RTLCheck: Verifying the Memory Consistency of RTL Designs

Yatin A. Manerkar, Daniel Lustig*,

Margaret Martonosi, and Michael Pellauer*

Princeton University

*NVIDIA

MICRO-50

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

Memory Consistency Models (MCMs) are Complex

Core 0	Core 1
Data = 100;	While (Flag != 1) {}
Flag = 1;	int r1 = Data;
(All locations initially have a value of 0)	

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

Memory Consistency Models (MCMs) are Complex

Core 0	Core 1
Data = 100;	While (Flag != 1) {}
Flag = 1;	int r1 = Data;
(All locations initially have a value of 0)	

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

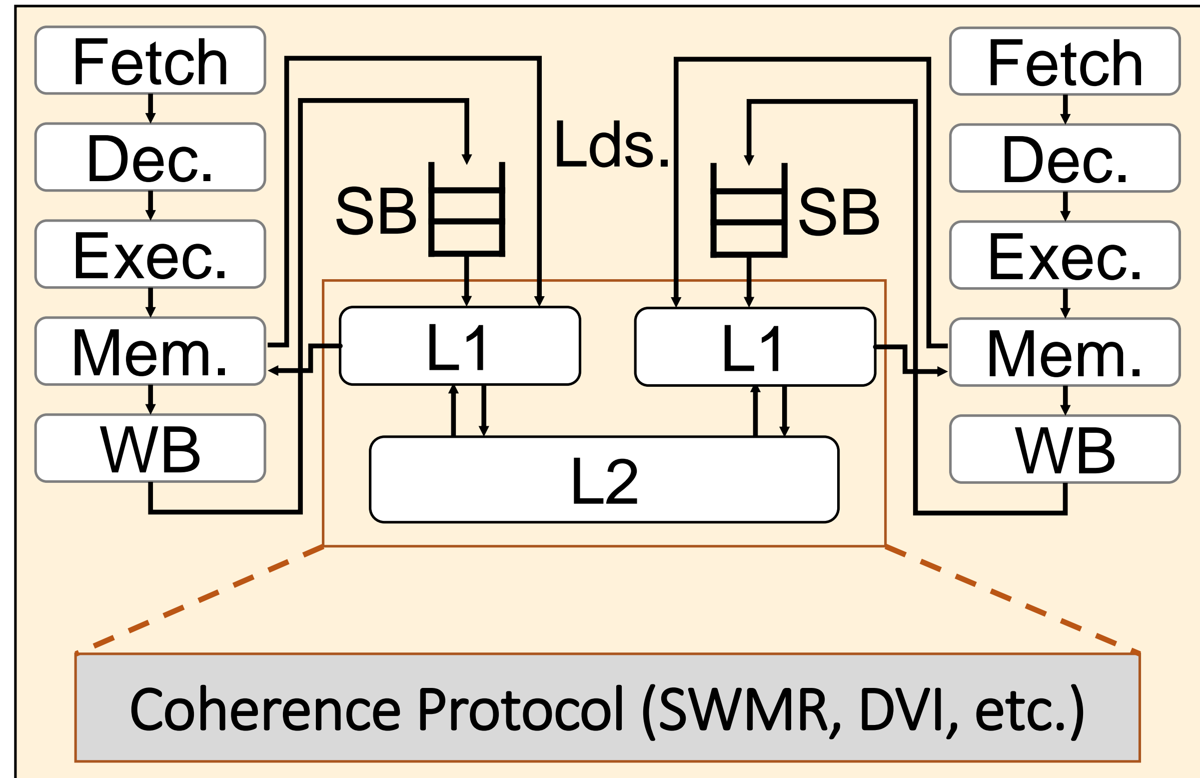
Memory Consistency Models (MCMs) are Complex

Core 0	Core 1
Flag = 1;	While (Flag != 1) {}
Data = 100;	int r1 = Data;
(All locations initially have a value of 0)	

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

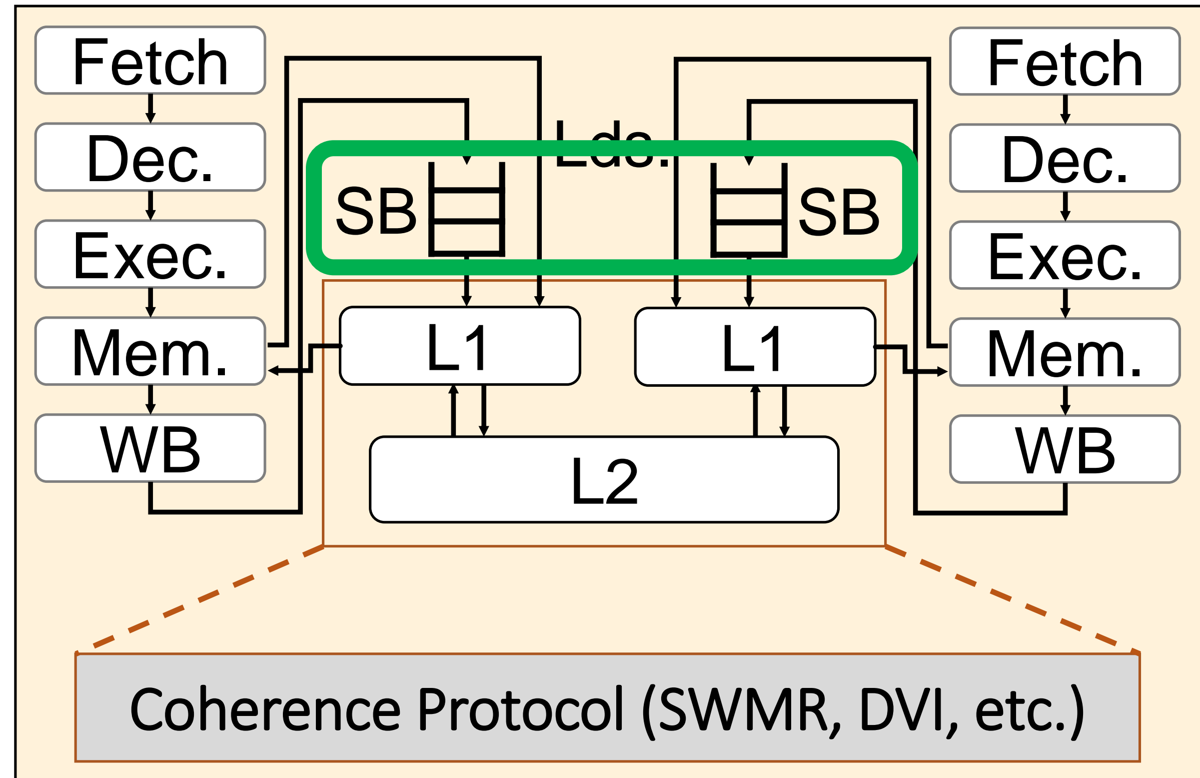
How to Verify Hardware MCM Behaviour?

