Why Memory Consistency Models Matter... And tools for analyzing and verifying them

Caroline Trippel & Yatin A. Manerkar
Princeton University
UPMARC 2018

While you wait:
1) Make sure you’ve got VirtualBox downloaded to your laptop:
https://www.virtualbox.org/wiki/Downloads
2) Make sure you have the most recent version of the Tutorial VM downloaded:
http://check.cs.princeton.edu/tutorial_vm/Check_Tools_VM.ova
   VM Password: mcmsarefun

http://check.cs.princeton.edu/tutorial.html
Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].
Memory Consistency Models

**Memory Consistency Models (MCMs)**

Specify rules and guarantees about the **ordering** and **visibility** of accesses to shared memory [Sorin et al., 2011].

---

**C11/ C++11**  
LLVM IR  
**Java Bytecode**  
JVM  
**Cuda**  
PTX  
**OpenCL**  
SPIR  
---

**x86 CPU**  
ARM CPU  
Power CPU  
**Nvidia GPU**  
**AMD GPU**  
---

**Shared Virtual Memory**
Memory Consistency Models

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].

- C11/ C++11
- Java Bytecode
- Cuda
- OpenCL
- LLVM IR
- JVM
- PTX
- SPIR
- x86 CPU
- ARM CPU
- Power CPU
- Nvidia GPU
- AMD GPU

Shared Virtual Memory
Memory Consistency Models (MCMs)

Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].
What can go wrong?

- A bug in any layer can cause a “correct” program to produce incorrect outcomes
  - Ill-specified HLL memory model
  - Incorrect HLL → ISA compilation
  - Inadequate ISA specification
  - Incorrect hardware implementation

- Benefits to verifying this stack as a whole
Goals

▪ Ultimately want to write correct and efficient concurrent programs
▪ Concurrent programs are compiled and eventually run on hardware
  • Hardware reorders instructions and state updates for performance
  • Shared memory for inter-thread communication
▪ Memory Consistency Models (MCMs): govern inter-thread communication in the presence of shared memory
  • Specified at the various layers of the hardware-software stack
  • Require precise specifications, translations between layers
▪ MCM bugs anywhere in hardware-software stack can cause a “correct” high-level language program can produce incorrect results
Our Approach Today

- Basic overview of MCMs
- Our suite of tools for MCM verification
- Hands-on verification examples
  - **PipeCheck**: Verification of a HW design w.r.t. an ISA MCM specification
  - **TriCheck**: Full-stack (HLL → Compiler → ISA → HW) MCM verification
- Provide you with a general modeling/verification approach that can be applied to other problem areas
Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
  - PipeCheck: Verifying Hardware Implementations against ISA Specs
    - Graph-based happens-before analysis of program executions on hardware
    - μspec DSL for specifying axiomatic models of hardware
  - TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, ...
Motivating Example: ARM Read-after-Read Hazard

- ARM ISA spec ambiguous regarding same-address Ld→Ld ordering:
  - ARM compilers did not insert fences
  - Some ARM implementations relax same-address Ld→Ld ordering
- C/C++ variables with atomic type require same-addr. Ld→Ld ordering
- ARM issued errata1:
  - Rewrite compilers to insert fences with performance penalties
- ARM had ordering instructions in ISA to guarantee correctness

ARM Read-Read Hazard

- ARM Cortex-A9

Microarchitecture

Hardware Implementation

ISA

Memory Model

Compilation

Software

Memory Model
ARM Read-Read Hazard

Software Memory Model
- Compilation
  - ISA Memory Model
    - Hardware Implementation
      - Microarchitecture

Which HLL(s) to support?

