table>

In TSO: $r1=2$, $r2=2$ Allowed
ViCLs in $\mu$hb Graphs

- ViCLs start at a ViCL Create event and end at a ViCL Expire event
  - Correspond to nodes in $\mu$hb graphs
  - Axioms over these nodes and edges enforce coherence and data movement orderings
- Use pipeline model from PipeCheck, but add ViCL nodes and edges

Litmus Test $\text{co-mp}$

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St $x \leftarrow 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i2) St $x \leftarrow 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i3) Ld r1 $\leftarrow x$</td>
<td>(i4) Ld r2 $\leftarrow x$</td>
<td></td>
</tr>
</tbody>
</table>

In TSO: r1=2, r2=2 Allowed
ViCLs in $\mu hb$ Graphs

- ViCLs start at a ViCL Create event and end at a ViCL Expire event
  - Correspond to nodes in $\mu hb$ graphs
  - Axioms over these nodes and edges enforce coherence and data movement orderings
- Use pipeline model from PipeCheck, but add ViCL nodes and edges

<table>
<thead>
<tr>
<th>Litmus Test</th>
<th>co-mp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td></td>
</tr>
<tr>
<td>(i1) St</td>
<td>x</td>
</tr>
<tr>
<td>(i2) St</td>
<td>x</td>
</tr>
<tr>
<td>Core 1</td>
<td></td>
</tr>
<tr>
<td>(i3) Ld</td>
<td>r1</td>
</tr>
<tr>
<td>(i4) Ld</td>
<td>r2</td>
</tr>
</tbody>
</table>

In TSO: r1=2, r2=2 Allowed
ViCLs start at a ViCL Create event and end at a ViCL Expire event
- Correspond to nodes in μhb graphs
- Axioms over these nodes and edges enforce coherence and data movement orderings

Use pipeline model from PipeCheck, but add ViCL nodes and edges
µhb Graph for the Peekaboo Problem

- Additional nodes represent ViCL requests and invalidations

**Solution:** Invalidated data only usable if accessing load/store is oldest in program order at time of request [Sorin et al. Primer 2011]

- TSO-CC protocol [Elver and Nagarajan HPCA 2014] was vulnerable to variant of Peekaboo!
  - Now fixed

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

Under SC: Forbid r1=1, r2=0
\[\mu hb\] Graph for the Peekaboo Problem

- Additional nodes represent ViCL requests and invalidations

- **Solution:** Invalidated data only usable if accessing load/store is oldest in program order at time of request [Sorin et al. Primer 2011]

- TSO-CC protocol [Elver and Nagarajan HPCA 2014] was vulnerable to variant of Peekaboo!
  - Now fixed

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) ([x] \leftarrow 1)</td>
<td>(i3) r1 (\leftarrow [y])</td>
</tr>
<tr>
<td>(i2) ([y] \leftarrow 1)</td>
<td>(i4) r2 (\leftarrow [x])</td>
</tr>
</tbody>
</table>

Under SC: Forbid r1=1, r2=0
μhb Graph for the Peekaboo Problem

- Additional nodes represent ViCL requests and invalidations
- **Solution:** Invalidated data only usable if accessing load/store is oldest in program order at time of request [Sorin et al. Primer 2011]
- TSO-CC protocol [Elver and Nagarajan HPCA 2014] was vulnerable to variant of Peekaboo!
  - Now fixed

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

Under SC: Forbid \(r1=1\), \(r2=0\)
μhb Graph for the Peekaboo Problem

- Additional nodes represent ViCL requests and invalidations

- **Solution**: Invalidated data only usable if accessing load/store is oldest in program order at time of request [Sorin et al. Primer 2011]

- TSO-CC protocol [Elver and Nagarajan HPCA 2014] was vulnerable to variant of Peekaboo!
  - Now fixed

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

Under SC: Forbid r1=1, r2=0
**μhb Graph for the Peekaboo Problem**

- Additional nodes represent ViCL requests and invalidations
- **Solution:** Invalidated data only usable if accessing load/store is oldest in program order at time of request [Sorin et al. Primer 2011]
- TSO-CC protocol [Elver and Nagarajan HPCA 2014] was vulnerable to variant of Peekaboo!
  - Now fixed

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

Under SC: Forbid r1=1, r2=0
CCICheck Takeaways

- Coherence & consistency often closely coupled in implementations
- In such cases, coherence & consistency cannot be verified separately

CCICheck: CCI-aware microarchitectural MCM checking
  - Uses ViCL (Value in Cache Lifetime) abstraction
- Discovered bug in TSO-CC lazy coherence protocol
ISA-level MCMs in the Hardware-Software Stack

High-Level Languages (HLLs)

New ISA-level MCM

Hardware
ISA-level MCMs in the Hardware-Software Stack

High-Level Languages (HLLs)

New ISA-level MCM

Which orderings must be guaranteed by hardware?

Hardware
ISA-level MCMs in the Hardware-Software Stack

High-Level Languages (HLLs)

Which orderings does the compiler need to enforce?

New ISA-level MCM

Which orderings must be guaranteed by hardware?

