PipeProof (including hands-on):

Verifying simpleSC across all programs
Does hardware correctly implement ISA MCM?

Microarchitecture

Coherence Protocol (SWMR, DVI, etc.)

Litmus Test

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

Under TSO: Forbid r1=1, r2=0

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

Microarchitecture

Coherence Protocol (SWMR, DVI, etc.)

SC/TSO/RISC-V MCM?
PipeCheck vs PipeProof

- **PipeCheck:**
  - Microarch. spec
  - Litmus Test
  - PipeCheck
  - \(\mu\)arch correct for litmus test

- **PipeProof:**
  - Microarch. spec
  - Auxiliary Inputs
  - PipeProof
  - \(\mu\)arch correct for all programs!
Why do we need PipeProof?

- Test-based verification only checks that tested programs run correctly!
- **Open question:** Does a suite of litmus tests cover all µarch bugs?
- **Example:** Remove EnforceWritePPO axiom from simpleSC
  - `/home/check/pipecheck_tutorial/uarches/SC_fillable.uarch`
  - Some orderings between same-core stores and loads removed, violating SC
  - Will bug be detected? *Depends what tests you run!*

Axiom "EnforceWritePPO":

```
forall microop "w",
forall microop "i",
(IsAnyWrite w \ SameCore w i \\
\ EdgeExists((w, Fetch), (i, Fetch), "")) => 
   AddEdge ((w, Writeback), (i, Execute)).
```
SimpleSC without EnforceWritePPO

**mp Litmus Test**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 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>
<tr>
<td>Forbid:</td>
<td>r1 = 1, r2 = 0</td>
</tr>
</tbody>
</table>

**Core 0**

x = 1;
y = 1;

**Core 1**

r1 = y;
r2 = x;

Forbid: r1 = 1, r2 = 0

**sb Litmus Test**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
</tr>
<tr>
<td>r1 = y;</td>
<td>r2 = x;</td>
</tr>
<tr>
<td>Forbid:</td>
<td>r1 = 0, r2 = 0</td>
</tr>
</tbody>
</table>

**Core 0**

x = 1;

**Core 1**

y = 1;

Forbid: r1 = 0, r2 = 0

**Cyclic => Still unobservable**
SimpleSC without EnforceWritePPO

mp Litmus Test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 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>

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

Cyclic => Still unobservable

sb Litmus Test

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

Forbid: \( r1 = 0, \ r2 = 0 \)

Acyclic => BUG!
Different tests catch different bugs!

To catch all bugs, must verify across all programs!
Verifying Across All Possible Programs

- Are all forbidden programs microarchitecturally unobservable?
  - If so, then microarchitecture is correct

- **Infinite** number of forbidden programs
  - E.g.: For SC, must check all possibilities of $cyclic(po \cup co \cup rf \cup fr)$

- How are these ISA-level patterns related to litmus tests?
Symbolic Analysis: Generalise to ISA-Level Cycles

- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

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

mp
Symbolic Analysis: Generalise to ISA-Level Cycles

- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

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

SC Forbids: r1=1, r2=0

Diagram:

- (i1) → (i2) → (i3) → (i4)
- mp
- SC Forbids: r1=1, r2=0
Symbolic Analysis: Generalise to ISA-Level Cycles

- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

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

SC Forbids: r1=1, r2=0

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

SC Forbids: r1=2, r2=0
Symbolic Analysis: Generalise to ISA-Level Cycles

- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

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

SC Forbids: r1=1, r2=0

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

SC Forbids: r1=2, r2=0
PipeProof: What’s Needed

1. Link ISA-level MCM to microarchitectural specification
   • ISA Edge Mapping

2. Add universal constraints that symbolic analysis must respect
   • Theory Lemmas

3. A finite representation of all forbidden ISA-level cycles
   • Transitive Chain (TC) Abstraction

4. Automated refinement checking of the finite representation
   • Microarchitectural Correctness Proof
   • Chain invariants (for termination)
Mapping ISA-Level Edges to Microarchitecture

