PipeProof:
Automated Memory Consistency Proofs for Microarchitectural Specifications

Yatin A. Manerkar, Daniel Lustig*, Margaret Martonosi, and Aarti Gupta

Princeton University *NVIDIA

MICRO-51

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

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for all possible programs
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for all possible programs
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for all possible programs

Target for compilers...

Compiler

ISA-Level MCM (x86-TSO, Power, ARMv8, etc)

Microarchitecture
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for **all possible programs**

Diagram:

```
Compiler

Target for compilers...

ISA-Level MCM (x86-TSO, Power, ARMv8, etc)

...and a specification that microarchitecture must implement

Microarchitecture
```
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for all possible programs

Compiler

Target for compilers...

???

...and a specification that microarchitecture must implement

Microarchitecture
The Infinite Forest
The Infinite Forest

Forest goes on forever (infinite number of possible programs)
The Infinite Forest

Can check known hideouts (verify design for test programs)

[Images: HeeWann Kim, tzblacktd, audino]
Are Pokemon lurking in unexplored areas? **(Do tested programs provide complete coverage?)**
Have we caught all the Pokemon? *(Are there any MCM bugs left in the design?)*
PipeProof Overview

- First automated all-program microarchitectural MCM verification!
  - Covers all possible addresses, values, numbers of cores
- Proof methodology based on automatic abstraction refinement
- Early-stage: Can be conducted before RTL is written!
Outline

- Background
  - ISA-level MCM specs
  - Microarchitectural ordering specs

- Microarchitectural Correctness Proof
  - Transitive Chain (TC) Abstraction

- Overall PipeProof Operation
  - TC Abstraction Support Proof
  - Chain Invariants

- Results
**ISA-Level MCM Specifications**

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]

- ISA-level executions are graphs
  - **Nodes**: instructions, **edges**: ISA-level relations between instrs

- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$

---

**Message passing (mp) litmus test**

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

---

**An ISA-level execution of mp**

(i1) \([x] \leftarrow 1\)

(i2) \([y] \leftarrow 1\)

(i3) \(r1 \leftarrow [y]\)

(i4) \(r2 \leftarrow [x]\)
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes**: instructions, **edges**: ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$

Message passing (mp) litmus test

<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

An ISA-level execution of mp
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes**: instructions, **edges**: ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - Eg: SC is $\text{acyclic}(po \cup co \cup rf \cup fr)$

**Message passing (mp) 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

![An ISA-level execution of mp](image-url)
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes**: instructions, **edges**: ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - Eg: SC is \( \text{acyclic}(po \cup co \cup rf \cup fr) \)

### Message passing (mp) 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

### An ISA-level execution of mp

- (i1) \([x] \leftarrow 1\)
- (i2) \([y] \leftarrow 1\)
- (i3) \(r1 \leftarrow [y]\)
- (i4) \(r2 \leftarrow [x]\)

- **po**
- **rf**
- **fr**
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]

- ISA-level executions are graphs
  - **Nodes:** instructions, **edges:** ISA-level relations between instrs

- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - Eg: SC is \texttt{acyclic}(\texttt{po} \cup \texttt{co} \cup \texttt{rf} \cup \texttt{fr})

Message passing (mp) litmus test

<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 \(r1=1, r2=0\)

An ISA-level execution of mp

- \(i1\): \([x] \leftarrow 1\)
- \(i2\): \([y] \leftarrow 1\)
- \(i3\): \(r1 \leftarrow [y]\)
- \(i4\): \(r2 \leftarrow [x]\)

- **po:** \(x\) is passed from \(i1\) to \(i4\)
- **rf:** \(x\) is read from \(i2\) before being written to \(i3\)
- **fr:** \(x\) is read from \(i2\) before being written to \(i4\)
- **po:** \(x\) is passed from \(i1\) to \(i4\)
Microarchitectural Ordering Specifications

- Set of axioms in µspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as µhb graphs
  - **Nodes**: instr. sub-events, **edges**: happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

