
Margaret Martonosi
Princeton University

http://www.istc-cc.cmu.edu/
Students on this project:
- Dan Lustig, Princeton
  - 2013 Intel Graduate Fellow
- Caroline Trippel, Princeton

Collaboration with Dr. Michael Pellauer, Intel VSSAD, Hudson, MA

Related papers:
- “PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models” To Appear. MICRO 2014.
- Other work in submission.

Other ISTC-CC students
- Elba Garza, Tae Jun Ham, Wenhao Jia, Logan Stafman, Ozlem Bilgir Yetim, Yavuz Yetim.
Heterogeneity: HW/SW Challenges

Heterogeneous Parallelism: Widespread, clear benefits:
- Specialization $\Rightarrow$ high perf-per-watt
- On-chip, On-Device, Datacenters, Cloud...

- But programming model is weak:
  - “1 CPU + peripheral devices” is nowhere near rich enough for today’s complex processors and systems.
- Our Overall Focus: Programming models and execution optimizations for Heterogeneous CMPs.
- One Specific Issue: Memory Consistency Models
The Challenge: Memory Consistency models are as diverse as the processors employing them!
Memory Consistency Models (MCMs) set the rules regarding which loads and stores may be reordered relative to each other.

- E.g., Total Store Order (TSO) used by x86

Other architectures make different (often much weaker) ordering promises.

- E.g. GPUs make very weak ordering promises
- Even other CPU ISAs (IBM, ARM) make fairly weak promises

If “default” promise is weak, programs/compilers use fence instructions to provide ordering guarantees required by the program.
Our Research Goals

- How to verify the implementation of a memory consistency model in a given processor pipeline?

- How to dynamically translate executing code from one consistency model to another at runtime?
  - Dynamic Consistency Model Translation $\approx$ Dynamic Binary Translation

- Seamless design of memory models with “black-box” IP Blocks?

- Dynamic optimization regarding migration and optimization of resources to use?
Our Research Goals

- How to verify the implementation of a memory consistency model in a given processor pipeline?

- How to dynamically translate executing code from one consistency model to another at runtime?
  - Dynamic Consistency Model Translation \(\approx\) Dynamic Binary Translation

- Seamless design of memory models with “black-box” IP Blocks?

- Dynamic optimization regarding migration and optimization of resources to use?
Architecture-Level Analysis: Happens-before Graphs

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 [y] → r1</td>
</tr>
<tr>
<td>(i2) st [y], 2</td>
<td>(i4) ld [x] → r2</td>
</tr>
</tbody>
</table>

TSO: Forbid r1=2, r2=0

Generally: A cycle implies that execution is forbidden
(Intuition: instruction can’t happen before itself)

Analysis:

[Alglave, FMSD ‘09]
How is the consistency model enforced at the **microarchitecture level**?

- Important events are distributed in both space and time.

```
Fetch  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Writeback</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Store Buffer  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
Memory Hierarchy  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

Reads from earlier ("from-reads")
PipeCheck Verification: Overview

- **Model Specification +**
- **Automatic Analysis**

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

![Diagram showing pipeline stages and memory hierarchy with annotated edges for dependence and completion]
µ-Happens-Before Graphs

Locations or Stages in Pipeline

- Fetch Stage
- Decode Stage
- Execute Stage

Instructions

- (i1) Instruction (i1) flowing through pipeline
- (i2) Program Order
- Decode stage maintains relative ordering of (i1) and (i2)
- Execute stage maintains relative ordering of (i1) and (i2)
μ-Happens-Before Graphs

Locations or Stages in Pipeline

Memory Stage

Store Buffer

Cache/Memory

Instructions

Performs with respect to remote cores

St \([z]\) 

Performs with respect to issuing core

P

Ld \([z]\)

Performs with respect to all cores

Reads From

Performs with respect to remote cores
More Complex Cases

With a microarch-level view, special exceptions are no longer needed.
PipeCheck Implementation Summary

- Pipelines modeled by specifying:
  - list of stages
  - list of possible paths through the pipeline
  - set of “non-local edges” (details in poster)
  - 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 Implementation Summary

- Tool written in Coq and extracted to OCaml
  - Open to future formal verification
- 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>Observable</th>
<th>Not Observable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permitted</td>
<td>OK</td>
</tr>
<tr>
<td>Forbidden</td>
<td>Pipeline bug!!</td>
</tr>
</tbody>
</table>
Verification Time

![Verification Time Chart]

- 5-Stg. (no SB)
- 5-Stg. (w/SB)
- gem5 O3
- OpenSPARC T2

**Tool Runtime (sec)**

**Litmus Test**

- iwp2.3a/amd4
- iwp2.3b1
- rwc-unfenced
- n6
- amd3
- n1
- iwp2.2/amd2
- iwp2.1/amd1
- n5
- iwp2.5/amd8
- iwp2.4/amd9
- amd6/IRIW
- iwp2.6
- n7
- n2
- n4
- Geomean
## 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 observed:** Found bugs in pipeline!
Other Work...

• ArMOR: Automatic translation from one memory consistency model to another.
  ▫ Binary translation for *consistency model* differences, not just ISA differences.
  ▫ Supports agile migration from one ISA+MCM to another

• Application-Aware Data Motion optimizations.
  ▫ Communication accelerators, not just computation accelerators.

• Design Space Exploration
  ▫ Fast regression-based methods for estimating optimal hw and sw design parameter choices in heterogeneous design spaces.
Conclusions

- Heterogeneous parallelism is here and growing.
- Heterogeneity affects every aspect of our ability to specify, verify, and building performance-optimized parallel systems.
- Recent work: Specification + Automatic Analysis
  - PipeCheck: Fast, automatic verification of consistency implementations against their higher-level architectural abstractions.
  - ArMOR: Fast binary translation of executables from one consistency model to another.
- Future: CPU + accelerator, cross-data-center, ...
Acknowledgements & Collaborators

- Students on this project:
  - Dan Lustig, Princeton
  - 2013 Intel Graduate Fellow
  - Caroline Trippel, Princeton

- Collaboration with Dr. Michael Pellauer, Intel VSSAD, Hudson, MA

- Related papers:
  - “PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models” To Appear. MICRO 2014.
  - Other work in submission.

- Other ISTC-CC students
  - Elba Garza, Tae Jun Ham, Wenhao Jia, Logan Stafman, Ozlem Bilgir Yetim, Yavuz Yetim.

Margaret Martonosi
Princeton University

http://www.istc-cc.cmu.edu/