Hardware
TriCheck checks that HLL, compiler, ISA, and hardware align on MCM requirements.
TriCheck: Layers of the Stack are Intertwined

- ISA-level MCMs should allow microarchitectural optimizations but also be compatible with HLLs
- **TriCheck** [Trippel et al. ASPLOS 2017] enables holistic analysis of HLL memory model, ISA-level MCM, compiler mappings, and microarchitectures
  - **Mapping:** translation of HLL synchronization primitives to one or more assembly language instructions
- Also useful for checking HLL compiler mappings to ISA-level MCMs
- Selected as one of 12 “Top Picks of Comp. Arch. Conferences” for 2017
TriCheck: Comparing HLL to Microarchitecture

- HLL Model e.g. C11
- HLL Litmus Test Variants
- HLL to ISA Compiler Mapping
- μspec Microarch. Model

Four Primary Inputs
Examine all C11 `memory_order` combinations (release, acquire, relaxed, seq_cst) for HLL litmus tests.
TriCheck: Comparing HLL to Microarchitecture

- **HLL Model** e.g. C11
- **HLL Litmus Test Variants**
- **HLL to ISA Compiler Mapping**
- **ISA-level litmus tests**
- **μspec Microarch. Model**

**Translate HLL Litmus Tests to ISA-level litmus tests**
TriCheck: Comparing HLL to Microarchitecture

- **HLL Model e.g. C11**
  - HLL Litmus Test Variants
  - Herd [Alglave et al. TOPLAS 2014]
- **HLL to ISA Compiler Mapping**
- **ISA-level litmus tests**
- **μspec Microarch. Model**

Use Herd to check HLL outcomes

HLL Outcome
Forbiden/Allowed?
TriCheck: Comparing HLL to Microarchitecture

- **HLL Model** e.g. C11
- **HLL Litmus Test Variants**
- **Herd** [Alglave et al. TOPLAS 2014]
- **HLL to ISA Compiler Mapping**
- **ISA-level litmus tests**
- **μspec Microarch. Model**

**Use μhb analysis to check microarch. outcomes**

- **HLL Outcome** Forbidden/Allowed?
- **Microarch. Outcome** Observable/Unobservable?
TriCheck: Comparing HLL to Microarchitecture

HLL Model e.g. C11

HLL Litmus Test Variants

Herd [Alglave et al. TOPLAS 2014]

HLL to ISA Compiler Mapping

ISA-level litmus tests

μspec Microarch. Model

Compare HLL and microarch. outcomes

HLL Outcome
Forbidden/Allowed?

Microarch. Outcome
Observable/Unobservable?

μhb Analysis with Check
TriCheck: Comparing HLL to Microarchitecture

HLL Model e.g. C11 → HLL Litmus Test Variants → HLL to ISA Compiler Mapping → ISA-level litmus tests → µspec Microarch. Model

Herd [Alglave et al. TOPLAS 2014] → Compare HLL and microarch. outcomes

HLL Outcome Forbidden → ? → Microarch. Outcome Observable
TriCheck: Comparing HLL to Microarchitecture

HLL Model e.g. C11 → HLL Litmus Test Variants → HLL to ISA Compiler Mapping → ISA-level litmus tests → \( \mu \text{hb} \) Analysis with Check → \( \mu \text{spec} \) Microarch. Model

Herd [Alglave et al. TOPLAS 2014] → Compare HLL and microarch. outcomes

HLL Outcome Forbidden

BUG!

Microarch. Outcome Observable
TriCheck: Comparing HLL to Microarchitecture

HLL Model e.g. C11

HLL Litmus Test Variants

HLL to ISA Compiler Mapping

ISA-level litmus tests

μspec Microarch. Model

Herd [Alglave et al. TOPLAS 2014]

If bugs found, iterate by changing the inputs and re-run

HLL Outcome Forbidden

BUG!

Microarch. Outcome Observable
Using TriCheck for ISA MCM Design: RISC-V

- Ran TriCheck on draft RISC-V ISA MCM with
  - C11 HLL MCM [Batty et al. POPL 2011] [Batty et al. POPL 2016]
  - Compiler mappings based on RISC-V manual
  - Variety of microarchitectures that relaxed various memory orderings
    - All legal according to draft RISC-V spec
    - Ranging from SC microarchitecture to one with reorderings allowed by ARM/Power

- Draft RISC-V MCM for Base ISA incapable of correctly compiling C11:
  - C11 outcome forbidden, but impossible to forbid on hardware
  - RISC-V fences too weak to restore orderings that implementations could relax
Current RISC-V Status

▪ In response to our findings, RISC-V Memory Model Working Group was formed (we are members)
  • Mandate to create an MCM for RISC-V that satisfies community needs

▪ Working Group has developed an MCM proposal that fixes the aforementioned bugs (and other issues)

▪ MCM proposal recently passed the 45-day public feedback period!
  • Well on its way to being included in the next version of the RISC-V ISA spec
TriCheck: Analysing Compiler Mappings

- **HLL Model**: e.g. C11
- **HLL Litmus Test Variants**
- **Herd** [Alglave et al. TOPLAS 2014]
- **HLL to ISA Compiler Mapping**
- **ISA-level litmus tests**
- **μspec Microarch. Model**

- Fix HLL model, microarch model, and ISA-level MCM

- **HLL Outcome** Forbidden/Allowed?
- **Microarch. Outcome** Observable/Unobservable?
TriCheck: Analysing Compiler Mappings

HLL Model e.g. C11

HLL Litmus Test Variants

Herd [Alglave et al. TOPLAS 2014]

HLL Outcome
Forbidden

HLL to ISA Compiler Mapping

ISA-level litmus tests

μhb Analysis with Check

Microarch. Outcome
Observable

μspec Microarch. Model
Checking C11 Mappings to ARMv7/Power

- Ran TriCheck on microarch. with reordering similar to ARMv7/Power
  - Utilised “trailing-sync” compiler mapping [Batty et al. POPL 2012]
  - Discovered 2 cases where C11 outcome forbidden, but allowed by hardware!
  - Deduced that the mapping must be flawed

- Mapping was supposedly proven correct [Batty et al. POPL 2012]
  - Traced the loophole in the proof [Manerkar et al. CoRR’16]

- **Problem: C11 model slightly too strong for mappings**
  - C11 has happens-before ($hb$) ordering and total order on all SC accesses ($sc$)
  - $hb$ and $sc$ orders must agree with each other
  - Trailing-sync mapping does not guarantee this for our counterexamples
Current state of C11

- “Leading-sync” mapping [McKenney and Silvera 2011]
  - Counterexample discovered concurrently to us [Lahav et al. PLDI 2017]