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



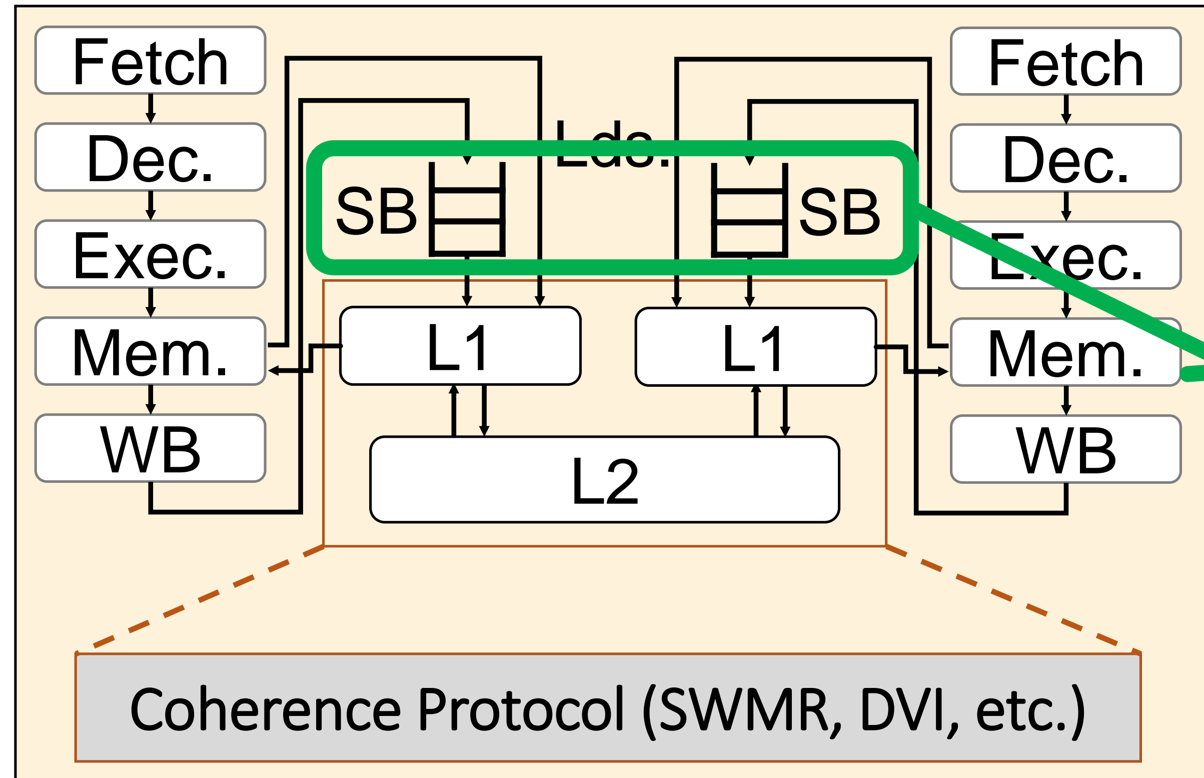
How to Verify Hardware MCM Behaviour?

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



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

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
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

(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
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

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

Our Prior Work: Microarchitectural Consistency Verification

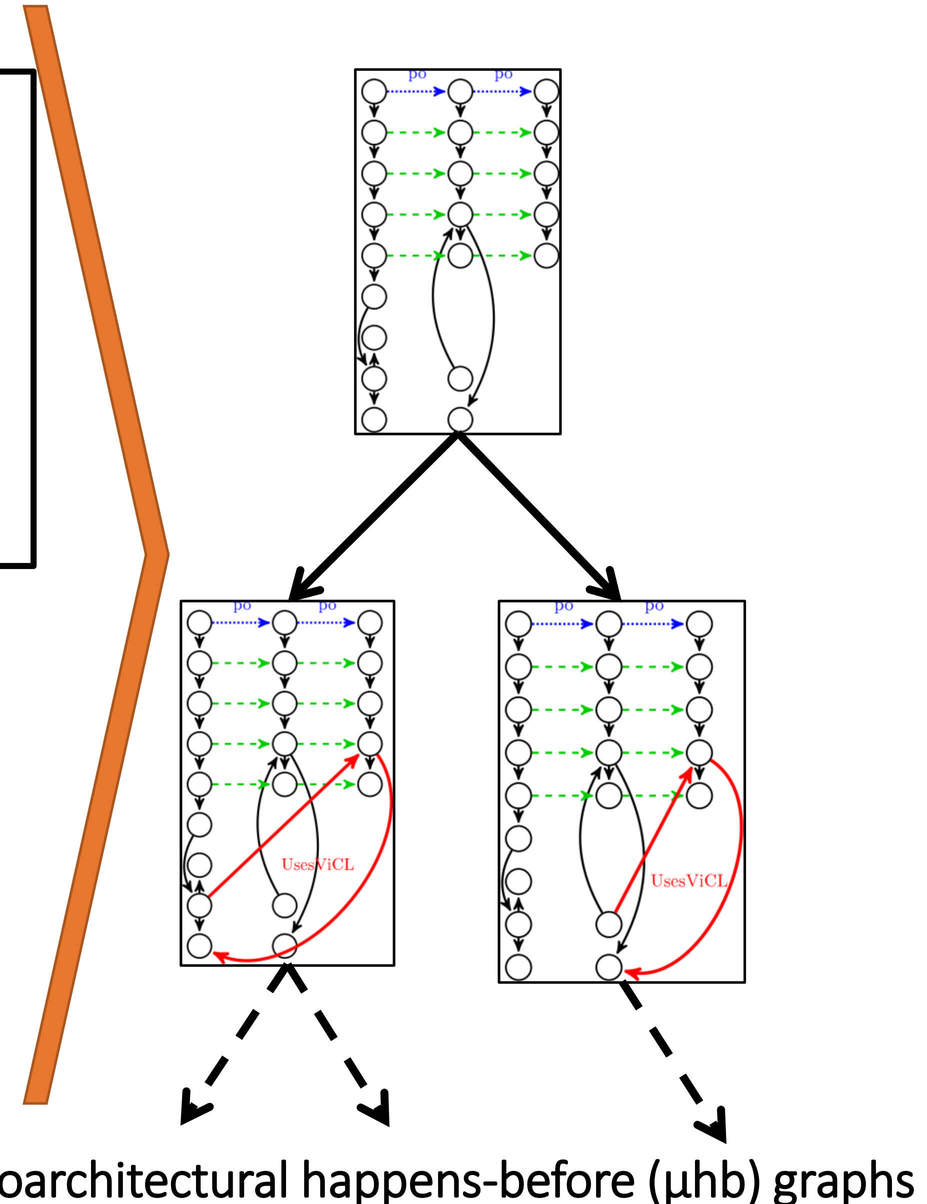
Microarchitecture in μspec DSL

```
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

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	



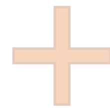
Our Prior Work: Microarchitectural Consistency Verification

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

Microarchitecture in μspec DSL

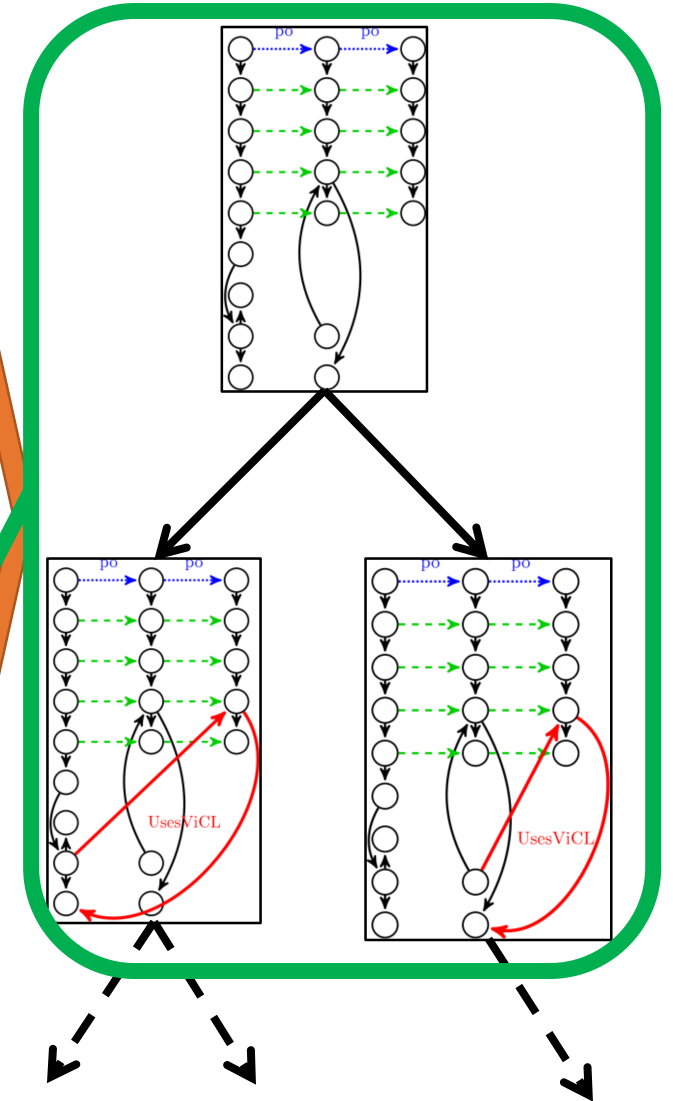
```
Axiom "StoreBuffer_is_in_order":
... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))
     => AddEdge ((i1, SB_Exit), (i2, SB_Exit)).