- Open `/home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch`
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of \( po \) edges

Axiom "Mapping_po":

forall microop "i",
forall microop "j",
(HasDependency po i j =>
  AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue")).
Mapping ISA-Level Edges to Microarchitecture

- Open /home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of $po$ edges

Axiom "Mapping_po": Check whether a po edge from $i$ to $j$ exists

forall microop "i",
forall microop "j",
(HasDependency po i j =>
 AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue"))).
Mapping ISA-Level Edges to Microarchitecture

- Open /home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of \( po \) edges

Axiom "Mapping_po":
for all microop "i",
for all microop "j",
(HasDependency po i j =>

```
AddEdge (((i, Fetch), (j, Fetch), "po_arch", "blue"));
```
)
Mapping ISA-Level Edges to Microarchitecture

- Open /home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of $po$ edges

**Axiom "Mapping_po":**

```plaintext
forall microop "i",
forall microop "j",
(HasDependency po i j =>
  AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue"))
```

*Blue edges between EX and WB stages added by other FIFO axioms (refer to µspec file)*
Mapping Axioms Hands-on

- How about mapping co (coherence order) edges?

- Hint:
  
  * po edge mapping was similar to PO_Fetch axiom
  * co edge mapping is based on WriteSerialization axiom

Axiom "Mapping_co":
forall microop "i",
forall microop "j",
(HasDependency co i j => SamePhysicalAddress i j /
   AddEdge ([(i, __________)], (j, __________), "co_arch")).
Mapping Axioms Hands-on

- How about mapping \( co \) (coherence order) edges?
- Hint:
  - \( po \) edge mapping was similar to PO_Fetch axiom
  - \( co \) edge mapping is based on WriteSerialization axiom

Axiom "Mapping_co":
forall microop "i",
forall microop "j",
(HasDependency co i j => SamePhysicalAddress i j \( / \) AddEdge ((i, Writeback), (j, Writeback), "co_arch")).
ISA Edge Mappings for SimpleSC

- Refer to `simpleSC_fill.uarch` to see mapping axioms for `rf, fr`
PipeProof: What’s Needed

1. Link ISA-level MCM to microarchitectural specification
   • ISA Edge Mapping

2. Add universal constraints that symbolic analysis must respect
   • Theory Lemmas

3. A finite representation of all forbidden ISA-level cycles
   • Transitive Chain (TC) Abstraction

4. Automated refinement checking of the finite representation
   • Microarchitectural Correctness Proof
   • Chain invariants (for termination)
Symbolic Analysis Requires Theory Lemmas

- Symbolic analysis: predicates are just variables that can be true or false
  - “Theory Lemmas” necessary to enforce “universal” laws on predicates
- **Example:** Is an instruction guaranteed to be a read or write?

```
 i: r1 = Load [x]
```

**Concrete:** Look at instruction -> IsAnyRead i is true
Symbolic Analysis Requires Theory Lemmas

- Symbolic analysis: predicates are just variables that can be true or false
  - “Theory Lemmas” necessary to enforce “universal” laws on predicates
- **Example:** Is an instruction guaranteed to be a read or write?

**Concrete:** Look at instruction -> **IsAnyRead i is true**

**Symbolic:** We now know nothing about the instruction!
Both **IsAnyRead i** and **IsAnyWrite i** could be false! (even though this can’t happen in reality)
Symbolic Analysis Requires Theory Lemmas

- Symbolic analysis: predicates are just variables that can be true or false
  - “Theory Lemmas” necessary to enforce “universal” laws on predicates
- **Example:** Is an instruction guaranteed to be a read or write?

Concrete: Look at instruction -> *IsAnyRead* \(i\) is true

Symbolic: We now know nothing about the instruction!
Both *IsAnyRead* \(i\) and *IsAnyWrite* \(i\) could be false! (even though this can’t happen in reality)

Need Additional Theory Lemma to enforce that op is either a read or write!

