RTLCheck: Verifying the Memory Consistency of RTL Designs

Yatin A. Manerkar, Daniel Lustig*, Margaret Martonosi, and Michael Pellauer*

Princeton University

*MICRO-50

http://check.cs.princeton.edu/
Memory Consistency Models (MCMs) are Complex

- MCMs specify ordering requirements of memory operations in parallel programs
  - Essential to correct parallel systems
- Difficult to specify and verify!

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data = 100;</td>
<td>While (Flag != 1) {}</td>
</tr>
<tr>
<td>Flag = 1;</td>
<td>int r1 = Data;</td>
</tr>
</tbody>
</table>

(All locations initially have a value of 0)
Memory Consistency Models (MCMs) are Complex

- MCMs specify ordering requirements of memory operations in parallel programs
  - Essential to correct parallel systems
- Difficult to specify and verify!

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data = 100;</td>
<td>While (Flag != 1) {}</td>
</tr>
<tr>
<td>Flag = 1;</td>
<td>int r1 = Data;</td>
</tr>
</tbody>
</table>

(All locations initially have a value of 0)
Memory Consistency Models (MCMs) are Complex

- MCMs specify ordering requirements of memory operations in parallel programs
  - Essential to correct parallel systems
- Difficult to specify and verify!

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag = 1;</td>
<td>While (Flag != 1) {}</td>
</tr>
<tr>
<td>Data = 100;</td>
<td>int r1 = Data;</td>
</tr>
<tr>
<td>(All locations initially have a value of 0)</td>
<td></td>
</tr>
</tbody>
</table>
How to Verify Hardware MCM Behaviour?

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/WB
  - Coherence protocol orderings
  - ...and many more
How to Verify Hardware MCM Behaviour?

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/WB
  - Coherence protocol orderings
  - ...and many more

![Diagram showing hardware components and coherence protocol](image-url)
How to Verify Hardware MCM Behaviour?

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/WB
  - Coherence protocol orderings
  - ...and many more

FIFO store buffers help ensure Total Store Order (TSO)
How to Verify Hardware MCM Behaviour?

- Hardware enforces consistency model using smaller localized orderings
  - In-order fetch/WB

Do **individual** orderings correctly work **together** to satisfy consistency model?
Our Prior Work: Microarchitectural Consistency Verification

Microarchitecture in μspec DSL

Axiom "StoreBuffer_is_in_order":
... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))
  => AddEdge ((i1, SB_Exit), (i2, SB_Exit)).

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
Our Prior Work: Microarchitectural Consistency Verification

Microarchitecture in µspec DSL

Axiom "StoreBuffer_is_in_order":
... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))
   => AddEdge ((i1, SB_Exit), (i2, SB_Exit)).

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

Each **axiom** specifies an ordering that µarch should respect
Our Prior Work: Microarchitectural Consistency Verification

Microarchitecture in µspec DSL

Axiom “StoreBuffer_is_in_order”:

... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))
   => AddEdge ((i1, SB.Exit), (i2, SB.Exit)).

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
Our Prior Work: Microarchitectural Consistency Verification

Microarchitecture in µspec DSL

Axiom "StoreBuffer_is_in_order":
... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))
    => AddEdge ((i1, SB.Exit), (i2, SB.Exit)).

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 happens-before (μhb) graphs
Our Prior Work: Microarchitectural Consistency Verification

**Microarchitecture in µspec DSL**

**Axiom “StoreBuffer_is_in_order”:**

... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))

=> AddEdge ((i1, SB_Exit), (i2, SB_Exit)).

**Axiom "PO_Fetch":**

... SameCore i1 i2 \ ProgramOrder i1 i2 =>

AddEdge ((i1, Fetch), (i2, Fetch)).

Microarchitecture verification checks that combination of axioms satisfies MCM

[http://check.cs.princeton.edu]
Higher-level verif. requires maintaining ordering axioms

Does RTL maintain microarchitectural orderings?
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!

[Reid et al. CAV 2016] [Stewart et al. DIFTS 2014]
- FormalISA
- Instruction Operational Semantics
- No multicore MCM verification!