- Both mappings currently broken

- Possible solutions under discussion by C11 memory model committee:
  - RC11 [Lahav et al. PLDI 2017]: remove req. that \(sc\) and \(hb\) orders agree
    - Current mappings work, but reduces intuition in an already complicated C11 model
  - Adding extra fences to mappings
    - low performance, requires recompilation, counterexample pattern not common
TriCheck Takeaways

▪ Both HLL memory models and microarchitectural optimizations influence the design of ISA-level MCMs

▪ **TriCheck** enables holistic analysis of HLL memory model, ISA-level MCM, compiler mappings, and microarchitectural implementations

▪ TriCheck discovered numerous issues with draft RISC-V MCM
  - Influenced the design of the new RISC-V MCM

▪ Discovered two counterexamples to C11 -> ARMv7/Power compiler mappings
  - Mappings were previously “proven” correct; isolated flaw in proof
Memory Consistency Checking for RTL

![Diagram showing microarchitecture checking](image)
Memory Consistency Checking for RTL

How to ensure RTL maintains orderings?

[RTL Image: Christopher Batten]
Memory Consistency Checking for RTL

How to ensure RTL maintains orderings?

RTL implementation

[RTL Image: Christopher Batten]
Memory Consistency Checking for RTL

Microarchitecture Checking

How to ensure RTL maintains orderings?

[RTL Image: Christopher Batten]
**RTLCheck: Checking RTL Implementations**

- **RTLCheck [Manerkar et al. MICRO 2017]** enables checking microarchitectural axioms against an implementation’s Verilog RTL for litmus test suites.
- This helps ensure that the RTL maintains orderings required for consistency.
- Selected as an Honorable Mention from the “*Top Picks of Comp. Arch. Conferences*” for 2017.
RTL Verification is Maturing…

- ...but usually ignores memory consistency!
- Often use SystemVerilog Assertions (SVA)
RTL Verification is Maturing…

- ...but usually ignores memory consistency!
- Often use SystemVerilog Assertions (SVA)

ISA-Formal [Reid et al. CAV 2016]
- Instr. Operational Semantics

No MCM verification
RTL Verification is Maturing…

- ...but usually ignores memory consistency!
- Often use SystemVerilog Assertions (SVA)

ISA-Formal [Reid et al. CAV 2016]
- Instr. Operational Semantics

No MCM verification

DOGReL [Stewart et al. DIFTS 2014]
- Memory subsystem transactions

No multicore MCM verification (?)
RTL Verification is Maturing…

- ...but usually ignores memory consistency!
- Often use SystemVerilog Assertions (SVA)

ISA-Formal [Reid et al. CAV 2016]
- Instr. Operational Semantics
  No MCM verification

DOGReL [Stewart et al. DIFTS 2014]
- Memory subsystem transactions
  No multicore MCM verification (?)

Kami
[Vijayaraghavan et al. CAV 2015] [Choi et al. ICFP 2017]
- MCM correctness for all programs, but...
  Needs Bluespec design and manual proofs!
RTL Verification is Maturing…

- …but usually ignores memory consistency!
- Often use SystemVerilog Assertions (SVA)

Lack of automated memory consistency verification at RTL!

[Vijayaraghavan et al. CAV 2015] [Choi et al. ICFP 2017]
-MCM correctness for all programs, but...

Needs Bluespec design and manual proofs!
RTLCheck: Checking RTL Consistency Orderings

- RTL Design
- Litmus Test
- Axioms
- Mapping Functions

RTLCheck

Temporal SystemVerilog Assertions (SVA)

Cadence JasperGold (RTL Verifier)

Proven?
RTLCheck: Checking RTL Consistency Orderings

- RTL Design
- Litmus Test
- μspec Microarch. Axioms
- Mapping Functions

User-provided mapping functions translate microarch. primitives to RTL equivalents

RTLCheck

Temporal SystemVerilog Assertions (SVA)

Cadence JasperGold (RTL Verifier)

Proven?
RTLCheck: Checking RTL Consistency Orderings

- RTL Design
- Litmus Test
- Microarch. Axioms
- Mapping Functions

RTLCheck automatically translates µarch. ordering axioms to temporal properties

Temporal SystemVerilog Assertions (SVA)

Cadence JasperGold (RTL Verifier)

Proven?
RTLCheck: Checking RTL Consistency Orderings

RTL Design → Litmus Test → μspec Microarch. Axioms → Mapping Functions → RTLCheck

Temporal SystemVerilog Assertions (SVA) → Cadence JasperGold (RTL Verifier)

Properties may be proven or counterexample found

Proven?
Meaning can be Lost in Translation!

小心地滑
Meaning can be Lost in Translation!

小心地滑
(Caution: Slippery Floor)
Meaning can be Lost in Translation!

小心地滑
(Caution: Slippery Floor)

[Image: Barbara Younger]
[Inspiration: Tae Jun Ham]
RTLCheck: Checking Consistency at RTL

Axiomatic Microarch. Analysis
RTLCheck: Checking Consistency at RTL

Axiomatic Microarch. Analysis

Temporal RTL Verification (SVA, etc)
RTLCheck: Checking Consistency at RTL

Axiomatic Microarch. Analysis

Temporal RTL Verification (SVA, etc)

Abstract nodes and happens-before edges
RTLCheck: Checking Consistency at RTL

Axiomatic Microarch. Analysis

Temporal RTL Verification (SVA, etc)

Abstract nodes and happens-before edges

Concrete signals and clock cycles
RTLCheck: Checking Consistency at RTL

Axiomatic Microarch. Analysis

Abstract nodes and happens-before edges

Axiomatic/Temporal Mismatch!