Axiom “Theory_Lemmas”:
forall microop "i",
...
"IsAnyRead\(i\) \lor\ IsAnyWrite\(i\)".
Theory Lemmas: Hands-on

Concrete: Directly compare instructions i and k -> SamePhysicalAddress i k is true
Concrete: Directly compare instructions i and k -> $\text{SamePhysicalAddress } i \text{ k}$ is true

Symbolic: co edge mapping gives $\text{SamePhysicalAddress } i \text{ j}$ and $\text{SamePhysicalAddress } j \text{ k}$

But $\text{SamePhysicalAddress } i \text{ k}$ could be false! (even though this can never happen in reality)
Concrete: Directly compare instructions $i$ and $k$ - > \texttt{SamePhysicalAddress i k is true}

Symbolic: co edge mapping gives \texttt{SamePhysicalAddress i j} and \texttt{SamePhysicalAddress j k}

But \texttt{SamePhysicalAddress i k} could be false! (even though this can never happen in reality)

Need Additional Theory Lemma for Transitivity of \texttt{SamePhysicalAddress}!

Axiom “Theory\_Lemmas”:
for all microop "i",
...
for all microop "j",
...
for all microop “k”,
(SamePhysicalAddress _ _ \&\& SamePhysicalAddress _ _ =>
SamePhysicalAddress _ _)
...
**Concrete**: Directly compare instructions $i$ and $k$ -> $\text{SamePhysicalAddress } i \text{ } k$ is true

**Symbolic**: co edge mapping gives $\text{SamePhysicalAddress } i \text{ } j$ and $\text{SamePhysicalAddress } j \text{ } k$

But $\text{SamePhysicalAddress } i \text{ } k$ could be false! (even though this can never happen in reality)

**Need Additional Theory Lemma for Transitivity of SamePhysicalAddress!**

Axiom “Theory_Lemmas”:
for all microop "i",
...
for all microop "j",
...
for all microop “k”,
$(\text{SamePhysicalAddress } i \text{ } j \land \text{SamePhysicalAddress } j \text{ } k \Rightarrow \text{SamePhysicalAddress } i \text{ } k)$...
PipeProof: What’s Needed

1. Link ISA-level MCM to microarchitectural specification
   • ISA Edge Mapping

2. Add universal constraints that symbolic analysis must respect
   • Theory Lemmas

3. A finite representation of all forbidden ISA-level cycles
   • Transitive Chain (TC) Abstraction

4. Automated refinement checking of the finite representation
   • Microarchitectural Correctness Proof
   • Chain invariants (for termination)
Verifying Across All Possible Programs

- **Infinite** number of forbidden programs
  - E.g.: For SC, must check all possibilities of \( \text{cyclic}(po \cup co \cup rf \cup fr) \)

- Prove using **abstractions and induction**
  - Based on Counterexample-guided abstraction refinement [Clarke et al. CAV 2000]
Verifying Across All Possible Programs

- **Infinite** number of forbidden programs
  - E.g.: For SC, must check all possibilities of \( \text{cyclic}( po \cup co \cup rf \cup fr) \)

- Prove using **abstractions and induction**
  - Based on Counterexample-guided abstraction refinement [Clarke et al. CAV 2000]
The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

(Infinite set)
All non-unary cycles containing \texttt{fr} (Infinite set)

\texttt{fr}

\begin{align*}
& \quad i_1 \overset{co}{\rightarrow} i_2 \overset{po}{\rightarrow} i_3 \\
& \quad i_1 \overset{po}{\rightarrow} i_2 \\
& \quad i_1 \overset{rf}{\rightarrow} i_2 \overset{co}{\rightarrow} i_3 \overset{po}{\rightarrow} i_4 \\
& \quad \cdots \\
& \quad i_1 \overset{rf}{\rightarrow} i_2 \overset{co}{\rightarrow} i_3 \overset{po}{\rightarrow} i_4
\end{align*}

\textbf{Cycle} = \textbf{Transitive Chain (sequence)} + \textbf{Loopback edge (fr)}
The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr (Infinite set)