**Message passing (mp) 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 Ordering Specifications

- Set of axioms in μspec DSL [Lustig et al. ASPLOS 2016]

- Used to generate microarchitectural executions as μhb graphs
  - **Nodes:** instr. sub-events, **edges:** happens-before relations between instrs

- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

---

**Message passing (mp) 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 Ordering Specifications

- Set of axioms in μspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as \( \mu hb \) graphs
  - **Nodes**: instr. sub-events, **edges**: happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph \( \rightarrow \) Unobservable
  - Acyclic graph \( \rightarrow \) Observable

Message passing (mp) litmus test

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

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

A \( \mu hb \) graph of mp on simpleSC

\( \text{fr} \) \( \text{po} \) \( \text{rf} \) \( \text{po} \)
Microarchitectural Ordering Specifications

- Set of axioms in $\mu$spec DSL [Lustig et al. ASPLOS 2016]

- Used to generate microarchitectural executions as $\mu$h$\mu$b graphs
  - **Nodes**: instr. sub-events, **edges**: happens-before relations between instrs

- Observability based on cyclicity of graphs
  - Cyclic graph $\rightarrow$ Unobservable
  - Acyclic graph $\rightarrow$ Observable

---

**Message passing (mp) litmus test**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) $[x] \leftarrow 1$</td>
<td>(i3) $r_1 \leftarrow [y]$</td>
</tr>
<tr>
<td>(i2) $[y] \leftarrow 1$</td>
<td>(i4) $r_2 \leftarrow [x]$</td>
</tr>
</tbody>
</table>

Under SC: Forbid $r_1=1$, $r_2=0
Microarchitectural Ordering Specifications

- Set of axioms in \( \mu \text{spec} \) DSL [Lustig et al. ASPLOS 2016]

- Used to generate microarchitectural executions as \( \mu \text{hb} \) graphs
  - **Nodes**: instr. sub-events, **edges**: happens-before relations between instrs

- Observability based on cyclicity of graphs
  - Cyclic graph \( \rightarrow \) Unobservable
  - Acyclic graph \( \rightarrow \) Observable

Message passing (mp) litmus test

<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
Our Prior Work: Litmus Test-Based MCM Verification

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

[Lustig et al. MICRO-47, ...]
Our Prior Work: Litmus Test-Based MCM Verification

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

[Lustig et al. MICRO-47, ...]
Our Prior Work: Litmus Test-Based MCM Verification

Microarchitecture in μspec DSL

**Axiom “Decode_is_FIFO”:**

... \( \text{EdgeExists} \) ((\( i_1 \), Decode), (\( i_2 \), Decode))

\( \Rightarrow \) \( \text{AddEdge} \) ((\( i_1 \), Execute), (\( i_2 \), Execute)).

... 

**Axiom "PO_Fetch":**

... \( \text{SameCore} \ i_1 \ i_2 \ /\ \text{ProgramOrder} \ i_1 \ i_2 \Rightarrow \)

\( \text{AddEdge} \) ((\( i_1 \), Fetch), (\( i_2 \), Fetch)).

<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>

Under SC: Forbid \( r_1 = 1 \), \( r_2 = 0 \)

[Lustig et al. MICRO-47, ...]
Our Prior Work: Litmus Test-Based MCM Verification

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)).

<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>

Microarchitectural happens-before (µhb) graphs

[Lustig et al. MICRO-47, ...]
Our Prior Work: Litmus Test-Based MCM Verification

Microarchitecture in μspec DSL

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

Perennial Question:
“Do your litmus tests cover all possible MCM bugs?”