Temporal RTL Verification (SVA, etc)

Concrete signals and clock cycles
Outcome Filtering in Axiomatic Analysis

- **Outcome Filtering**: Restrict test outcome to one particular outcome
  - Allows for more efficient verification

- Axiomatic models make outcome filtering **easy**

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

\( mp \) (Message Passing)
Outcome Filtering in Axiomatic Analysis

- **Outcome Filtering**: Restrict test outcome to one particular outcome
  - Allows for more efficient verification

- Axiomatic models make outcome filtering easy

```plaintext
mp (Message Passing)

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

**Outcome:** r1 = 1, r2 = 1

Execution examined as a whole, so outcome can be enforced!
Outcome Filtering in Axiomatic Analysis

- **Outcome Filtering**: Restrict test outcome to one particular outcome
  - Allows for more efficient verification

- Axiomatic models make outcome filtering *easy*

---

**mp (Message Passing)**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) x = 1;</td>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<td>(i2) y = 1,</td>
<td>(i4) r2 = x;</td>
</tr>
</tbody>
</table>

**Outcome**: r1 = 1, r2 = 1

**Execution examined as a whole**, so outcome can be enforced!
Outcome Filtering in Axiomatic Analysis

- **Outcome Filtering:** Restrict test outcome to one particular outcome
  - Allows for more efficient verification

- Axiomatic models make outcome filtering **easy**

mp (Message Passing)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) ( x = 1 )</td>
<td>(i3) ( r_1 = y )</td>
</tr>
<tr>
<td>(i2) ( y = 1 )</td>
<td>(i4) ( r_2 = x )</td>
</tr>
</tbody>
</table>

**Outcome:** \( r_1 = 1, \ r_2 = 1 \)

**Execution examined as a whole,**
so outcome can be enforced!
Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires **expensive global analysis**
  - Not done by many SVA verifiers, including JasperGold!

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1)</td>
<td>x = 1;</td>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<td>(i2)</td>
<td>y = 1;</td>
<td>(i4) r2 = x;</td>
</tr>
<tr>
<td><strong>Is r1 = 1, r2 = 0 possible?</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires **expensive global analysis**
  - **Not done** by many SVA verifiers, including JasperGold!

```
mp

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

Is r1 = 1, r2 = 0 possible?
```

(i1) x = 1

Step 1
Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires **expensive global analysis**
  - **Not done** by many SVA verifiers, including JasperGold!

```
mp

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

**Is r1 = 1, r2 = 0 possible?**

(i1) x = 1  →  (i2) y = 1  →  (i3) r1 = y = 1  →  (i4) r2 = x = 1

Step 1  →  Step 2  →  Step 3  →  Step 4
Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires **expensive global analysis**
  - Not done by many SVA verifiers, including JasperGold!

\[
\begin{array}{|c|c|}
\hline
\text{Core 0} & \text{Core 1} \\
\hline
(i1) \ x = 1; & (i3) \ r1 = y; \\
(i2) \ y = 1; & (i4) \ r2 = x; \\
\hline
\end{array}
\]

Is \( r1 = 1 \), \( r2 = 0 \) possible?

---

(i1) \( x = 1 \)  (i2) \( y = 1 \)  (i3) \( r1 = y = 1 \)  (i4) \( r2 = x = 0? \)
Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires **expensive global analysis**
  - **Not done** by many SVA verifiers, including JasperGold!

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

Is \(r1 = 1\), \(r2 = 0\) possible?

Need to examine **all possible paths** from current step to end of execution: **too expensive!**
Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires **expensive global analysis**
  - Not done by many SVA verifiers, including JasperGold!

---

**SVA Verifier Approximation:** Only check if constraints hold **up to current step**

Makes Outcome Filtering impossible!

---

(i1) \( x = 1 \)

Step 1

(i2) \( y = 1 \)

Step 2

(i3) \( r_1 = y = 1 \)

Step 3

(i4) \( r_2 = x = 0? \)

Step 4
µspec Analysis Uses Outcome Filtering

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

SC Forbids: r1 = 1, r2 = 0

Axiom "Read\_Values":
Every load either reads **BeforeAllWrites** OR reads **FromLatestWrite**
μspec Analysis Uses Outcome Filtering

<table>
<thead>
<tr>
<th>mp</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(i1) x = 1;</td>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<td></td>
<td>(i2) y = 1;</td>
<td>(i4) r2 = x;</td>
</tr>
<tr>
<td>SC Forbids:</td>
<td>r1 = 1, r2 = 0</td>
<td></td>
</tr>
</tbody>
</table>

Axiom "Read.Values":
Every load either reads **BeforeAllWrites** OR reads **FromLatestWrite**

Note: Axioms abstracted for brevity
### µspec Analysis Uses Outcome Filtering

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) ( x = 1; )</td>
<td>(i3) ( r_1 = y; )</td>
</tr>
<tr>
<td>(i2) ( y = 1; )</td>
<td>(i4) ( r_2 = x; )</td>
</tr>
</tbody>
</table>

SC Forbids: \( r_1 = 1, r_2 = 0 \)

**Axiom "Read\_Values":**
Every load either reads **BeforeAllWrites** OR reads **FromLatestWrite**

**No write for load to read from!**

Note: Axioms abstracted for brevity
μspec Analysis Uses Outcome Filtering

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1)</td>
<td>(x = 1);</td>
<td>((i3) \ r_1 = y;)</td>
</tr>
<tr>
<td>(i2)</td>
<td>(y = 1);</td>
<td>((i4) \ r_2 = x;)</td>
</tr>
<tr>
<td>SC Forbids:</td>
<td>(r_1 = 1, \ r_2 = 0)</td>
<td></td>
</tr>
</tbody>
</table>

Axiom "Read\_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite

Outcome Filtering leads to simpler axioms!

Note: Axioms abstracted for brevity
Temporal Outcome Filtering Fails!