Transitive chain (sequence) of ISA-level edges

Cycle = Transitive Chain (sequence) + Loopback edge (fr)
The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr (Infinite set)

Cycle = Transitive Chain (sequence) + Loopback edge (fr)

ISA-level transitive chain => Microarch. level transitive connection

The Transitive Chain (sequence)
+ Loopback edge (fr)
The Transitive Chain (TC) Abstraction

Infinite!

\[ 
\begin{array}{c}
  \text{fr} \\
  i_1 \xrightarrow{\text{co}} i_2 \xrightarrow{\text{po}} i_3 \\
  \text{fr} \\
  i_1 \xrightarrow{\text{po}} i_2 \xrightarrow{\text{rf}} i_3 \xrightarrow{\text{po}} i_4 \\
  \text{fr} \\
  i_1 \xrightarrow{\text{rf}} i_2 \xrightarrow{\text{co}} i_3 \xrightarrow{\text{po}} i_4 \\
  \text{fr} \\
  i_1 \xrightarrow{\text{po}} i_2 \\
  \ldots
\end{array} \]
The Transitive Chain (TC) Abstraction

Infinite!

Finite!

Using TC Abstraction

3 x 3 = 9 possible transitive connections from \( i_1 \) to \( i_n \)
The Transitive Chain (TC) Abstraction

Infinite!

\[ \text{fr} \]
\[ i_1 \rightarrow i_2 \rightarrow i_3 \]

\[ \text{po} \]

\[ i_2 \rightarrow i_3 \rightarrow i_4 \]

Finite!

Using TC Abstraction

\[ \text{fr} \]
\[ i_1 \rightarrow i_2 \rightarrow i_3 \rightarrow i_4 \]

\[ \text{co} \]
\[ i_2 \rightarrow i_3 \rightarrow i_4 \]

\[ \text{po} \]

\[ i_3 \rightarrow i_4 \rightarrow i_1 \]

\[ \ldots \]
The Transitive Chain (TC) Abstraction

Abstraction soundness automatically verified as a supporting proof!

Using TC Abstraction
**Transitive Chain (TC) Abstraction Support Proof**

- Ensure that ISA-level pattern and µarch. support TC Abstraction

- **Base case:** Do initial ISA-level edges guarantee connection?

- **Inductive case:** Extend transitive chain => extend transitive connection?
PipeProof: What’s Needed

1. Link ISA-level MCM to microarchitectural specification
   • ISA Edge Mapping

2. Add universal constraints that symbolic analysis must respect
   • Theory Lemmas

3. A finite representation of all forbidden ISA-level cycles
   • Transitive Chain (TC) Abstraction

4. Automated refinement checking of the finite representation
   • Microarchitectural Correctness Proof
   • Chain invariants (for termination)
Microarchitectural Correctness Proof

Cycles containing $fr$

i_1 \rightarrow i_n (transitive connection)

All possible transitive connections

Cycles containing $po$

i_1 \rightarrow i_n (transitive connection)

Other ISA-level cycles...
Microarchitectural Correctness Proof

Cycles containing fr

All possible transitive connections

Cycles containing po

Other ISA-level cycles...

Other transitive connections...

NoDecomp

Microarchitectural Correctness Proof
Microarchitectural Correctness Proof

Cycles containing fr

Cycles containing po

All possible transitive connections

NoDecomp

AbsCounterX?

Acyclic graph with transitive connection =>

Abstract Counterexample (i.e. possible bug)
Microarchitectural Correctness Proof

Cycles containing \(fr\)

\[\begin{array}{c}
\text{i}_1 \\
\text{i} \\
\text{n}
\end{array}\]

Some \(\mu h b\) edge from \(\text{i}_1\) to \(\text{i}_n\) (transitive connection)

Cycles containing \(po\)

\[\begin{array}{c}
\text{i}_1 \\
\text{i} \\
\text{n}
\end{array}\]

Some \(\mu h b\) edge from \(\text{i}_1\) to \(\text{i}_n\) (transitive connection)

All possible transitive connections

NoDecomp

\[\begin{array}{c}
\text{i}_1 \\
\text{i} \\
\text{n}
\end{array}\]

Transitive connection (green edge) may represent one or multiple ISA-level edges

AbsCounterX

\[\begin{array}{c}
\text{i}_1 \\
\text{i} \\
\text{n}
\end{array}\]

Other ISA-level cycles...