<table>
<thead>
<tr>
<th>C11/C++11</th>
<th>ARMv7</th>
</tr>
</thead>
<tbody>
<tr>
<td>st(rlx)</td>
<td>STR</td>
</tr>
<tr>
<td>ld(rlx)</td>
<td>LDR</td>
</tr>
<tr>
<td>ld(acq)</td>
<td>LDR; DMB</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

ARM Cortex-A9
ARM Read-Read Hazard

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>st(data, 1, rlx)</td>
<td>st(data, 2, rlx)</td>
</tr>
<tr>
<td>r1 = ld(*ptr, rlx)</td>
<td>r2 = ld(data, rlx)</td>
</tr>
</tbody>
</table>

Initial conditions: data = 0, atomic *ptr = &data

Forbidden by C11: r1 = 2, r2 = 1

<table>
<thead>
<tr>
<th>C11/C++11</th>
<th>ARMv7</th>
</tr>
</thead>
<tbody>
<tr>
<td>st(rlx)</td>
<td>STR</td>
</tr>
<tr>
<td>ld(rlx)</td>
<td>LDR</td>
</tr>
<tr>
<td>ld(acq)</td>
<td>LDR; DMB</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

How does this affect real programs?

Loading data through a pointer

ARM Cortex-A9
ARM Read-Read Hazard

Software Memory Model

Compilation

ISA Memory Model

Hardware Implementation

Microarchitecture

Initial conditions: data=0, atomic *ptr=&data
Forbidden by C11: r1=2, r2=1

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>st(data, 1, rlx)</td>
<td>st(data, 2, rlx)</td>
</tr>
<tr>
<td>r1=ld(*ptr, rlx)</td>
<td></td>
</tr>
<tr>
<td>r2=ld(data, rlx)</td>
<td></td>
</tr>
</tbody>
</table>

C11/C++11             ARMv7

- st(rlx)                      STR
- ld(rlx)                      LDR
- ld(acq)                      LDR; DMB
- ...                          ...

Naïve compilation from C11 to ARMv7

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST [data]→1</td>
<td>ST [data]→2</td>
</tr>
<tr>
<td>LD [ptr]→r0</td>
<td></td>
</tr>
<tr>
<td>LD [r0]→r1</td>
<td></td>
</tr>
<tr>
<td>LD [data]→r2</td>
<td></td>
</tr>
</tbody>
</table>
**ARM Read-Read Hazard**

- **Software Memory Model**
- **Compilation**
- **ISA Memory Model**
- **Hardware Implementation**
- **Microarchitecture**

---

**Initial conditions:** data=0, atomic *ptr=&data

Forbidden by C11: r1=2, r2=1

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>st</td>
<td>st(data,1,rlx)</td>
<td>st(data,2,rlx)</td>
</tr>
<tr>
<td>r1</td>
<td>ld(*ptr,rlx)</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>ld(data,rlx)</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>C11/C++11</th>
<th>ARMv7</th>
</tr>
</thead>
<tbody>
<tr>
<td>st(rlx)</td>
<td>STR</td>
</tr>
<tr>
<td>ld(rlx)</td>
<td>LDR</td>
</tr>
<tr>
<td>ld(acq)</td>
<td>LDR; DMB</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

---

**C0**

- ST [data] ← 1
- LD [ptr] → r0
- LD [r0] → r1
- LD [data] → r2

**C1**

- ST [data] ← 2

---

Two loads of the same address

Forbidden outcome observable on Cortex-A9

---

ARM Cortex-A9
ARM Read-after-Read Hazard Demo
Google Nexus 6 (Snapdragon 805)

```cpp
std::atomic<int> z = {0};
std::atomic<int> *y = {&z};

void thread0()
{
    z.store(1, std::memory_order_relaxed);
    int r0 = y->load(std::memory_order_relaxed);
    int r1 = z.load(std::memory_order_relaxed);
    if(r0 != r1)
        z.store(3, std::memory_order_relaxed);
}

void thread1()
{
    z.store(2, std::memory_order_relaxed);
}
```

http://check.cs.princeton.edu/tutorial_extras/SnapVideo.mov
Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
  - PipeCheck: Verifying Hardware Implementations against ISA Specs
    - Graph-based happens-before analysis of program executions on hardware
    - μspec DSL for specifying axiomatic models of hardware
  - TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, ...
Benefits of Full-Stack Verification

- Categorization & quantification of bugs in the HW-SW stack
  - Effects of ISA MCM issues on the correctness of HLL programs
  - Effects of desirable HW optimizations on ISA-HLL compatibility

- We have found real bugs:
  - RISC-V MCM draft spec
  - Compiler mappings from C11 to Power and ARMv7, leading to discovery of C11 MCM bug
Full-Stack is the Result of a Whole Line of Work