Filtered Read_Values:
Unless Load returns non-zero value,
Load happens before all stores to its address

<table>
<thead>
<tr>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
</tr>
<tr>
<td>Core[0].Commit</td>
</tr>
<tr>
<td>Core[0].SData</td>
</tr>
<tr>
<td>Core[1].Commit</td>
</tr>
<tr>
<td>Core[1].LData</td>
</tr>
</tbody>
</table>

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

SC Forbids: r1 = 1, r2 = 0

Note: Axioms/properties abstracted for brevity
Temporal Outcome Filtering Fails!

Filtered Read Values:
Unless Load returns non-zero value,
Load happens before all stores to its address

<table>
<thead>
<tr>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1)</td>
<td>x = 1;</td>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<td>(i2)</td>
<td>y = 1;</td>
<td>(i4) r2 = x;</td>
</tr>
<tr>
<td>SC Forbids:</td>
<td>r1 = 1, r2 = 0</td>
<td></td>
</tr>
</tbody>
</table>

Note: Axioms/properties abstracted for brevity
Temporal Outcome Filtering Fails!

**Filtered Read.Values:**
Unless **Load** returns non-zero value,
Load happens before all stores to its address

```
(i1) x = 1;
(i2) y = 1;
(i3) r1 = y;
(i4) r2 = x;
```

SC Forbids: r1 = 1, r2 = 0

**Note:** Axioms/properties abstracted for brevity

<table>
<thead>
<tr>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

**After 3 cycles:**
Store happens before load!
Property Violated?
Temporal Outcome Filtering Fails!

Filtered Read_Values:
Unless Load returns non-zero value,
Load happens before all stores to its address

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>clk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Core[0].Commit
Core[0].SData
Core[1].Commit
Core[1].LData

After 3 cycles:
Store happens before load!
Property Violated?

After 6 cycles:
Load does not read 0
No Violation!

Note: Axioms/properties abstracted for brevity
Temporal Outcome Filtering Fails!

Filtered Read_Values:
Unless Load returns non-zero value,
    Load happens before all stores to its address

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>St x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>St y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Ld y</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Ld x</td>
<td></td>
</tr>
</tbody>
</table>

Core[0].Commit
Core[0].SData
Core[1].Commit
Core[1].LData

Note: Axioms/properties abstracted for brevity

After 3 cycles:
Store happens before load!
Property Violated?

After 6 cycles:
Load does not read 0
No Violation!
But SVA verifiers don’t check future cycles!
Temporal Outcome Filtering Fails!

Filtered Read.Values:
Unless Load returns non-zero value, Load happens before all stores to its address.

After 6 cycles:
Load does not read 0
No Violation!
But SVA verifiers don’t check future cycles!

After 3 cycles:
Store happens before load!
Property Violated?

Counterexample flagged despite hardware doing nothing wrong!

Note: Axioms/properties abstracted for brevity
Solution: Load Value Constraints

- Don’t simplify axioms; translate **all** cases
- Tag each case with appropriate **load value constraints**
  - reflect the data constraints required for edge(s)

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

SC Forbids: r1 = 1, r2 = 0

Axiom "Read_Values":
Every load either reads **BeforeAllWrites** OR reads **FromLatestWrite**

Property to check:
mapNode(Ld x → St x, Ld x == 0) or mapNode(St x → Ld x, Ld x == 1);

Note: Axioms and properties abstracted for brevity
Solution: Load Value Constraints

- Don’t simplify axioms; translate all cases
- Tag each case with appropriate load value constraints
  - reflect the data constraints required for edge(s)

Axiom "Read_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite

Property to check:
mapNode(Ld x \rightarrow St x, Ld x \equiv 0) or mapNode(St x \rightarrow Ld x, Ld x \equiv 1);

Note: Axioms and properties abstracted for brevity
Solution: Load Value Constraints

- Don’t simplify axioms; translate all cases
- Tag each case with appropriate load value constraints
  - reflect the data constraints required for edge(s)

Axiom "Read_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite

Property to check:
mapNode(Ld x \rightarrow St x, Ld x == 0) or mapNode(St x \rightarrow Ld x, Ld x == 1);

Note: Axioms and properties abstracted for brevity.
Solution: Load Value Constraints

- Don’t simplify axioms; translate all cases
- Tag each case with appropriate load value constraints
  - reflect the data constraints required for edge(s)

Axiom "Read Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite

Property to check:
mapNode(Ld x → St x, Ld x == 0) OR mapNode(St x → Ld x, Ld x == 1);

Note: Axioms and properties abstracted for brevity
Multi-V-scale: a Multicore Case Study

Core 0

IF

DX

WB

Core 1

IF

DX

WB

Core 2

IF

DX

WB

Core 3

IF

DX

WB

Arbiter

Memory
Multi-V-scale: a Multicore Case Study

- Core 0: IF → DX → WB
- Core 1: IF → DX → WB
- Core 2: IF → DX → WB
- Core 3: IF → DX → WB

Arbiter

Memory

3-stage in-order pipelines
Arbiter enforces that only one core can access memory at any time.
Bug Discovered in V-scale

- V-scale memory internally writes stores to \textit{wdata} register
- \textit{wdata} pushed to memory when subsequent store occurs
- Akin to single-entry store buffer
- When two stores are sent to memory in successive cycles, first of two stores is \textit{dropped} by memory!
- Fixed bug by eliminating \textit{wdata}
- V-scale has since been deprecated by RISC-V Foundation
Bug Discovered in V-scale

- V-scale memory internally writes stores to \texttt{wdata} register
- \texttt{wdata} pushed to memory when subsequent store occurs
- Akin to single-entry store buffer
- When two stores are sent to memory in successive cycles, first of two stores is dropped by memory!
- Fixed bug by eliminating \texttt{wdata}
- V-scale has since been deprecated by RISC-V Foundation
Bug Discovered in V-scale