How to automatically prove correctness for all programs?
The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

- \( fr \)
  - \( i_1 \rightarrow_{1 \ldots n-1} i_n \)

- \( fr \)
  - \( i_1 \rightarrow co \rightarrow po \rightarrow i_3 \)

- \( fr \)
  - \( i_1 \rightarrow po \rightarrow i_2 \)

- \( fr \)
  - \( i_1 \rightarrow po \rightarrow rf \rightarrow po \)
  - \( i_3 \rightarrow po \)
  - \( i_4 \)

- \( fr \)
  - \( i_1 \rightarrow rf \rightarrow co \rightarrow po \)
  - \( i_2 \)
  - \( i_3 \)
  - \( i_4 \)

- \( \ldots \)
All non-unary cycles containing fr

Transitive chain (sequence) of ISA-level edges
The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

Using TC Abstraction

fr

$\mathbf{r}_{1\ldots n-1}$

$\mathbf{i}_1 \rightarrow \mathbf{i}_n$

fr

$\mathbf{i}_1 \xrightarrow{\mathbf{co}} \mathbf{i}_2 \xrightarrow{\mathbf{po}} \mathbf{i}_3$

fr

$\mathbf{i}_1 \xrightarrow{\mathbf{po}} \mathbf{i}_2 \xrightarrow{\mathbf{rf}} \mathbf{i}_3 \xrightarrow{\mathbf{po}} \mathbf{i}_4$

fr

$\mathbf{i}_1 \xrightarrow{\mathbf{rf}} \mathbf{i}_2 \xrightarrow{\mathbf{co}} \mathbf{i}_3 \xrightarrow{\mathbf{po}} \mathbf{i}_4$

$\ldots$
The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

Using TC Abstraction

Some µhb edge from $i_1$ to $i_n$ (transitive connection)
The Transitive Chain (TC) Abstraction

Infinite

Using TC Abstraction
The Transitive Chain (TC) Abstraction

Using TC Abstraction

Infinite

\[ i_1 \xrightarrow{co} i_2 \xrightarrow{po} i_3 \]

\[ i_1 \xrightarrow{po} i_2 \xrightarrow{rf} i_3 \xrightarrow{po} i_4 \]

\[ i_1 \xrightarrow{rf} i_2 \xrightarrow{co} i_3 \xrightarrow{po} i_4 \]

\[ i_1 \xrightarrow{po} i_2 \]

\[ \ldots \]

Finite!

\[ i_1 \xrightarrow{fr} i_2 \xrightarrow{fr} i_3 \xrightarrow{fr} i_4 \]
The Transitive Chain (TC) Abstraction

Infinite

\[
\begin{align*}
& i_1 \xrightarrow{co} i_2 \xrightarrow{po} i_3 \\
& i_1 \xrightarrow{po} i_2 \xrightarrow{fr} i_3 \xrightarrow{po} i_4 \\
& i_1 \xrightarrow{rf} i_2 \xrightarrow{co} i_3 \xrightarrow{po} i_4 \\
& i_1 \xrightarrow{po} i_2 \\
& \ldots
\end{align*}
\]

Using TC Abstraction

Finite!

\[
\begin{align*}
& i_1 \xrightarrow{fr} i_2 \\
& i_1 \xrightarrow{fr} i_2 \rightarrow i_3 \rightarrow i_4 \\
& \ldots
\end{align*}
\]

Soundness verified as a supporting proof!
Microarchitectural Correctness Proof

Cycles containing $fr$

All possible tran. conns.

Cycles containing $po$

Other ISA-level cycles...
Microarchitectural Correctness Proof

Cycles containing \( fr \)

\[ \text{Some } \mu \text{h} \text{b edge from } i_3 \text{ to } i_n \text{ (transitive connection)} \]

\[ \text{fr} \]

\[ \text{All possible tran. conns.} \]

\[ \text{NoDecomp} \]

Cycles containing \( po \)

\[ \text{Some } \mu \text{h} \text{b edge from } i_3 \text{ to } i_n \text{ (transitive connection)} \]

\[ \text{po} \]

Other ISA-level cycles...

Other transitive connections...
Microarchitectural Correctness Proof

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

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

All possible tran. conns.

\( NoDecomp \)
- \( i_1 \) to \( i_n \)

AbsCounterX?
- \( i_1 \) to \( i_n \)

Other ISA-level cycles...

