PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models

Daniel Lustig*, Michael Pellauer*, and Margaret Martonosi*

+Princeton University  *Intel VSSAD

MICRO 47
Multicore Systems Need Memory Consistency Models

Model: TSO

CPU CPU CPU CPU

L2$

Main Memory
Motivation: Verify correctness of memory consistency model implementation

Model:

CPU | CPU
---|---
L2$
Main Memory

Loads

Store Buffer

Fetch → Decode → Execute → Memory → Writeback
Motivation: Verify correctness of memory consistency model \textbf{implementation}

- Systems are getting more complicated!
Talk Outline

• Architectural Models: Defining Correctness

• PipeCheck μarch-Level Analysis
  – Basic “μarch-happens-before” Graphs
  – Advanced μarch Optimizations
  – Comparing arch and μarch

• Automated Tool and Performance Results

• Conclusions
8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as “write ordered with store-buffer forwarding.” This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (Note the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term “processor” refers to a logical processor. For example, a physical processor supporting multiple cores and/or HyperThreading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
  - writes executed with the CLFLUSH instruction;
  - streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
  - string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as “write ordered with store-buffer forwarding.” This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (Note the memory-ordering model for processor systems are written from the perspective of “processor” refers to a logical processor. For example, HyperThreading Technology is treated as a multi-core processor):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes:
  - writes executed with the CLFLUSH instruction
  - streaming stores (writes) executed with MOVNTDQ, MOVNTPS, and MOVNTPD;
  - string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.

---

Lustig, Pellauer, Martonosi, “PipeCheck”
8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as “write ordered with store-buffer forwarding.” This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model is processor-ordered. For example, a multi-threaded system with one thread using streaming stores (writes) executed with MOVNTDQ, MOVNTPS, and MOVNTPD; string operations (see Section 8.2.4.1); and reads may be reordered with older writes to different locations but not with older writes to the same location.

How to automate verification of microarchitectural implementations?

<table>
<thead>
<tr>
<th></th>
<th>Type A (first)</th>
<th>Type B (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>Load</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Store</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
What is the Definition of “Correct”? 

Preserved Program Order (PPO) for x86-TSO: 
Reorderings Permitted?

<table>
<thead>
<tr>
<th>Type A (first)</th>
<th>Type B (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Store</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
What is the Definition of “Correct”?  

Preserved Program Order (PPO) for x86-TSO: 
Reorderings Permitted?

<table>
<thead>
<tr>
<th></th>
<th>Type A (first)</th>
<th>Type B (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Store</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
What is the Definition of “Correct”?

Initially: \([x] = [y] = 0\)

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

Can core 1 observe (i2) before it observes (i1)?

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]

Lustig, Pellauer, Martonosi, “PipeCheck”
What is the Definition of “Correct”?

Initially: \([x] = [y] = 0\)

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

Can core 1 observe (i2) before it observes (i1)?

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
What is the Definition of “Correct”? 

Initially: $[x]=[y]=0$ 

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

Can core 1 observe (i2) before it observes (i1)?

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
What is the Definition of “Correct”? 

Initially: $[x]=[y]=0$

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

$r1=1$

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]

Lustig, Pellauer, Martonosi, “PipeCheck”
What is the Definition of “Correct”?  

Initially: \([x]=[y]=0\)

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

\(r1=1, r2=0\)

Reads from earlier store ("from-reads")

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
What is the Definition of “Correct”? 

Litmus test mp:
Initially: \([x]=[y]=0\)

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

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

Cycle implies forbidden execution: instruction can’t happen before itself

Reads from earlier store ("from-reads")

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
Is the **Implementation** Correct?

How is the consistency model **enforced at the microarchitecture level?**

Lustig, Pellauer, Martonosi, “PipeCheck”
Is the **Implementation** Correct?

How is the consistency model **enforced at the microarchitecture level?**

Lustig, Pellauer, Martonosi, “PipeCheck”
Is the **Implementation** Correct?

How is the consistency model enforced at the microarchitecture level?