- Architecture
- Implementation

PipeCheck [Micro-47]

- Formal specifications -> Happens-before graphs
- Litmus tests for events of interest
- Check Happens-Before Graphs via Efficient SMT solvers
  - Cyclic => A->B->C->A... Can’t happen
  - Acyclic => Scenario is observable
- Fast enough for broad testing
Full Stack is the Result of a Whole Line of Work

High-Level Languages

Compiler

OS

Architecture

Implementation

RTL

ArMOR [ISCA ‘15]

RTLCheck [MICRO-50]

TriCheck [ASPLOS ‘17]

COATCheck [ASPLOS ‘16]

PipeCheck [Micro-47]

CCICheck [Micro-48]

• Formal specifications -> Happens-before graphs
• Litmus tests for events of interest
• Check Happens-Before Graphs via Efficient SMT solvers
  • Cyclic => A->B->C->A... Can’t happen
  • Acyclic => Scenario is observable
• Fast enough for broad testing
Full Stack is the Result of a Whole Line of Work

High-Level Languages

Compiler

OS

Architecture

Implementation

RTL

Tricheck [ASPLOS '17]

CoatCheck [ASPLOS '16]

PipeCheck [Micro-47]

CciCheck [Micro-48]

ArMOR [ISCA '15]

RtlCheck [MICRO-50]

So far, tools have found bugs in:
- Widely-used Research simulator
- Cache coherence paper
- IBM XL C++ compiler (fixed in v13.1.5)
- In-design commercial processors
- RISC-V ISA specification
- Compiler mapping proofs
- C++ 11 mem model
Outline

- Introduction
- Motivating Example
- Overview of Our Work
  - MCM Background & Our Approach
  - PipeCheck: Verifying Hardware Implementations against ISA Specs
    - Graph-based happens-before analysis of program executions on hardware
    - μspec DSL for specifying axiomatic models of hardware
  - TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, ...
Sequential Consistency (SC)

- Defined by [Lamport 1979], execution is the same as if:
  - (R1) Memory ops of each processor appear in program order
  - (R2) Memory ops of all processors were executed in some global sequential order

<table>
<thead>
<tr>
<th>Program</th>
<th>Legal Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>x=1</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>y=1</td>
</tr>
<tr>
<td>r1=y</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>r2=x</td>
</tr>
</tbody>
</table>

x=1   x=1   x=1   y=1   y=1   y=1   y=1
r1=y   y=1   y=1   r2=x   x=1   x=1   x=1
y=1   r1=y   r2=x   x=1   r2=x   r1=y   r1=y
r2=x   r2=x   r1=y   r1=y   r1=y   r2=x
Total Store Order (TSO)

Thread 0
x=1
r1=y

Thread 1
y=1
r2=x

<table>
<thead>
<tr>
<th>TSO PPO</th>
<th>Ld</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>St</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

First inst. (i0)

Second inst. (i1)

Store Buffer

Core A

Store Buffer

Core B

Store Buffer

Core C

Shared Virtual Memory

x=0

y=0
Total Store Order (TSO)

Thread 0
x=1
r1=y

Thread 1
y=1
r2=x

<table>
<thead>
<tr>
<th>First inst. (i0)</th>
<th>TSO PPO</th>
<th>Ld</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO PPO</td>
<td>Ld</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ld</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>St</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

First inst. (i0)

Second Inst. (i1)

Store Buffer
x=1

Core A

Core B

Core C

Store Buffer

Shared Virtual Memory

x=0

y=0
Total Store Order (TSO)

Thread 0
x=1
r1=y

Thread 1
y=1
r2=x

First inst. (i0)

<table>
<thead>
<tr>
<th>TSO PPO</th>
<th>Ld</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>St</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Second Inst. (i1)

r1=y
r2=x

Core A

Core B

Core C

Store Buffer

Shared Virtual Memory

Thread 0
x=1

Thread 1
y=1

x=0
y=0
Total Store Order (TSO)

<table>
<thead>
<tr>
<th>First inst. (i0)</th>
<th>Second Inst. (i1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO PPO</td>
<td>Ld</td>
</tr>
<tr>
<td>Ld</td>
<td>✓</td>
</tr>
<tr>
<td>St</td>
<td>✓</td>
</tr>
</tbody>
</table>