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



Litmus Test

Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]



Microarch. verification checks that **combination** of axioms satisfies MCM

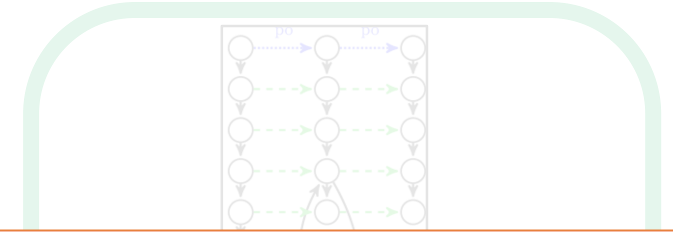
rchitectural happens-before (μhb) graphs

Our Prior Work: Microarchitectural Consistency Verification

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

Microarchitecture in μspec DSL

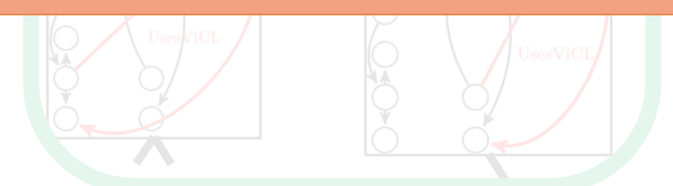
```
Axiom "StoreBuffer_is_in_order":
... EdgeExists ((i1, SB_Enter), (i2, SB_Enter))
      => AddEdge ((i1, SB_Exit), (i2, SB_Exit)).
```



Higher-level verif. requires maintaining ordering axioms

Does RTL maintain microarchitectural orderings?

Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]



Microarch. verification checks that **combination** of axioms satisfies MCM

…rchitectural happens-before (μhb) graphs

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: 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)

Proven?

RTLCheck automatically translates µarch. ordering **axioms** to **temporal** properties

RTLCheck: Verifying Consistency Orderings at RTL

RTL Design

Litmus Test

µspec Microarch. Axioms

Mapping Functions

RTLCheck

Temporal SystemVerilog Assertions (SVA)

JasperGold (RTL Verifier)

Proven?

Properties may be **proven** or **counterexample** found

Meaning can be Lost in Translation!

小心地滑

Meaning can be Lost in Translation!

小心地滑

(Caution: Slippery Floor)

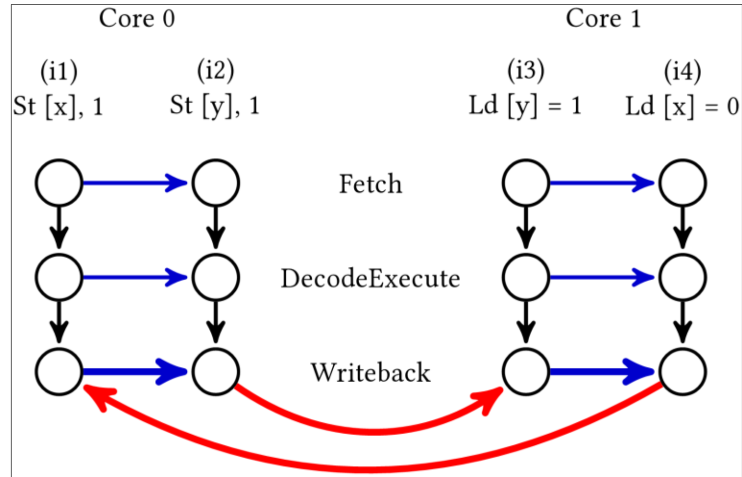
Meaning can be Lost in Translation!