Writeback → Memory Store → Memory Hierarchy

Fetch → Decode → Execute → Memory → Writeback

Diagram:

- **i1**: Sx1
- **i2**: Sy1
- **i3**: Ly1
- **i4**: Lx0

Reads from earlier store ("from-reads")

Reads from

PPO
Is the **Implementation** Correct?

How is the consistency model enforced at the microarchitecture level?

- Fetch
- Decode
- Execute
- Memory
- Writeback

Diagram:

- Stores: i1, Sx1, i3, Ly1, i2, Sy1, i4, Lx0
- Loads: PPO
- Reads from earlier store ("from-reads")

Lustig, Pellauer, Martonosi, “PipeCheck”
Microarchitectural Happens-Before Graphs

Lustig, Pellauer, Martonosi, “PipeCheck”
PipeCheck Overview

• Key idea: building *microarchitectural* happens-before graphs

• Inputs: arch. spec, µarch. spec, test suite
  – PPO Tests
  – Litmus Tests

• Approach: exhaustively enumerate and check all µhb graphs for cycles
  – Quick runtimes due to modest graph sizes
Microarchitectural Happens-Before Graphs

Fetch → Decode → Execute → Memory → Writeback

Memory Hierarchy

Loads

(i1) FetchStage → DecodeStage → ExecuteStage → MemoryStage → WritebackStage → StoreBuffer → MemHierarchy → Completed

(i2) FetchStage → DecodeStage → ExecuteStage → MemoryStage → WritebackStage → StoreBuffer → MemHierarchy → Completed

(i3) FetchStage → DecodeStage → ExecuteStage → MemoryStage → WritebackStage → StoreBuffer → MemHierarchy → Completed

(i4) FetchStage → DecodeStage → ExecuteStage → MemoryStage → WritebackStage → StoreBuffer → MemHierarchy → Completed

Lustig, Pellauer, Martonosi, “PipeCheck”
Microarchitectural Happens-Before Graphs

Lustig, Pellauer, Martonosi, “PipeCheck”
A Closer Look

Locations or Stages in Pipeline

- Fetch Stage
  - Fetch stage maintains relative ordering of (i1) and (i2)
- Decode Stage
  - Decode stage maintains relative ordering of (i1) and (i2)
  - Instruction (i1) flowing through pipeline
- Execute Stage
  - Execute stage maintains relative ordering of (i1) and (i2)

Instructions

(i1) → (i2)

Program Order
A Closer Look

Locations or Stages in Pipeline

Fetch Stage

Decode Stage
InSTRUCTION (i1) flowing through pipeline

Execute Stage

Instructions

(i1)
Fetch stage maintains relative ordering of (i1) and (i2)

(i2)
Decode stage maintains relative ordering of (i1) and (i2)

Out-of-Order Execute: no ordering maintained
Microarchitectural Happens-Before Graphs

Lustig, Pellauer, Martonosi, “PipeCheck”
Reading From Particular Stages

Locations or Stages in Pipeline

- Memory Stage
- Writeback Stage
- Cache/Memory

Instructions

Performs with respect to all cores

Performs with respect to remote cores

Reads From Memory

Lustig, Pellauer, Martonosi, “PipeCheck”
Reading From Particular Stages

Locations or Stages in Pipeline

Memory Stage

Writeback Stage

Cache/Memory

Instructions

Performs with respect to remote cores

Performs with respect to issuing core

Performs with respect to all cores

Reads From Store Buffer
Microarchitectural Happens-Before Graphs

Fetch → Decode → Execute → Memory → Writeback

Store Buffer → Memory Hierarchy

Writeback Stage

FetchStage
DecodeStage
ExecuteStage
MemoryStage
WritebackStage
StoreBuffer
MemHierarchy
Completed

(i1) (i2) (i3) (i4)