Thread 0: x=1
r1=y
r1=0

Thread 1: y=1
r2=x
Total Store Order (TSO)

<table>
<thead>
<tr>
<th>TSO PPO</th>
<th>Ld</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>St</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

First inst. (i0)  Second Inst. (i1)

Thread 0  Thread 1  
x=1   y=1  
r1=y   r2=x

r1=0  r2=0

Core A  Core B  Core C

Store Buffer  Store Buffer  Store Buffer

Shared Virtual Memory

x=1  y=1  x=0  y=0
Total Store Order (TSO)

<table>
<thead>
<tr>
<th>First inst. (i0)</th>
<th>Second Inst. (i1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO PPO</td>
<td>Ld</td>
</tr>
<tr>
<td>Ld</td>
<td>✓</td>
</tr>
<tr>
<td>St</td>
<td></td>
</tr>
</tbody>
</table>

Thread 0
- x = 1
- r1 = y = 1
- r2 = 0

Thread 1
- y = 1
- r1 = y
- r2 = x

Core A
- y = 1

Core B

Core C

Store Buffer

Shared Virtual Memory

x = 1
y = 0
Total Store Order (TSO)

<table>
<thead>
<tr>
<th>TSO PPO</th>
<th>Ld</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>St</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Thread 0
x=1
r1=y
r1=0

Thread 1
y=1
r2=x
r2=0

First inst. (i0)

Second Inst. (i1)

Shared Virtual Memory
## Total Store Order

<table>
<thead>
<tr>
<th>Program</th>
<th>Legal Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>x=1</td>
</tr>
<tr>
<td>r1=y</td>
<td>y=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r2=x</td>
<td>r1=y</td>
</tr>
<tr>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td>r1=y</td>
<td>y=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r2=x</td>
<td>r1=y</td>
</tr>
<tr>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td>r1=y</td>
<td>y=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r2=x</td>
<td>r1=y</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
<tr>
<td>x=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>x=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>x=1</td>
</tr>
</tbody>
</table>
Memory Consistency Models: Critical ISA & System Component

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 multiprocessor 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 processor systems are written from the perspective of a "processor" referring to a logical processor. For example, HyperThreading Technology is treated as a multiprocessor system):

- 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.
Memory Consistency Models: Critical ISA & System Component

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 Itanium processors support a memory-ordering model that can be further defined as "weakly-serializing," and can be characterized as follows.

In a single-processor system for memory consistency, the following principles hold:

- "Processor" refers to a single processor core in a multi-core processor system.
- "HyperThread" refers to an additional software execution thread on a single physical core.
- Processor thread A and processor thread B are considered to be in a single processor.
- Rest of the software stack expects them to be implemented correctly.

In a nutshell: MCMs are part of the ISA

- They are the spec of what value will be returned when your program does a load
- Rest of the software stack expects them to be implemented correctly.
Microarchitectural Consistency Verification

- Microarch. enforces ISA-level MCM through many small orderings
  - In-order fetch/commit
  - FIFO store buffers
  - Coherence protocol
  - ...

- Difficult to ensure that these orderings *always* enforce the required orderings

- Designs may also be complicated by optimizations (*speculative load reordering, early fence retirement, OoO execution*), or novel organization (heterogeneity)
Our approach: “microarchitecturally happens-before” (μhb) graphs

Key Idea: Model executions of programs on HW as μhb graphs

- **Nodes**: Microarchitectural events in an execution
- **Edges**: Happens-before relationships between nodes

1. Draw edges that correspond to outcome-independent orderings

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

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

Program outcome of interest: \(r1=1, r2=0, r3=1,\) and \(r4=0\)
Our approach: “microarchitecturally happens-before” (μhb) graphs

Initially: [x]=[y]=0

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

Program outcome of interest: r1=1, r2=0, r3=1, and r4=0

Key Idea: Model executions of programs on HW as μhb graphs

- **Nodes**: Microarchitectural events in an execution
- **Edges**: Happens-before relationships between nodes

2. Draw edges that correspond to outcome-dependent orderings
Our approach: “microarchitecturally happens-before” (μhb) graphs

- **F**: Microarchitectural events in an execution
- **X**: Happens-before relationships between nodes