小心地滑
(Caution: Slippery Floor)

[Image: Barbara Younger]
[Inspiration: Tae Jun Ham]

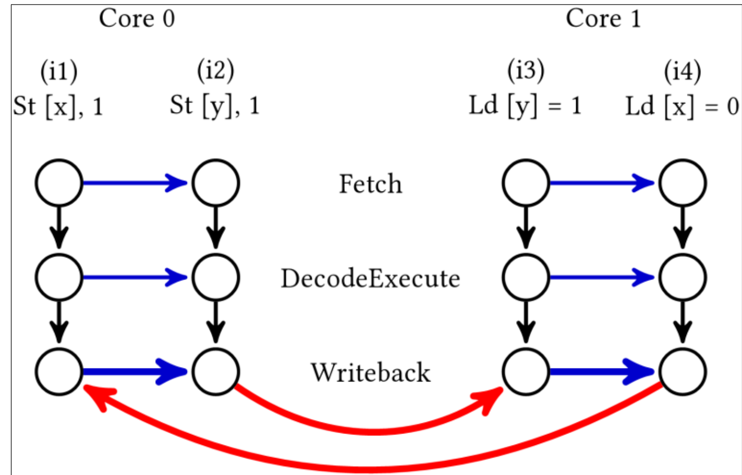
RTLCheck: Verifying Consistency at RTL

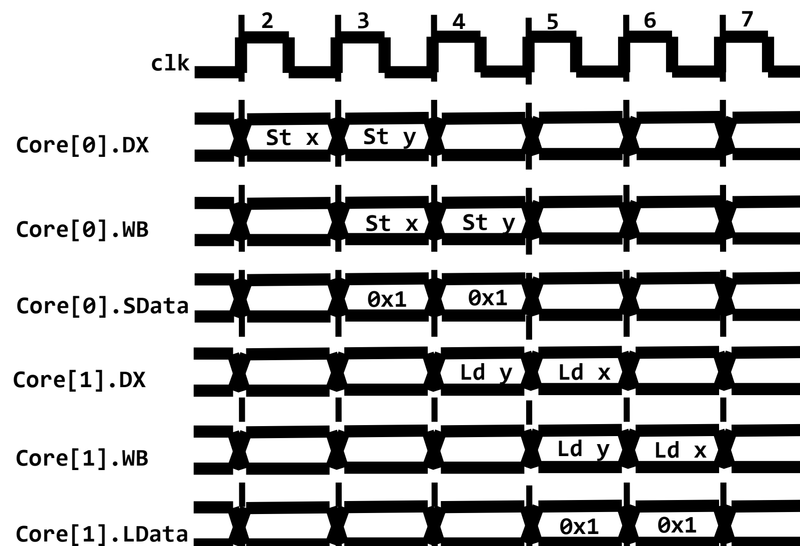
Axiomatic
Microarch.
Verification



RTLCheck: Verifying Consistency at RTL

Axiomatic
Microarch.
Verification

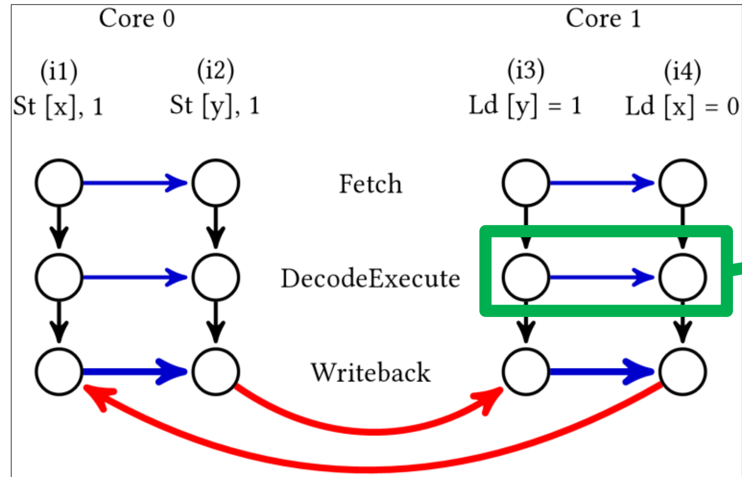




Temporal
RTL Verification
(SVA, etc)

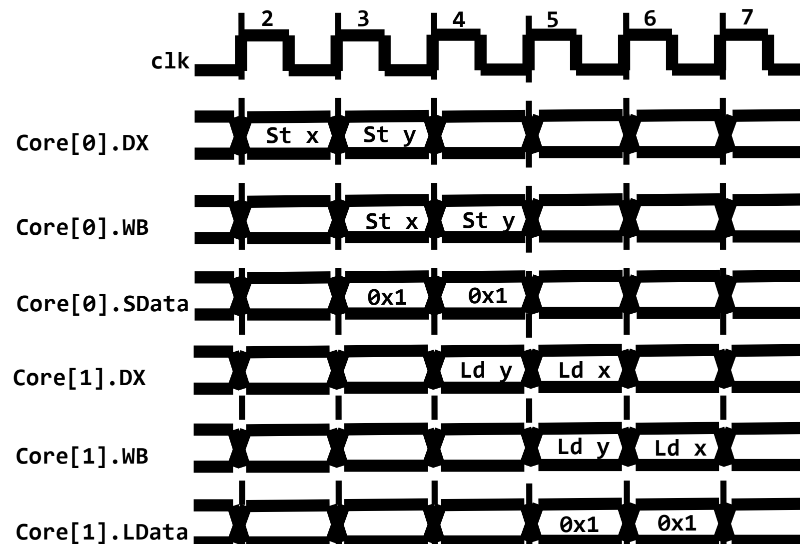
RTLCheck: Verifying Consistency at RTL

Axiomatic
Microarch.
Verification



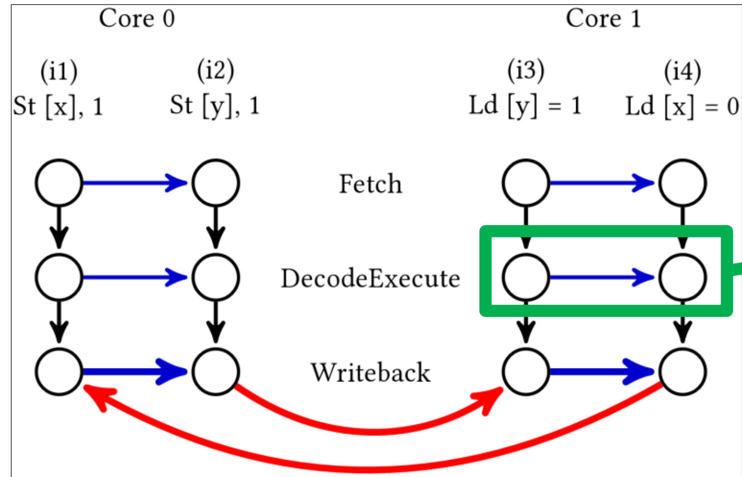
Abstract nodes and happens-before edges

Temporal
RTL Verification
(SVA, etc)