F

D

E

M

W

S

M

W

C

P

O

F

D

E

M

W

S

M

W

C

P

O

Lustig, Pellauer, Martonosi, “PipeCheck”
Verifying Preserved Program Order

Locations or Stages in Pipeline

- Store Buffer
- Memory Hierarchy
- Completed

Instructions

Store buffer only allows at most one outstanding request at a time

Lustig, Pellauer, Martonosi, “PipeCheck”
Verifying Preserved Program Order

Locations or Stages in Pipeline

- Store Buffer
- Memory Hierarchy
- Completed

Instructions

- St [z]
- Preserved Program Order
- St [z]
Verifying Preserved Program Order

Locations or Stages in Pipeline

- Store Buffer
- Memory Hierarchy
- Completed

Instructions

Store buffer only allows at most one outstanding request at a time

Lustig, Pellauer, Martonosi, “PipeCheck”
Verifying Preserved Program Order

Locations or Stages in Pipeline

Store Buffer

Memory Hierarchy

Completed

Instructions

St [z]

Preserved Program Order

Store buffer only allows at most one outstanding request at a time

St [z]
Microarchitectural Optimizations

PipeCheck also supports:

- OoO cores
- Speculative load reordering
- WeeFence
- Heterogeneity
- and so on
PipeCheck Software Implementation

- Tool written in Coq and extracted to OCaml
- PipeCheck software verifies each pipeline against suite of litmus tests and PPO tests
  - Automatically enumerate all possible executions (i.e. all possible graphs) for each pipeline model/test pair

<table>
<thead>
<tr>
<th></th>
<th>Observable</th>
<th>Not Observable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permitted</td>
<td>OK</td>
<td>OK (μarch stricter than necessary)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>Pipeline bug!!</td>
<td>OK</td>
</tr>
</tbody>
</table>
PipeCheck Pipeline Models

• Pipelines modeled by specifying:
  – list of stages
  – list of possible paths through the pipeline
  – set of “non-local edges” (details in paper)
  – list of “performing locations”

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic 5-Stage Pipeline without Store Buffer</td>
<td>37</td>
</tr>
<tr>
<td>Classic 5-Stage Pipeline with Store Buffer</td>
<td>62</td>
</tr>
<tr>
<td>gem5 O3 CPU Model</td>
<td>106</td>
</tr>
<tr>
<td>OpenSPARC T2</td>
<td>115</td>
</tr>
</tbody>
</table>
PipeCheck Litmus Test Results

<table>
<thead>
<tr>
<th>Litmus Test</th>
<th>Expected</th>
<th>5-Stage w/o St. Buf</th>
<th>5-Stage w/ St. Buf</th>
<th>gem5 O3</th>
<th>OpenSPARC T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>iwp2.1/amd1</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.2/amd2</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.3a/amd4</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.3b</td>
<td>Permit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.4/amd9</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.5/amd8</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.6</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>amd3</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>amd6</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n1</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n2</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n4</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n5</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n6</td>
<td>Permit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n7</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>rwc-unfenced</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1. Permitted results not observable: pipeline stronger than necessary

2. Forbidden results observable: found bugs in pipeline!
### PipeCheck Litmus Test Results

<table>
<thead>
<tr>
<th>Litmus Test</th>
<th>Expected</th>
<th>5-Stage w/o St. Buf</th>
<th>5-Stage w/ St. Buf</th>
<th>gem5 O3</th>
<th>OpenSPARC T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>iwp2.1/amd1</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.2/amd2</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.3a/amd4</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.3b</td>
<td>Permit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.4/amd9</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.5/amd8</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iwp2.6</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>amd3</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>amd6</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n1</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n2</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n4</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n5</td>
<td>Forbid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n6</td>
<td>Permit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>n7</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>rwc-unfenced</td>
<td>Permit</td>
<td>Not obs.¹</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

#### Observations:

1. Permitted results not observable: pipeline stronger than necessary
2. Forbidden results observable: found bugs in pipeline!
Interpreting Bugs Found By PipeCheck

Initially: \([x]=[y]=0\)

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

TSO: **Forbid**: \(r_1=1\), \(r_2=0\)

Reads from earlier store ("from-reads")
PipeCheck Verification Time

![Bar chart showing Tool Runtime (sec) for different Litmus Test cases with various configurations. The configurations include 5-Stg. (no SB), 5-Stg. (w/SB), gem5 O3, and OpenSPARC T2. Each bar represents the average runtime across different tools and configurations.]
PipeCheck Conclusions

• PipeCheck verifies the memory model *implementation* against vendor spec
  – Automated tool with practical runtimes