[Choi et al. ICFP 2017]
- Memory subsystem transactions
- Needs Bluespec design and manual proofs!
RTLCheck: Verifying Consistency Orderings at RTL

RTL Design  Litmus Test  μspec Microarch. Axioms  Mapping Functions

RTLCheck

Temporal SystemVerilog Assertions (SVA)

JasperGold (RTL Verifier)

Proven?
RTLCheck: Verifying Consistency Orderings at RTL

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

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

RTLCheck

Temporal SystemVerilog Assertions (SVA)

JasperGold (RTL Verifier)

Proven?
RTLCheck: Verifying Consistency Orderings at RTL

RTL Design  
Litmus Test  
μspec Microarch. Axioms  
Mapping Functions  
RTLCheck  
Temporal SystemVerilog Assertions (SVA)  
JasperGold (RTL Verifier)

RTLCheck automatically translates μarch. ordering axioms to temporal properties

Proven?
RTLCheck: Verifying Consistency Orderings at RTL

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

RTLCheck

Temporal SystemVerilog Assertions (SVA)

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)
RTLCheck: Verifying Consistency at RTL

Axiomatic Microarch. Verification
RTLCheck: Verifying Consistency at RTL

Axiomatic Microarch. Verification

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

**Axiomatic Microarch. Verification**

**Temporal RTL Verification (SVA, etc)**

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

**Axiomatic Microarch. Verification**

**Temporal RTL Verification (SVA, etc)**

![Diagram showing the verification process with abstract nodes and edges, as well as concrete signals and clock cycles.]

- Abstract nodes and happens-before edges
- Concrete signals and clock cycles
RTLCheck: Verifying Consistency at RTL

Axiomatic Microarch. Verification

Axiomatic/Temporal Mismatch!

Temporal RTL Verification (SVA, etc)

Abstract nodes and happens-before edges

Concrete signals and clock cycles
Instances of the Axiomatic/Temporal Mismatch

- Outcome Filtering: enforcing particular outcome for litmus test
  - Discussed next

- Mapping Individual Happens-Before Edges (*detailed in paper*)

- Filtering Match Attempts (*detailed in paper*)
**Outcome Filtering in Axiomatic Verification**

- Axiomatic models make outcome filtering **easy and efficient**

---

**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 Filtering in Axiomatic Verification

- Axiomatic models make outcome filtering **easy and efficient**

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

**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 Verification

- Axiomatic models make outcome filtering **easy and efficient**

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 Verification

- Axiomatic models make outcome filtering **easy and efficient**

mp (Message Passing)

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

<table>
<thead>
<tr>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<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 Temporal Verification

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

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

Is r1 = 1, r2 = 0 possible?
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!

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

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

---

(i1) x = 1
Step 1

(i2) y = 1
Step 2

(i3) r1 = y = 1
Step 3

(i4) r2 = x = 1
Step 4
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>
<tr>
<td><strong>Is r1 = 1, r2 = 0 possible?</strong></td>
<td></td>
</tr>
</tbody>
</table>

(i1) x = 1 
Step 1

(i2) y = 1 
Step 2

(i3) r1 = y = 1 
Step 3

(i4) r2 = x = 1 
Step 4

(i4) r2 = x = 0?

mp
Outcome Filtering in Temporal Verification

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

$$\begin{align*}
(i1) \ x = 1; \\
(i2) \ y = 1; \\
(i3) \ r1 = y; \\
(i4) \ r2 = x;
\end{align*}$$

Core 0 | Core 1
---|---
(i1) x = 1; | (i3) r1 = y; 
(i2) y = 1; | (i4) r2 = x; 
**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!
µspec Verification 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) r1 = y;</td>
</tr>
<tr>
<td>(i2)</td>
<td>y = 1;</td>
<td>(i4) r2 = x;</td>
</tr>
<tr>
<td><strong>SC Forbids:</strong></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 Verification Uses Outcome Filtering

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

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

Note: Axioms abstracted for brevity
**microspec Verification 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>
<tr>
<td><strong>SC Forbids:</strong> $r_1 = 1$,</td>
<td></td>
</tr>
<tr>
<td>$r_2 = 0$</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Note: Axioms abstracted for brevity
**μspec Verification 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><strong>SC Forbids:</strong></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!

**BeforeAllWrites:**

**Unless** Load returns non-zero value,

Load happens before all stores to its address

<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!

**BeforeAllWrites:**

Unless **Load** returns non-zero value,

Load happens before all stores to its address

<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

After 3 cycles:

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

**BeforeAllWrites:**

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

<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: \( r_1 = 1 \), \( r_2 = 0 \)

**After 3 cycles:**

Store happens before load! **Property Violated?**

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

**BeforeAllWrites:**

Unless **Load** returns non-zero value,
Load happens before all stores to its address

**Core[0].Commit**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Core[0].SData**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x1</td>
<td>0x1</td>
</tr>
</tbody>
</table>

**Core[1].Commit**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Core[1].LData**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x0</td>
<td>0x0</td>
</tr>
</tbody>
</table>

**mp**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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

**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!

**BeforeAllWrites:**

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

<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

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

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

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

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

<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>SC Forbids: r1 = 1, r2 = 0</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Countermexample 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)

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 → 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 → 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 $\text{BeforeAllWrites}$ or reads $\text{FromLatestWrite}$