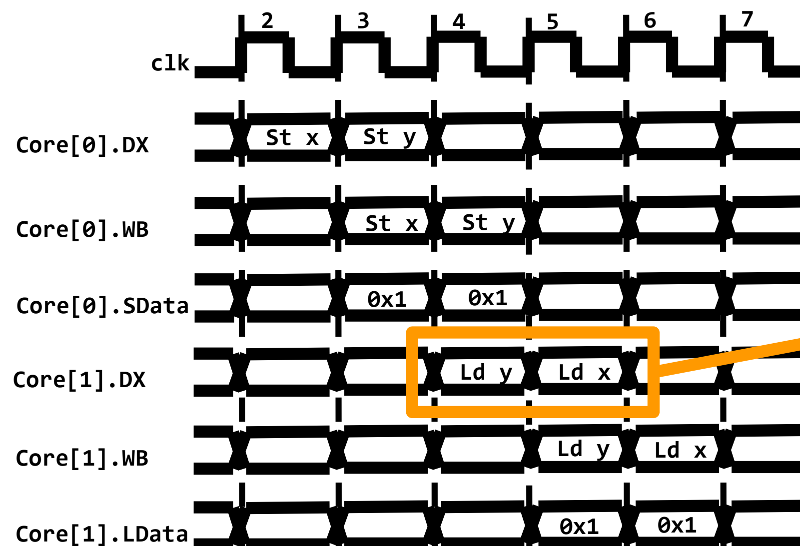
RTLCheck: Verifying Consistency at RTL

Axiomatic Microarch. Verification



Abstract nodes and happens-before edges

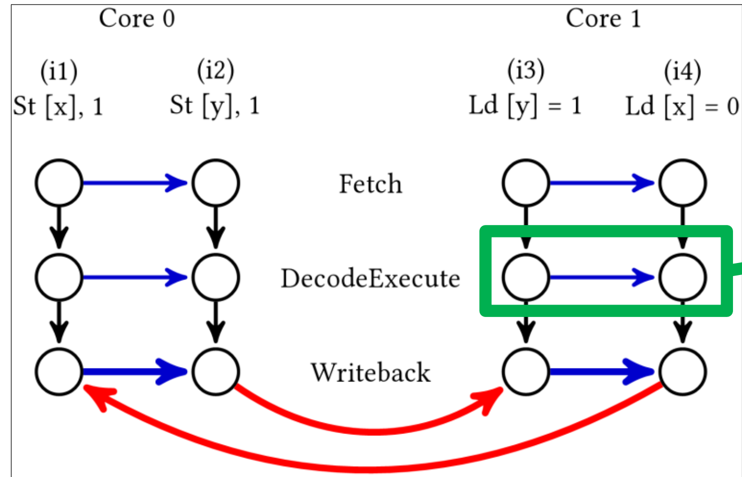
Temporal RTL Verification (SVA, etc)



Concrete signals and clock cycles

RTLCheck: Verifying Consistency at RTL

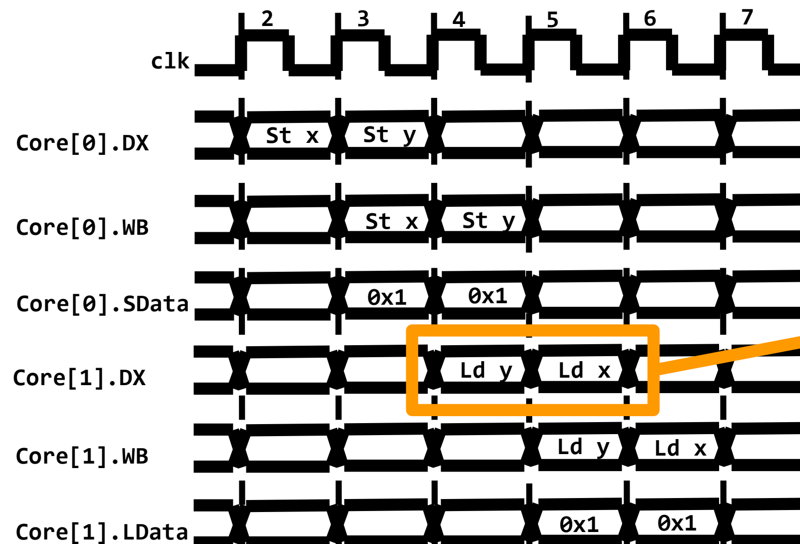
Axiomatic
Microarch. Verification



Abstract nodes and happens-before edges

Axiomatic/Temporal Mismatch!

Temporal
RTL Verification (SVA, etc)



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)

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;

Outcome Filtering in Axiomatic Verification

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

mp (Message Passing)

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;

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 <u>**easy and efficient**</u>

mp (Message Passing)

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;

rf

Outcome: r1 = 1, r2 = 1

<u>**Execution examined as a whole**</u>,
so outcome can be enforced!

Outcome Filtering in Axiomatic Verification

- Axiomatic models make outcome filtering <u>**easy and efficient**</u>

mp (Message Passing)

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;

rf

Outcome: r1 = 1, r2 = 1

<u>**Execution examined as a whole**</u>, so outcome can be enforced!

Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires <u>expensive global analysis</u>
  - <u>Not done</u> by many SVA verifiers, including JasperGold!

mp

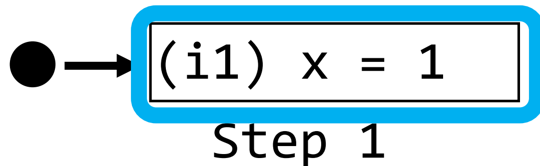
Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
Is r1 = 1, r2 = 0 possible?	

Outcome Filtering in Temporal Verification

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
Is r1 = 1, r2 = 0 possible?	



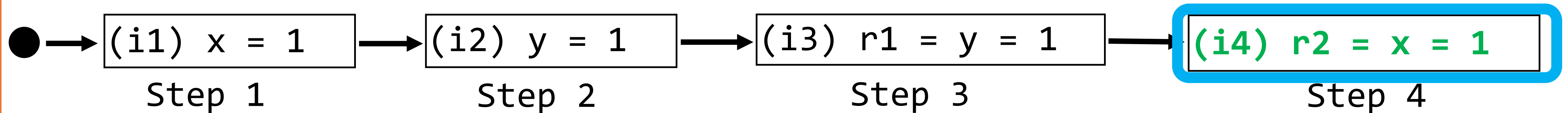
Step 1

Outcome Filtering in Temporal Verification

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
Is r1 = 1, r2 = 0 possible?	