• Opens door to systematized checking of heterogeneous mix of memory referencers: CPU, GPU, accelerators, or anything else
PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models

Daniel Lustig\(^{+}\), Michael Pellauer\(^{*}\), and Margaret Martonosi\(^{+}\)

\(^{+}\)Princeton University \hspace{1cm} \(^{*}\)Intel VSSAD

MICRO 47
PipeCheck Overview

- Microarchitectural equivalent of “happens-before” graphs
PipeCheck Overview

- **Microarchitectural equivalent** of “happens-before” graphs

Diagram showing the pipeline stages: Fetch, Decode, Execute, Memory, Writeback, Store Buffer, Memory Hierarchy, and the load operation graph with arrows indicating data flow and dependencies.
Non-SC Models: Not Every Cycle Forbids a Proposed Outcome

Litmus test iwp2.4/amd9:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st [x], 1</td>
<td>(i4) st [y], 1</td>
</tr>
<tr>
<td>(i2) ld [x] → r1</td>
<td>(i5) ld [y] → r3</td>
</tr>
<tr>
<td>(i3) ld [y] → r2</td>
<td>(i6) ld [x] → r4</td>
</tr>
</tbody>
</table>

Outcome: r1=r3=1, r2=r4=0

Observed on Hardware

[Intel, AMD, Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]
Lustig, Pellauer, Martonosi, “PipeCheck”
More Complexity = Special Cases

Litmus test iwp2.4/amd9:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st [x], 1</td>
<td>(i4) st [y], 1</td>
</tr>
<tr>
<td>(i2) ld [x] → r1</td>
<td>(i5) ld [y] → r3</td>
</tr>
<tr>
<td>(i3) ld [y] → r2</td>
<td>(i6) ld [x] → r4</td>
</tr>
</tbody>
</table>

TSO: **Allow** r1=r3=1, r2=r4=0

To fully capture all model details, need either complex nodes or complex **edges**

Architectural Analysis:

- **From-reads**
  - i1 -> i2
  - i4 -> i5
- **RFI**
  - i1 -> i3
  - i4 -> i6
- **PPO**
  - i2 -> i3
  - i5 -> i6

[Alglave, FMSD ’12, Owens et al., TPHOLs ‘09]

Lustig, Pellauer, Martonosi, “PipeCheck”
More Complexity = Special Cases

Litmus test iwp2.4/amd9:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) st [x], 1</td>
<td>(i4) st [y], 1</td>
</tr>
<tr>
<td>(i2) ld [x] → r1</td>
<td>(i5) ld [y] → r3</td>
</tr>
<tr>
<td>(i3) ld [y] → r2</td>
<td>(i6) ld [x] → r4</td>
</tr>
</tbody>
</table>

TSO: **Allow** r1=r3=1, r2=r4=0

To fully capture all model details, need either complex **nodes** or complex edges

Architectural Analysis:

From-reads

Lustig, Pellauer, Martonosi, “PipeCheck”
What is the Definition of “Correct”? 

Litmus test mp:
Initially: \([x]=[y]=0\)

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

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

Cycle implies forbidden execution: instruction can’t happen before itself

Lustig, Pellauer, Martonosi, “PipeCheck”
Insights from Microarchitecture

• How is the consistency model enforced at the microarchitecture level?

Lustig, Pellauer, Martonosi, “PipeCheck”
Insights from Microarchitecture

• How is the consistency model enforced at the microarchitecture level?

Lustig, Pellauer, Martonosi, “PipeCheck”
Insights from Microarchitecture

• How is the consistency model enforced at the microarchitecture level?

- Fetch
- Decode
- Execute
- Memory
- Writeback

Store Buffer

Memory Hierarchy

Reads from earlier ("from-reads")

Loads

PPO

Reads from

?
At \( \mu \)arch Level, Cycle = Forbidden

Satisfy reads from memory \( \rightarrow \) \( \mu \)hb cycle \( \rightarrow \) forbidden

Lustig, Pellauer, Martonosi, “PipeCheck”
At μarch Level, Cycle = Forbidden

Satisfy reads from store buffer → no μhb cycle → permitted
Supporting Heterogeneity

Different cores can have different \( \mu \)arches – no special support is needed.