- V-scale memory internally writes stores to \texttt{wdata} register
- \texttt{wdata} pushed to memory when subsequent store occurs
- Akin to single-entry store buffer
- When two stores are sent to memory in successive cycles, first of two stores is **dropped** by memory!
- Fixed bug by eliminating \texttt{wdata}
- V-scale has since been deprecated by RISC-V Foundation
RTLCheck Takeaways

- Microarchitectural models must be validated against RTL

**RTLCheck:** Automated translation of microarch. axioms into equivalent temporal SVA properties for litmus test suites
  - Translation is complicated by the axiomatic-temporal mismatch
  - JasperGold was able to prove 90% of properties/test in 11 hours runtime

- Last piece of the Check suite; now have tools at all levels of the stack!
The Check suite provides automated full-stack MCM checking of implementations

- Litmus-test based verification to concentrate on error-prone cases

- Can check:
  - Implementation of HLL requirements
  - Virtual memory implementation
  - HLL Compiler mappings
  - Microarchitectural Orderings (including coherence)
  - and even RTL (Verilog)!

- All tools are open-source and publicly available!
With Thanks to…

▪ Collaborators:
  • Margaret Martonosi
  • Daniel Lustig
  • Caroline Trippel
  • Michael Pellauer
  • Aarti Gupta

▪ Funding:
  • Princeton Wallace Memorial Honorific Fellowship
  • STARnet C-FAR (Center for Future Architectures Research)
  • JUMP ADA Center (Applications Driving Architectures)
  • National Science Foundation
Questions?

http://www.cs.princeton.edu/~manerkar


- Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. The 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2017.

Coherence and Consistency

- Most coherence protocols are not that simple!
  - Partial incoherence (e.g. GPUs) [Wickerson et al. OOPSLA 2016]
  - Lazy coherence (e.g. TSO-CC) [Elver and Nagarajan HPCA 2014]

CCI: Coherence-Consistency Interface
Coherence and Consistency

- Most coherence protocols are not that simple!
  - Partial incoherence (e.g. GPUs) [Wickerson et al. OOPSLA 2016]
  - Lazy coherence (e.g. TSO-CC) [Elver and Nagarajan HPCA 2014]

cci: Coherence-Consistency Interface

Coherence and consistency often interwoven
Coherence and Consistency

- Most coherence protocols are not that simple!
  - Partial incoherence (e.g. GPUs) [Wickerson et al. OOPSLA 2016]
  - Lazy coherence (e.g. TSO-CC) [Elver and Nagarajan HPCA 2014]

- **CCI: Coherence-Consistency Interface**

Coherence
- Verifiers can’t ignore consistency implications!

Consistency
- Verifiers can’t assume abstract coherence/memory hierarchy!

Coherence and consistency often interwoven
Coherence and Consistency

- Most coherence protocols are not that simple!
  - Partial incoherence (e.g. GPUs) [Wickerson et al. OOPSLA 2016]
  - Lazy coherence (e.g. TSO-CC) [Elver and Nagarajan HPCA 2014]

**CCI: Coherence-Consistency Interface**
Issue with Draft RISC-V MCM: Cumulativity

Consider this litmus test variant (WRC):

- C11 atomics can specify memory orderings: REL = release, ACQ = acquire

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{St} (x, 1, REL)</td>
<td>r0 = \text{Ld} (x, ACQ)</td>
<td>r1 = \text{Ld} (y, ACQ)</td>
</tr>
<tr>
<td>\text{St} (y, 1, REL)</td>
<td>r2 = \text{Ld} (x, ACQ)</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Forbidden} by C11: r0 = 1, r1 = 1, r2 = 0

RISC-V lacked cumulative fences to enforce this ordering:

- (x5 and x6 contain addresses of x and y)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{sw} x1, (x5)</td>
<td>lw x2, (x5)</td>
<td>lw x3, (x6)</td>
</tr>
<tr>
<td></td>
<td>fence r, rw</td>
<td>fence r, rw</td>
</tr>
<tr>
<td></td>
<td>fence rw, w</td>
<td>lw x4, (x5)</td>
</tr>
<tr>
<td></td>
<td>sw x2, (x6)</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Allowed} by draft RISC-V: x1 = 1, x2 = 1, x3 = 1, x4 = 0
Issue with Draft RISC-V MCM: Cumulativity

Consider this litmus test variant (WRC):

- C11 atomics can specify memory orderings: REL = release, ACQ = acquire

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, REL)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r1 = Ld (y, ACQ)</td>
</tr>
<tr>
<td>St (y, 1, REL)</td>
<td>r2 = Ld (x, ACQ)</td>
<td></td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 1, r2 = 0

- RISC-V lacked cumulative fences to enforce this ordering:
  - (x5 and x6 contain addresses of x and y)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw x1, (x5)</td>
<td>lw x2, (x5)</td>
<td>lw x3, (x6)</td>
</tr>
<tr>
<td>fence r, rw</td>
<td>fence r, rw</td>
<td></td>
</tr>
<tr>
<td>fence rw, w</td>
<td></td>
<td>lw x4, (x5)</td>
</tr>
<tr>
<td>sw x2, (x6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allowed by draft RISC-V: x1 = 1, x2 = 1, x3 = 1, x4 = 0
Issue with Draft RISC-V MCM: Cumulativity

- Consider this litmus test variant (WRC):
  - C11 atomics can specify memory orderings: REL = release, ACQ = acquire
    
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, REL)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r1 = Ld (y, ACQ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>St (y, 1, REL)</td>
<td>r2 = Ld (x, ACQ)</td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 1, r2 = 0

- RISC-V lacked cumulative fences to enforce this ordering:
  - (x5 and x6 contain addresses of x and y)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw x1, (x5)</td>
<td>lw x2, (x5)</td>
<td>lw x3, (x6)</td>
</tr>
<tr>
<td></td>
<td>fence r, rw</td>
<td>fence r, rw</td>
</tr>
<tr>
<td></td>
<td>fence rw, w</td>
<td>lw x4, (x5)</td>
</tr>
<tr>
<td></td>
<td>sw x2, (x6)</td>
<td></td>
</tr>
</tbody>
</table>