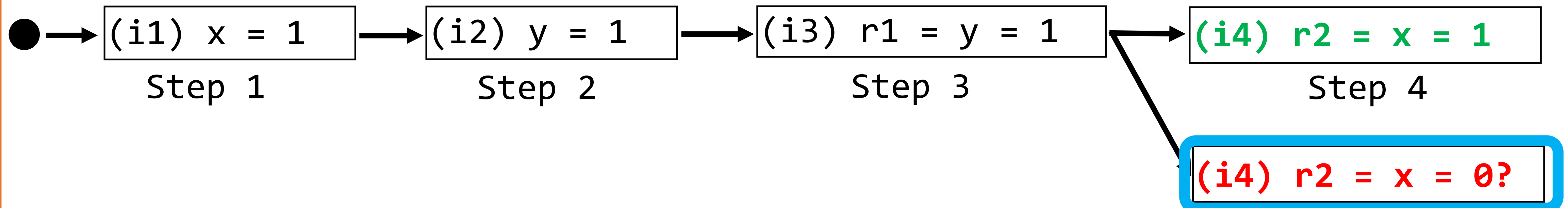


Outcome Filtering in Temporal Verification

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
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

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!**

(i1) x = 1
Step 1

(i3) r1 = y = 0

(i2) y = 1
Step 2

(i3) r1 = y = 1
Step 3

(i4) r2 = x = 1
Step 4

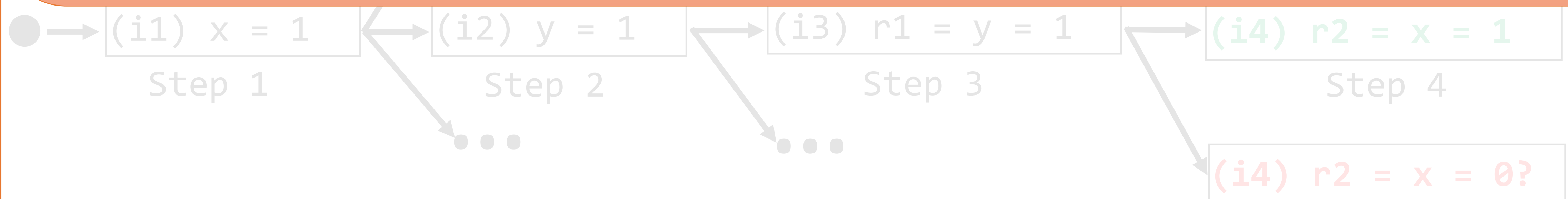
(i4) r2 = x = 0?

Outcome Filtering in Temporal Verification

- Filtering executions by outcome requires <u>expensive global analysis</u>
  - <u>Not done</u> by many SVA verifiers, including JasperGold!

<u>SVA Verifier Approximation:</u> Only check if constraints hold **<u>up to current step</u>**

<u>Makes Outcome Filtering impossible!</u>



µspec Verification Uses Outcome Filtering

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

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

Note: Axioms abstracted for brevity

µspec Verification Uses Outcome Filtering

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

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

Note: Axioms abstracted for brevity

μspec Verification Uses Outcome Filtering

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 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 Verification Uses Outcome Filtering

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

```
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
```

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Note: Axioms/properties abstracted for brevity

Temporal Outcome Filtering Fails!

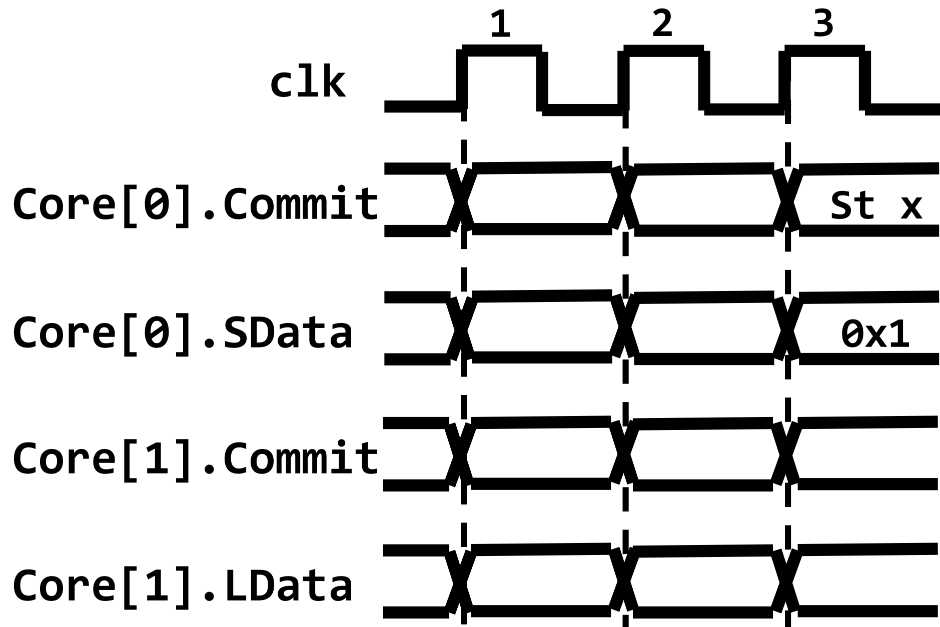
BeforeAllWrites:

Unless Load returns non-zero value,

Load happens before all stores to its address

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	



After 3 cycles:

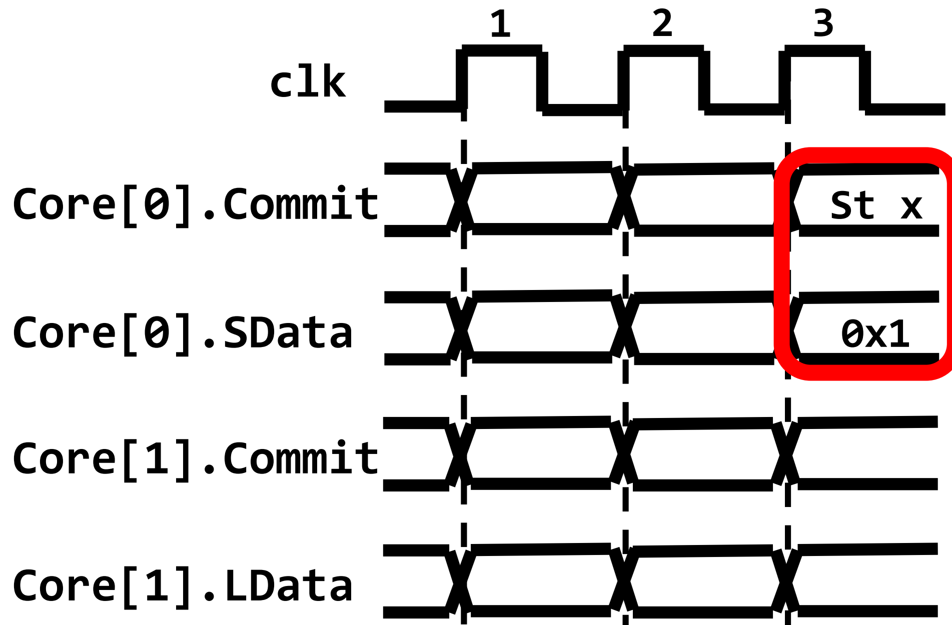
Note: Axioms/properties abstracted for brevity

Temporal Outcome Filtering Fails!