Other transitive connections...
Microarchitectural Correctness Proof

Cycles containing $fr$

Cycles containing $po$

All possible transitive connections

Try to Concretize (Replace transitive connection with one ISA-level edge)

Microarchitectural Buggy, Return Counterexample

Transitive connection (green edge) may represent one or multiple ISA-level edges
Microarchitectural Correctness Proof

Cycles containing fr

Some µhb edge from \( i_1 \) to \( i_n \) (transitive connection)

All possible transitive connections

NoDecomp

Transitive connection (green edge) may represent one or multiple ISA-level edges

Cycles containing po

Some µhb edge from \( i_1 \) to \( i_n \) (transitive connection)

AbsCounterX

Try to Concretize (Replace transitive connection with one ISA-level edge)

Consider all Decompositions (Inductively break down Transitive Chain)

Other ISA-level cycles...

Other transitive connections...

Microarch Buggy, Return Counterexample

Unobs.

Observable
Microarchitectural Correctness Proof

Cycles containing \(fr\)
- Some µb edge from \(i_1\) to \(i_n\) (transitive connection)

Cycles containing \(po\)
- Some µb edge from \(i_1\) to \(i_n\) (transitive connection)

All possible transitive connections

NoDecomp
- Transitive connection (green edge) may represent one or multiple ISA-level edges

"Refinement Loop"

Try to Concretize (Replace transitive connection with one ISA-level edge)

Unobs.
- Consider all Decompositions (Inductively break down Transitive Chain)

Observable

Other transitive connections...

Microarch Buggy, Return Counterexample

Cycles containing \(fr\)

Other ISA-level cycles...

Microarchitectural Correctness Proof

Transitive connection (green edge) may represent one or multiple ISA-level edges
Refinement Loop: Concretization

- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)
Refinement Loop: Concretization

- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)
Refinement Loop: Concretization

- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)
Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

\[
\text{factorial}(n) = \text{factorial}(n-1) \times n
\]
Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

\[
\text{factorial}(n) = \text{factorial}(n-1) \times n \\
\text{Chain of length } n = \text{Chain of length } n-1 + \text{“Peeled-off” edge}
\]
Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

\[
\text{factorial}(n) = \text{factorial}(n-1) \times n
\]

Chain of length \(n\) = Chain of length \(n-1\) + “Peeled-off” edge
Refinement Loop: Decomposition

- Inductively break down transitive chain
  
  • Additional constraints may be enough to make execution unobservable

\[
\text{factorial}(n) = \text{factorial}(n-1) \times n
\]

\[
\text{Chain of length } n = \text{Chain of length } n-1 + \text{“Peeled-off” edge}
\]
Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

\[
\text{factorial}(n) = \text{factorial}(n-1) \times n
\]

Chain of length \( n \) = Chain of length \( n-1 \) + “Peeled-off” edge

If decomposition is abstract counterexample, repeat concretization and decomposition!
Hands-on: Let’s Run PipeProof!

# Assuming you are in ~/pipeproof_tutorial/uarches/
$ prove_uarch -m simpleSC_fill.uarch -i SC -n

- What happens?
Hands-on: Let’s Run PipeProof!