<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: $r1 = 1$, $r2 = 0$

Property to check:
$\text{mapNode}(\text{Ld} x \rightarrow \text{St} x, \text{Ld} x == 0) \text{ or } \text{mapNode}(\text{St} x \rightarrow \text{Ld} x, \text{Ld} x == 1)$

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

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 \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 wdata register
- 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 wdata
- V-scale has since been deprecated by RISC-V Foundation
Bug Discovered in V-scale

- V-scale memory internally writes stores to `wdata` register
- `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 `wdata`
- V-scale has since been deprecated by RISC-V Foundation
Results: Time to Verification

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

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

Verified very quickly through covering traces (details in paper)
Results: Time to Verification

- 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
MCM Verification: The Big Picture

High-Level Languages (HLL)
- [Batty et al. POPL 2012]
- [TriCheck, ASPLOS 2017]

Compiler
- [COATCheck, ASPLOS 2016]
- [Vafeiadis et al. PLDI 2017]

Architecture
- [Sarkar et al. PLDI 2011]
- [Alglave et al. TOPLAS 2014]

Microarchitecture
- [PipeCheck, MICRO-47]
- [CCICheck, MICRO-48]

Processor RTL
- [Vijayaraghavan et al. CAV 2015]
- [Choi et al. ICFP 2017]
MCM Verification: The Big Picture

High-Level Languages (HLL)
[Batty et al. POPL 2012] [TriCheck, ASPLOS 2017]

Compiler
[COATCheck, ASPLOS 2016] [Vafeiadis et al. PLDI 2017]

Architecture
[Sarkar et al. PLDI 2011] [Alglave et al. TOPLAS 2014]

Microarchitecture
[PipeCheck, MICRO-47] [CCICheck, MICRO-48]

Processor RTL

Higher-level tools directly or indirectly assume correctness of underlying RTL!
MCM Verification: The Big Picture

High-Level Languages (HLL)
- [Batty et al. POPL 2012]
- [TriCheck, ASPLOS 2017]

Compiler
- [COATCheck, ASPLOS 2016]
- [Vafeiadis et al. PLDI 2017]

Architecture
- [Sarkar et al. PLDI 2011]
- [Alglave et al. TOPLAS 2014]

Microarchitecture
- [PipeCheck, MICRO-47]
- [CCICheck, MICRO-48]

Processor RTL

Higher-level tools directly or indirectly assume correctness of underlying RTL!

Requires Bluespec design and manual proof

[Batty et al. POPL 2012]
[Trichckeck, ASPLOS 2017]

[COATCheck, ASPLOS 2016]
[Vafeiadis et al. PLDI 2017]

[Sarkar et al. PLDI 2011]
[Alglave et al. TOPLAS 2014]

[PipeCheck, MICRO-47]
[CCICheck, MICRO-48]

[Vijayaraghavan et al. CAV 2015]
[Choi et al. ICFP 2017]
MCM Verification: The Big Picture

- **High-Level Languages (HLL)**
  - [Batty et al. POPL 2012] [TriCheck, ASPLOS 2017]

- **Compiler**
  - [COATCheck, ASPLOS 2016] [Vafeiadis et al. PLDI 2017]

- **OS**
  - [Sarkar et al. PLDI 2011] [Alglave et al. TOPLAS 2014]

- **Architecture**
  - [PipeCheck, MICRO-47] [CCICheck, MICRO-48]

- **Microarchitecture**
  - [RTLCheck, MICRO-50]
    - [Vijayaraghavan et al. CAV 2015]
    - [Choi et al. ICFP 2017]

- **Processor RTL**

- Higher-level tools **directly or indirectly assume** correctness of underlying RTL!

- **RTLCheck validates** RTL against µarch, filling µarch-RTL verification gap!
- **Automated** MCM verification of arbitrary RTL for suite of litmus tests
Conclusions

- **RTLCheck**: Automated MCM Verification of *arbitrary* RTL against *arbitrary* microarchitectural orderings
  - Translates microarch. axioms into equivalent temporal SVA properties
  - Allows RTL to be validated against µarch ordering specification
  - Capable of handling arbitrary ISA-level MCMs (SC, TSO, ARM, Power,...)
  - Most of generated properties proven by JasperGold in **minutes or hours**

- RTLCheck enables **full-stack HLL-to-RTL** MCM verification (with rest of Check suite) across a collection of litmus tests

Code available at [https://github.com/ymanerka/rtlcheck](https://github.com/ymanerka/rtlcheck)
RTLCheck: Verifying the Memory Consistency of RTL Designs

Yatin A. Manerkar, Daniel Lustig*, Margaret Martonosi, and Michael Pellauer*

Code available at
https://github.com/ymanerka/rtlcheck

http://check.cs.princeton.edu/