Allowed by draft RISC-V: x1 = 1, x2 = 1, x3 = 1, x4 = 0
Issue with Draft RISC-V MCM: Cumulativity

- Consider this litmus test variant (WRC):
  - C11 atomics can specify memory orderings: REL = release, ACQ = acquire
  - RISC-V lacked cumulative fences to enforce this ordering:
    - (x5 and x6 contain addresses of x and y)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, REL)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r1 = Ld (y, ACQ)</td>
</tr>
<tr>
<td></td>
<td>St (y, 1, REL)</td>
<td>r2 = Ld (x, ACQ)</td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 1, r2 = 0

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw x1, (x5)</td>
<td>lw x2, (x5)</td>
<td>lw x3, (x6)</td>
</tr>
<tr>
<td></td>
<td>fence r, rw</td>
<td>fence r, rw</td>
</tr>
<tr>
<td></td>
<td>fence rw, w</td>
<td>lw x4, (x5)</td>
</tr>
<tr>
<td></td>
<td>sw x2, (x6)</td>
<td></td>
</tr>
</tbody>
</table>

Allowed by draft RISC-V: x1 = 1, x2 = 1, x3 = 1, x4 = 0
Issue with Draft RISC-V MCM: Cumulativity

- Consider this litmus test variant (WRC):
  - C11 atomics can specify memory orderings: REL = release, ACQ = acquire

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, REL)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r1 = Ld (y, ACQ)</td>
</tr>
<tr>
<td></td>
<td>St (y, 1, REL)</td>
<td>r2 = Ld (x, ACQ)</td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 1, r2 = 0

- RISC-V lacked cumulative fences to enforce this ordering:
  - (x5 and x6 contain addresses of x and y)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw x1, (x5)</td>
<td>lw x2, (x5)</td>
<td>lw x3, (x6)</td>
</tr>
<tr>
<td>fence r, rw</td>
<td>fence r, rw</td>
<td></td>
</tr>
<tr>
<td>fence rw, w</td>
<td></td>
<td>lw x4, (x5)</td>
</tr>
<tr>
<td>sw x2, (x6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allowed by draft RISC-V: x1 = 1, x2 = 1, x3 = 1, x4 = 0
Issue with Draft RISC-V MCM: Cumulativity

- Consider this litmus test variant (WRC):
  - C11 atomics can specify memory orderings: REL = release, ACQ = acquire

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{St} (x, 1, REL)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r1 = Ld (y, ACQ)</td>
</tr>
<tr>
<td>St (y, 1, REL)</td>
<td>r2 = Ld (x, ACQ)</td>
<td></td>
</tr>
</tbody>
</table>

\textit{Forbidden} by C11: r0 = 1, r1 = 1, r2 = 0

- RISC-V lacked cumulative fences to enforce this ordering:
  - (x5 and x6 contain addresses of x and y)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw x1, (x5)</td>
<td>lw x2, (x5)</td>
<td>lw x3, (x6)</td>
</tr>
<tr>
<td></td>
<td>fence r, rw</td>
<td>fence r, rw</td>
</tr>
<tr>
<td></td>
<td>fence rw, w</td>
<td>lw x4, (x5)</td>
</tr>
<tr>
<td></td>
<td>sw x2, (x6)</td>
<td></td>
</tr>
</tbody>
</table>

\textit{Allowed} by draft RISC-V: x1 = 1, x2 = 1, x3 = 1, x4 = 0
ARMv7/Power Trailing-Sync Counterexample

- Consider this litmus test variant (IRIW):
  - Total order over all SC atomic accesses is required

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, SC)</td>
<td>St (y, 1, SC)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r2 = Ld (y, ACQ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r1 = Ld (y, SC)</td>
<td>r3 = Ld (x, SC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Forbidden</strong> by C11: r0 = 1, r1 = 0, r2 = 1, r3 = 0</td>
</tr>
</tbody>
</table>

- With the trailing-sync mapping, this compiles to the following:
  - Allowed on Power [Sarkar et al. PLDI 2011] and ARMv7 [Alglave et al. TOPLAS 2014]

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>str 1, [x]</td>
<td>str 1, [y]</td>
<td>ldr r1, [x]</td>
<td>ldr r3, [y]</td>
</tr>
<tr>
<td>ctrlisb/ctrlisync</td>
<td>ctrlisb/ctrlisync</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldr r2, [y]</td>
<td>ldr r4, [x]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Allowed</strong> by Power/ARMv7: r1 = 1, r2 = 0, r3 = 1, r4 = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ARMv7/Power Trailing-Sync Counterexample

Consider this litmus test variant (IRIW):

- Total order over all SC atomic accesses is required

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, SC)</td>
<td>St (y, 1, SC)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r2 = Ld (y, ACQ)</td>
</tr>
<tr>
<td>r1 = Ld (y, SC)</td>
<td>r3 = Ld (x, SC)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 0, r2 = 1, r3 = 0

- SC total order must respect happens-before i.e. (sb U sw)+

[Generated with CPPMEM from Cambridge]
ARMv7/Power Trailing-Sync Counterexample

- Consider this litmus test variant (IRIW):
  - Total order over all SC atomic accesses is required

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, SC)</td>
<td>St (y, 1, SC)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r2 = Ld (y, ACQ)</td>
</tr>
<tr>
<td>r1 = Ld (y, SC)</td>
<td>r3 = Ld (x, SC)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 0, r2 = 1, r3 = 0

- SC total order must respect happens-before i.e. (sb U sw)+

[Generated with CPPMEM from Cambridge]
ARMv7/Power Trailing-Sync Counterexample

- Consider this litmus test variant (IRIW):
  - Total order over all SC atomic accesses is required
    - SC total order must respect happens-before i.e. (sb U sw)+