- PipeProof does not terminate; why?

```plaintext
... // Checking Path: (1/1, fr;)
// Checking Path: (1/1, fr;)(1/1, po;fr;)
// Checking Path: (1/1, fr;)(1/1, po;fr;)(1/1, po;po;fr;)
// Checking Path: (1/1, fr;)(1/1, po;fr;)(1/1, po;po;fr;)(1/1, po;po;po;fr;)
...```

...
Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- Inductively proven by PipeProof before their use in proof algorithms
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)
Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- Inductively proven by PipeProof before their use in proof algorithms
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)
Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- **Inductively proven by PipeProof before their use in proof algorithms**
- **Example:** checking for edge from $i_1$ to $i_5$ (TC abstraction support proof)

Can continue decomposing in this way forever!
Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- **Inductively proven by PipeProof before their use in proof algorithms**
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)

![Chain Invariant Applied Diagram]

- **po_plus** = arbitrary number of repetitions of **po**
- Next edge peeled off will be something other than **po**
Adding the Chain Invariant for po+

- Uncomment the invariant at the end of `simpleSC_fill.uarch`:

```
Axiom "Invariant_poplus":
forall microop "i",
forall microop "j",
HasDependency po_plus i j =>
   (AddEdge ((i, Fetch), (j, Fetch), "") \ SameCore i j).
```

- Now re-run PipeProof:

```
# Assuming you are in ~.pipeproof_tutorial/uarches/
$ prove_uarch -m simpleSC_fill.uarch -i SC
```

- Should be proven in about a minute on the VM
PipeProof Block Diagram

- Microarchitecture Ordering Spec.
- ISA-Level MCM Spec.
- ISA Edge -> Microarch. Mapping
- Chain Invariants

**Result:** All-Program MCM Correctness Proof?
Counterexample found?

- Proof of Chain Invariants
  - Transitive Chain Abstraction Support Proof
    - Generate Counterexample
    - Microarch. Correctness Proof
      - Theory Lemmas
PipeProof Does the Difficult Stuff for You!

- Users simply provide axioms, mappings, theory lemmas, and invariants
- PipeProof takes care of:
  - Proving TC Abstraction soundness
  - Proving any chain invariants
  - Refining abstraction (concretization and decomposition)
  - Inductively generating ISA-level cycles and covering all possibilities
- Architects can use PipeProof; not just for formal methods experts!
PipeProof: TSO Case Study

- Provided in VM as `solutions/simpleTSO.uarch`
  - Can try on your own time
  - Requires additional ISA-level relations, theory lemmas, and chain invariants
  - Will take at least 41 minutes to verify
Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO\(^1\)) μarches
  - 3-stage in-order pipelines

- TSO verification made feasible by optimizations
  - Explicitly checking all decompositions => case explosion
  - **Covering Sets Optimization** (eliminate redundant transitive connections)
  - **Memoization** (eliminate previously checked ISA-level cycles)

<table>
<thead>
<tr>
<th></th>
<th>simpleSC</th>
<th>simpleSC (w/ Covering Sets + Memoization)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Time</strong></td>
<td>225.9 sec</td>
<td>19.1 sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>simpleTSO (w/ Covering Sets + Memoization)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Time</strong></td>
<td>Timeout</td>
</tr>
<tr>
<td></td>
<td>2449.7 sec (≈ 41 mins)</td>
</tr>
</tbody>
</table>

\(^1\)TSO (Total Store Order) is the MCM of Intel x86 processors. It relaxes Store->Load ordering.
PipeProof Takeaways

- Automated All-Program Microarchitectural MCM Verification
  - Designers no longer need to choose between completeness and automation
  - Can verify microarchitectures themselves, before RTL is written!
- Based on techniques from formal methods (CEGAR) [Clarke et al. CAV 2000]
- Transitive Chain (TC) Abstraction models infinite set of executions
- Open-source: https://github.com/ymanerka/pipeproof
- Accolades:
  - Nominated for Best Paper at MICRO 2018
  - “Hon. Mention” from 2018 IEEE Micro Top Picks of Comp. Arch. Conferences
Backup Slides
Covering Sets Optimization

- Must verify across all possible transitive connections

- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion

- The Covering Sets Optimization eliminates redundant transitive connections
Covering Sets Optimization

- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

Graph A has an edge from \( x \rightarrow z \) (tran conn.)
Covering Sets Optimization

- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

Graph A has an edge from $x \rightarrow z$ (tran conn.)

Graph B has edges from $y \rightarrow z$ (tran conn.) and $x \rightarrow z$ (by transitivity)
Covering Sets Optimization

- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

- Graph A has an edge from $x \rightarrow z$ (tran conn.)
- Graph B has edges from $y \rightarrow z$ (tran conn.) and $x \rightarrow z$ (by transitivity)

Correctness of A => Correctness of B (since B contains A’s tran conn.)

Checking B explicitly is redundant!
Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

Diagram:

```
   i1 ------- po ------- i2
   |            |            |
   rf          fr          po
   |            |            |
   i3 ------- po ------- i4
```

Legend:
- po: Edge from i1 to i2
- rf: Edge from i1 to i3
- fr: Edge from i2 to i3
- po: Edge from i3 to i4
Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times.
- Memoization eliminates redundant checks of cycles that have already been verified.

![Diagram of cycle](image)
Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified
Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

Same cycle is checked 3 times!
Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

Procedure: If all ISA-level cycles containing edge $r_i$ have been checked, do not peel off $r_i$ edges when checking subsequent cycles
The Adequate Model Over-Approximation

- Addition of an instruction can make unobservable execution observable!
- Need to work with over-approximation of microarchitectural constraints
- PipeProof sets all `exists` clauses to true as its over-approximation
Filtering Invalid Decompositions

- When decomposing a transitive connection, the decomposition should guarantee the transitive connections of its parent abstract cexes.
- Decompositions that do not do this are invalid and filtered out.
PipeProof Block Diagram

Microarchitecture Ordering Spec. ➔ ISA-Level MCM Spec. ➔ ISA Edge -> Microarch. Mapping ➔ Chain Invariants

Proof of Chain Invariants ➔ Transitive Chain Abstraction Support Proof ➔ Microarch. Correctness Proof

PipeProof

Result: All-Program MCM Correctness Proof? Counterexample found?
PipeProof Block Diagram

Microarchitecture Ordering Spec. → ISA-Level MCM Spec. → ISA Edge -> Microarch. Mapping → Chain Invariants

- Proof of Chain Invariants
- Transitive Chain Abstraction Support Proof
- Microarch. Correctness Proof

PipeProof

Result: All-Program MCM Correctness Proof?
Counterexample found?
PipeProof Block Diagram

Links ISA-level and µarch executions

- Microarchitecture Ordering Spec.
- ISA-Level MCM Spec.
- ISA Edge -> Microarch. Mapping
- Chain Invariants

- Proof of Chain Invariants
- Transitive Chain Abstraction Support Proof
- Microarch. Correctness Proof
- Cex. Generation

Result: All-Program MCM Correctness Proof? Counterexample found?
PipeProof Block Diagram

Microarchitecture Ordering Spec. → ISA-Level MCM Spec. → ISA Edge → Microarch. Mapping → Chain Invariants

Proof of Chain Invariants → Transitive Chain Abstraction Support Proof → Microarch. Correctness Proof

Cex. Generation

PipeProof

Result: All-Program MCM Correctness Proof? Counterexample found?

Represent repeated ISA-level patterns
PipeProof Block Diagram

Microarchitecture Ordering Spec.  ISA-Level MCM Spec.  ISA Edge -> Microarch. Mapping  Chain Invariants

Proof of Chain Invariants  Transitive Chain Abstraction Support Proof  Microarch. Correctness Proof

Cex. Generation

Result: All-Program MCM Correctness Proof?  Counterexample found?

If design can’t be verified, a counterexample (a forbidden execution that is observable) is often returned
PipeProof Block Diagram

- Microarchitecture Ordering Spec.
- ISA-Level MCM Spec.
- ISA Edge -> Microarch. Mapping
- Chain Invariants

Supporting proofs provide foundation for correctness proof

Result: All-Program MCM Correctness Proof?
Counterexample found?