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	



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

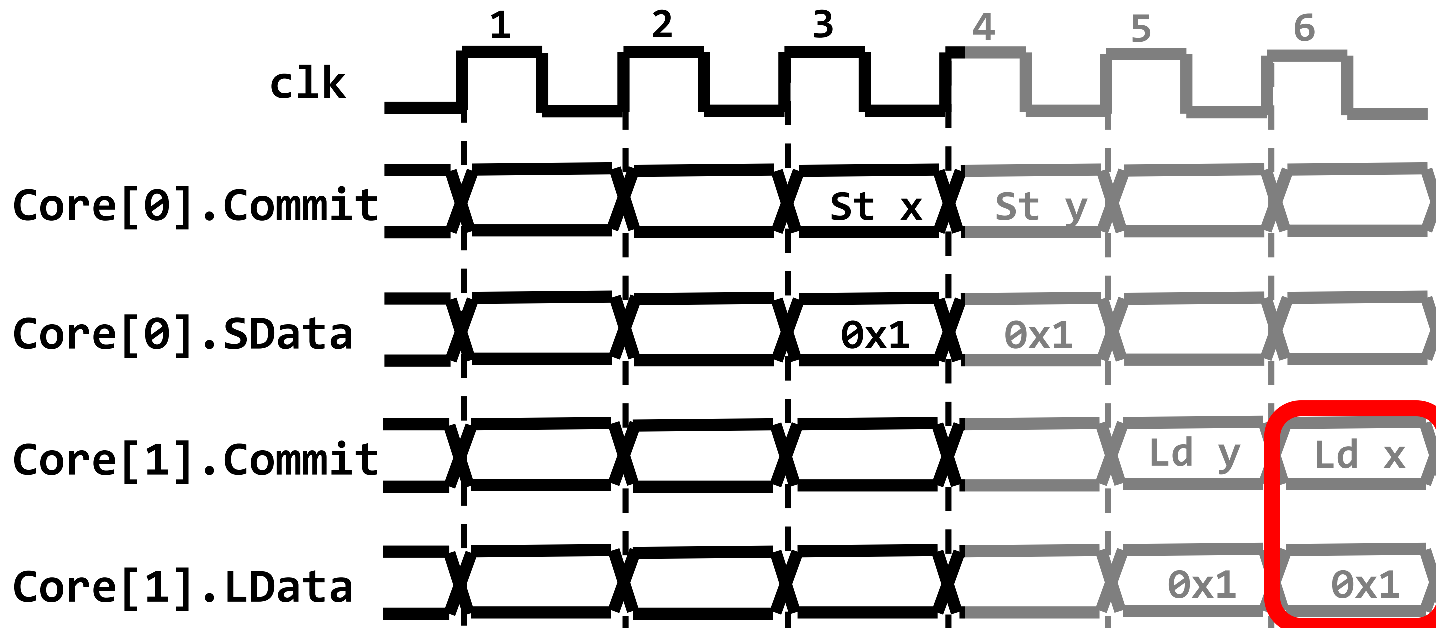
Note: Axioms/properties abstracted for brevity

Temporal Outcome Filtering Fails!

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
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!

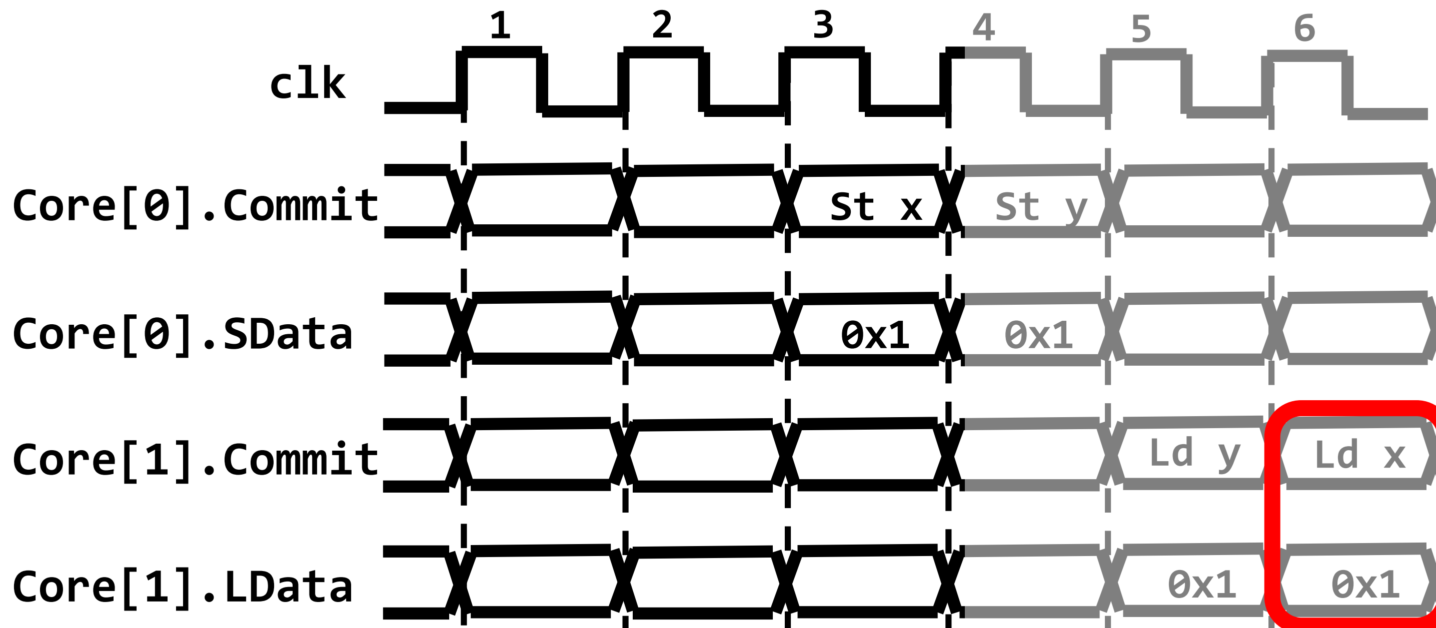
Note: Axioms/properties abstracted for brevity

Temporal Outcome Filtering Fails!

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
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!