Other transitive connections...
Microarchitectural Correctness Proof

Cycles containing \( fr \)

\[ i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_n \]

Some \( \mu h b \) edge from \( i_1 \) to \( i_n \) (transitive connection)

All possible tran. conns.

\[ \text{NoDecomp} \]

\[ i_1 \rightarrow i_n \]

\( fr \)

AbsCounterX?

Acyclic graph with transitive connection \( \Rightarrow \)

Abstract Counterexample

(i.e. possible bug)

Cycles containing \( po \)

\[ i_1 \rightarrow i_n \]

Some \( \mu h b \) edge from \( i_1 \) to \( i_n \) (transitive connection)

Other ISA-level cycles...

Other transitive connections...
Microarchitectural Correctness Proof

Cycles containing \( fr \)

All possible tran. conns.

NoDecomp

AbsCounterX?

Cycles containing \( po \)

Transitive connection may represent one or multiple ISA-level edges

Other ISA-level cycles...

Other transitive connections...
Microarchitectural Correctness Proof

Cycles containing \(fr\)

\[
\begin{align*}
&\text{i}_1 \\
\rightarrow &\text{i}_2 \\
\downarrow &\text{Some µhb edge from } \text{i}_2 \text{ to } \text{i}_n \text{ (transitive connection)} \\
\rightarrow &\text{i}_n
\end{align*}
\]

Cycles containing \(po\)

\[
\begin{align*}
&\text{i}_1 \\
\downarrow &\text{Some µhb edge from } \text{i}_2 \text{ to } \text{i}_n \text{ (transitive connection)} \\
\rightarrow &\text{i}_n
\end{align*}
\]

All possible tran. conns.

NoDecomp ✓

\[
\begin{align*}
&\text{i}_1 \\
\rightarrow &\text{i}_2 \\
\downarrow &\text{fr} \\
\rightarrow &\text{i}_n
\end{align*}
\]

AbsCounterX?

\[
\begin{align*}
&\text{i}_1 \\
\rightarrow &\text{i}_2 \\
\downarrow &\text{fr} \\
\rightarrow &\text{i}_n
\end{align*}
\]

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

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

Observable

Microarch Buggy, Return Counterexample
Microarchitectural Correctness Proof

Cycles containing \textit{fr}

- Some µh b edge from \textit{i}_2 to \textit{i}_n (transitive connection)

Cycles containing \textit{po}

- Some µh b edge from \textit{i}_2 to \textit{i}_n (transitive connection)

Other ISA-level cycles...

NoDecomp ✓

AbsCounterX?

All possible tran. conns.

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

Microarch Buggy, Return Counterexample

Consider all Decompositions (Inductively break down Transitive Chain)

Observable

Unobs.
Microarchitectural Correctness Proof

Cycles containing \( fr \)

Cycles containing \( po \)

All possible tran. conns.

AbsCounterX?

NoDecomp

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

Consider all Decompositions (Inductively break down Transitive Chain)

“Refinement Loop”

Unobs.

Observable

Microarch Buggy, Return Counterexample

Other ISA-level cycles...
Concretization

- All concretizations must be unobservable
- Observable concretizations are counterexamples

**Concretization:**
Replace transitive connection with single ISA-level edge
Concretization

- All concretizations must be unobservable
- Observable concretizations are counterexamples

**Concretization:**
Replace transitive connection with single ISA-level edge
Decomposition

- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

\[ \text{Decomposition:} \]
Inductively break down transitive chain

(Chain of length \( n \) == Chain of length \( n-1 \) + single “peeled-off” edge)
Decomposition

- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

Decomposition:
Inductively break down transitive chain
(Chain of length $n =$ Chain of length $n-1 +$ single “peeled-off” edge)
Decomposition

- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

Decomposition:
Inductively break down transitive chain
(Chain of length n == Chain of length n-1 + single “peeled-off” edge)
Decomposition

- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

**Decomposition:**
Inductively break down transitive chain
(Chain of length $n$ == Chain of length $n-1 +$ single “peeled-off” edge)
Decomposition

- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

Decomposition:
Inductively break down transitive chain
(Chain of length n == Chain of length n-1 + single “peeled-off” edge)
Outline

▪ Background
  • ISA-level MCM specs
  • Microarchitectural ordering specs

▪ Microarchitectural Correctness Proof
  • Transitive Chain (TC) Abstraction

▪ Overall PipeProof Operation
  • TC Abstraction Support Proof
  • Chain Invariants

▪ Results
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?
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 ▶ Fail Microarch. Correctness Proof

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

Links ISA-level and \( \mu \)arch 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

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

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

Represent repeated ISA-level patterns
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

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

PipeProof  Cex. Generation

Result: All-Program MCM Correctness Proof? Counterexample found?
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?
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 i1 to i5 (TC abstraction support proof)

Can continue decomposing in this way forever!

![Diagram of Repeating ISA-Level Pattern](image)
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

```
  i1 -- po_plus -- i4   fr   i5
    /       |      /     |
   /        |     /      |
  i2       i3     i4    i5
```

- po_plus = arbitrary number of repetitions of po
- Next edge peeled off will be something other than po
In the paper...

- Optimizations
  - Covering Sets: Eliminate redundant transitive connections
  - Memoization: Eliminate redundant ISA-level cycles

- Inductive ISA edge generation

- Adequate Model Over-Approximation
  - Needed to ensure soundness of PipeProof’s abstraction-based approach

- ...and more!
Results

- Ran PipeProof on `simpleSC` (SC) and `simpleTSO` (TSO) μarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

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

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

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) microarchitectures
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

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

<table>
<thead>
<tr>
<th>SimpleTSO</th>
<th>SimpleTSO (w/ Covering Sets + Memoization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>Timeout</td>
</tr>
</tbody>
</table>
Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) microarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

<table>
<thead>
<tr>
<th></th>
<th>simpleSC</th>
<th>simpleSC (w/ Covering Sets + Memoization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</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>Total Time</td>
<td>Timeout</td>
</tr>
</tbody>
</table>
Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) μarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

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

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

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) microarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

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

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

▪ **PipeProof**: Automated *All-Program* Microarchitectural MCM Verification
  • Designers no longer need to choose between completeness and automation

▪ **Transitive Chain Abstraction** allows inductive modelling and verification of the infinite set of all possible executions
  • Abstraction is automatically refined as necessary to prove correctness

▪ Verified simple microarchitectures implementing SC and TSO in < 1 hour!

Code available at https://github.com/ymanerka/pipeproof
PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications

Yatin A. Manerkar, Daniel Lustig*, Margaret Martonosi, and Aarti Gupta

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

http://check.cs.princeton.edu/
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 $x\to z$ (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
Memoization Optimization

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

```
IF  EX  WB  fr
Some Tran. Conn.
i_1  i_n
```

Diagram:

- i1 → i3
- i2 → i3
- i3 → i4
- i4 → i1
- po → rf → po
- po → fr → po

```
i1  
|   
|→ 
|   
|  
i2  
|  
|  
|→ 
|   
i3  
|  
|  
|→ 
|  
i4  
```
Memoization Optimization

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

```
IF
/------------------------\
|   Some Tran. Conn.    |
|     i_n               |
\------------------------/
EX
/------------------------\
|   Some Tran. Conn.    |
|     i_1               |
\------------------------/
WB
```

```
IF
/------------------------\
|   Some Tran. Conn.    |
|     i_1               |
\------------------------/
WB
```
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 \texttt{exists} clauses to true as its over-approximation

\[
\begin{align*}
\text{SubsetExec} & : \quad i_1 \to i_2 \to i_3 \\
\text{SubsetWithExternal} & : \quad i_1 \to i_2 \to i_3 \to i_4
\end{align*}
\]