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, SC)</td>
<td>St (y, 1, SC)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r2 = Ld (y, ACQ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r1 = Ld (y, SC)</td>
<td>r3 = Ld (x, SC)</td>
</tr>
</tbody>
</table>

Forbidden by C11: r0 = 1, r1 = 0, r2 = 1, r3 = 0

Generated with CPPMEM from Cambridge
**ARMv7/Power Trailing-Sync Counterexample**

- Consider this litmus test variant (IRIW):
  - Total order over all SC atomic accesses is required

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, SC)</td>
<td>St (y, 1, SC)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r2 = Ld (y, ACQ)</td>
</tr>
<tr>
<td>r1 = Ld (y, SC)</td>
<td>r3 = Ld (x, SC)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Forbidden by C11: r0 = 1, r1 = 0, r2 = 1, r3 = 0*

- SC total order must respect happens-before i.e. (sb U sw)+

[Generated with CPPMEM from Cambridge]
ARMv7/Power Trailing-Sync Counterexample

Consider this litmus test variant (IRIW):

- Total order over all SC atomic accesses is required

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>St (x, 1, SC)</td>
<td>St (y, 1, SC)</td>
<td>r0 = Ld (x, ACQ)</td>
<td>r2 = Ld (y, ACQ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r1 = Ld (y, SC)</td>
<td>r3 = Ld (x, SC)</td>
</tr>
</tbody>
</table>

*Forbidden* by C11: r0 = 1, r1 = 0, r2 = 1, r3 = 0

- SC reads must be before later SC writes

[Generated with CPPMEM from Cambridge]
ARMv7/Power Trailing-Sync Counterexample

- Consider this litmus test variant (IRIW):
  - Total order over all SC atomic accesses is required
    - SC reads must be before later SC writes
  
  **Cycle in the SC order implies outcome is forbidden**
  **But compiled code allows the behaviour!**

[Generated with CPPMEM from Cambridge]
What went wrong?

- It was thought that program order and coherence edges **directly between SC accesses** were all that needed enforcing [Batty et al. POPL 2012]

- But \( hb \) edges can arise between SC accesses through the transitive composition of edges to and from a non-SC **intermediate** access

- Occurs in IRIW counterexample:
What went wrong?

- It was thought that program order and coherence edges directly between SC accesses were all that needed enforcing [Batty et al. POPL 2012]

- But $hb$ edges can arise between SC accesses through the transitive composition of edges to and from a non-SC intermediate access

- Occurs in IRIW counterexample:
What went wrong?

- It was thought that program order and coherence edges directly between SC accesses were all that needed enforcing [Batty et al. POPL 2012]
- But $hb$ edges can arise between SC accesses through the transitive composition of edges to and from a non-SC intermediate access
- Occurs in IRIW counterexample:
Assumption Generation

- Need to restrict executions to those of litmus test

- Three classes of assumptions:
  - Memory initialization
    - Instr. mem and data mem
  - Register initialization
  - Value assumptions
    - **Load value assumptions**: loads return correct value (*when they occur*)
    - **Final value assumptions**: Required final values of memory are respected

- RTLCheck generates **SystemVerilog Assumptions** to constrain executions
  - Utilises user-provided **program mapping function**
Assumption Generation

- **Covering trace**: execution where assumption condition is enforced
  - Eg: execution where load of x returns 0
  - Must obey all assumptions

- **Covering final value assum. == finding forbidden execution!**
  - No covering trace => equivalent to verifying overall test!

- Quicker verification for some tests
  - Expect benefit to be largest for small designs
The Benefits of Final Value Assumptions

- Why generate final value assumptions if test has no final conditions?
- Answer: **Covering traces** can lead to faster verification
- These are traces where assumption condition occurs and can be enforced
The Benefits of Final Value Assumptions

- Why generate final value assumptions if test has no final conditions?
- Answer: **Covering traces** can lead to faster verification
- These are traces where assumption condition occurs and can be enforced

Covering trace for final value assumption is **complete execution** of litmus test
The Benefits of Final Value Assumptions

- Why generate final value assumptions if test has no final conditions?
- Answer: **Covering traces** can lead to faster verification
- These are traces where assumption condition occurs and can be enforced

Covering trace for final val assumption is **complete execution** of litmus test

Covering trace must also obey other assumptions, including **load val assumptions**
(For mp, Ld y = 1 and Ld x = 0)
The Benefits of Final Value Assumptions

- Why generate final value assumptions if test has no final conditions?
- Answer: **Covering traces** can lead to faster verification
- These are traces where assumption condition occurs and can be enforced

Covering trace for final val assumption is **complete execution** of litmus test

Covering trace must also obey other assumptions, including **load val assumptions** (For mp, Ld y = 1 and Ld x = 0)

Thus, covering trace for mp final val assumption (full execution with Ld y=1 and Ld x=0) is **equivalent** to finding **forbidden execution** of mp!

![Diagram of Core[0] and Core[1] states and transitions with corresponding values and labels.](image-url)
Results: Time to Prove Properties

- Two configurations (Hybrid and Full_Proof), avg. runtime 6.2 hrs
  - See paper for configuration details
Results: Time to Prove Properties

- Two configurations (Hybrid and Full_Proof), avg. runtime 6.2 hrs
  - See paper for configuration details

Complete quickly due to covering traces
Results: Time to Prove Properties

- Two configurations (Hybrid and Full_Proof), avg. runtime 6.2 hrs
  - See paper for configuration details

Max runtime 11 hours (if some properties unproven)
**Results: Proven Properties**

- **Full_Proof** generally better (90%/test) than **Hybrid** (81%/test)
- On average, **Full_Proof** can prove more properties in same time
Results: Proven Properties

- **Full_Proof** generally better (90%/test) than **Hybrid** (81%/test)
- On average, **Full_Proof** can prove more properties in same time

**Hybrid** better for only a few tests