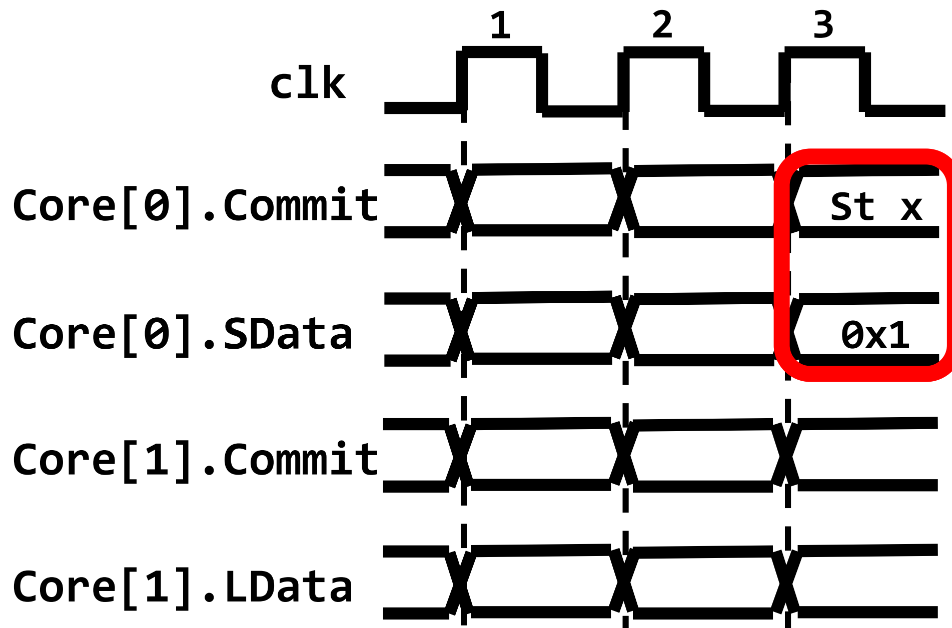
Note: Axioms/properties abstracted for brevity

Temporal Outcome Filtering Fails!

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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
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!

Counterexample flagged despite hardware doing **nothing wrong!**

Note: Axioms/properties abstracted for brevity

Solution: Load Value Constraints

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

- 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)

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
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)

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
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

mp

Core 0	Core 1
(i1) x = 1;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

- 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

3-stage
in-order
pipelines

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 | Core 1 | Core 2 | Core 3

IF → DX → WB (each core)

Arbiter

Memory

Arbiter enforces that only **one core** can access memory at any time

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

Core 0
IF
y = 1
x = 1
Core 1
IF
DX
WB
Core 2
IF
DX
WB
Core 3
IF
DX
WB
Arbiter
Memory
Stores
wdata
Mem array

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

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

Core 0
IF
DX
WB

Core 1
IF
DX
WB

Core 2
IF
DX
WB

Core 3
IF
DX
WB

Arbiter

Memory

Stores

y = 1

Mem array

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Results: Time to Verification

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

Time (hours)
12
10
8
6
4
2
0
safe006
lb
safe007
mp
safe022
safe010
ssl
safe000
safe008
n4
n5
co-mp
safe001
wrc
sb
safe018
podwr000
safe003
mp+staleld
safe012
safe002
safe014
iwp23b
safe009
safe029
safe027
rwc
n2
rfi013
safe030
safe011
rfi015
rfi003
safe021
iriw
n7
iwp24
podwr001
safe017
rfi012
n6
safe019
rfi001
rfi000
rfi011
safe026
safe004
safe016
rfi002
rfi005
rfi014
rfi004
rfi006
n1
amd3
co-iriw
Mean
Hybrid
Full_Proof

Results: Time to Verification

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

Time (hours)

12
10
8
6
4
2
0

safe006 lb safe007 mp safe022 safe010 ssl safe000 safe008 n4 n5 co-mp safe001 wrc sb safe018 podwr000 safe003 safe012 safe002 safe014 iwp23b safe009 safe029 safe027 rwc n2 rfi013 safe030 safe011 rfi015 rfi003 safe021 iriw n7 iwp24 podwr001 safe017 rfi012 n6 safe019 rfi001 rfi000 rfi011 safe026 safe004 safe016 rfi002 rfi005 rfi014 rfi004 rfi006 n1 amd3 co-iriw Mean

Hybrid Full_Proof

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

Time (hours)

12 10 8 6 4 2 0

safe006 lb safe007 mp safe022 safe010 ssl safe000 safe008 n4 n5 co-mp safe001 wrc sb safe018 podwr000 safe003 mp+staleld safe012 safe002 safe014 iwp23b safe009 safe029 safe027 rwc n2 rfi013 safe030 safe011 rfi015 rfi003 safe021 iriw n7 iwp24 podwr001 safe017 rfi012 n6 safe019 rfi001 rfi000 rfi011 safe026 safe004 safe016 rfi002 rfi005 rfi014 rfi004 rfi006 n1 amd3 co-iriw Mean

Hybrid Full_Proof

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

% Proven Properties

100
90
80
70
60
50
40
30
20
10
0

safe006 lb safe007 safe000 n4 safe011 safe016 safe030 rfi000 safe017 safe019 safe004 safe021 rfi011 rfi006 n1 rfi012 n7 co-iriw rfi005 safe002 n2 iriw rfi002 safe012 rfi003 safe003 safe014 safe001 iwp24 rfi015 rfi001 safe026 safe027 podwr001 safe008 rfi014 n6 n5 wrc safe018 rwc safe009 rfi004 amd3 mp+staleld rfi013 mp safe022 safe010 ssl co-mp sb podwr000 iwp23b safe029 Mean

Hybrid Full_Proof

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

% Proven Properties

0 10 20 30 40 50 60 70 80 90 100

safe006 lb safe007 safe000 n4 safe011 safe016 safe030 rfi000 safe017 safe019 safe004 safe021 rfi011 rfi006 n1 rfi012 n7 co-iriw rfi005 safe002 n2 iriw rfi002 safe012 rfi003 safe003 safe014 safe001 iwp24 rfi015 rfi001 safe026 safe027 podwr001 safe008 rfi014 n6 n5 wrc safe018 rwc safe009 rfi004 amd3 mp+staleld rfi013 mp safe022 safe010 ssl co-mp sb podwr000 iwp23b safe029 Mean

Hybrid Full_Proof

MCM Verification: The Big Picture

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

Compiler | OS — [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 / OS — [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]

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

OS

[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]

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

Requires **Bluespec design** and **manual proof**

MCM Verification: The Big Picture

Layer	References
High-Level Languages (HLL)	[Batty et al. POPL 2012] [TriCheck, ASPLOS 2017]
Compiler / OS	[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	**[RTLCheck, MICRO-50]** [Vijayaraghavan et al. CAV 2015] [Choi et al. ICFP 2017]

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**

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