Key Idea: Model executions of programs on HW as μhb graphs

- **Nodes**: Microarchitectural events in an execution
- **Edges**: Happens-before relationships between nodes

Program outcome of interest: 
- \( r_1 = 1 \), \( r_2 = 0 \), \( r_3 = 1 \), and \( r_4 = 0 \)

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

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>((i1)) st [x] ( \leftarrow 1)</td>
<td>((i4)) st [y] ( \leftarrow 1)</td>
</tr>
<tr>
<td>((i2)) ld [x] ( \rightarrow r_1)</td>
<td>((i5)) ld [y] ( \rightarrow r_3)</td>
</tr>
<tr>
<td>((i3)) ld [y] ( \rightarrow r_2)</td>
<td>((i6)) ld [x] ( \rightarrow r_4)</td>
</tr>
</tbody>
</table>

No cycle in graph, so program outcome is observable!
Litmus test verification

- Litmus tests – small parallel programs
  - Used to highlight memory model differences/features
  - Typically there is one non-SC outcome of interest

- Different litmus tests associated with different ISA models
  - ISA memory model often characterized by their Permitted and Forbidden non-SC litmus test outcomes
  - TSO litmus test suite, Power litmus test suite, ARM litmus test suite

- Why litmus test verification?
  - Higher performance when evaluating complex designs
  - Enables us to have a fast, iterative design process
  - Focus on verification cases most likely to exhibit bugs
Many litmus tests have been developed over the years; they have names e.g., MP, SB. Initial conditions are all 0 unless otherwise stated.

This tutorial: we use a sprinkling of established tests.

### Litmus test verification (for TSO)

**Message Passing (MP)**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W x ← 1</td>
<td>R y → 0</td>
</tr>
<tr>
<td>W y ← 1</td>
<td>R x → 0</td>
</tr>
</tbody>
</table>

**SC Outcome Permitted**

**Non-SC Outcome Forbidden**

**Store Buffering (SB)**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W x ← 1</td>
<td>W y ← 1</td>
</tr>
<tr>
<td>R y → 0</td>
<td>R x → 0</td>
</tr>
</tbody>
</table>

**Non-SC Outcome Permitted**

**SC Outcome Permitted**
Compare ISA Executions with Hardware Executions

- At the **ISA level** a litmus test outcome can be:
  - Permitted
  - Forbidden

- **Our approach:** At the **hardware level** a litmus test outcome can be:
  - Observable
  - Unobservable

<table>
<thead>
<tr>
<th></th>
<th>What ISA level analysis tells us</th>
<th>What our hardware-level analysis tells us</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Observable</td>
<td>Unobservable</td>
</tr>
<tr>
<td>Permitted</td>
<td>OK</td>
<td>OK (Stricter than necessary)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>BUG</td>
<td>OK</td>
</tr>
</tbody>
</table>
Does hardware correctly implement memory model?

Instruction level analysis

<table>
<thead>
<tr>
<th>Permitted AND Observable</th>
<th>Unobservable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observable</td>
<td>Unobservable</td>
</tr>
<tr>
<td>Permitted</td>
<td>OK</td>
</tr>
<tr>
<td>Forbidden</td>
<td>BUG</td>
</tr>
</tbody>
</table>

Our analysis

Core 0
(i1) St\[x\] \leftarrow 1
(i2) St\[y\] \leftarrow 1
Under TSO: Forbid \( r1=1, r2=0 \)

Core 1
(i3) Ld\( r1 \leftarrow \[y\] \)
(i4) Ld\( r2 \leftarrow \[x\] \)

Litmus Test

Microarchitecture

Coherence Protocol (SWMR, DVI, etc.)

Lds.
L2
L1
L1
SB
SB
WB
WB
Mem.
Mem.
Exec.
Exec.
Dec.
Dec.
Fetch
Fetch

Check Inputs: Microarchitecture Spec + Litmus Tests

Microarchitecture Specification in \( \mu \text{spec} \) DSL

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

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

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \( \text{\cap} \) ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO").

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \( \text{\cap} \)
EdgeExists ((i1, Fetch), (i2, Fetch)) =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").

Refer to Quick Start Guide for more information on the \( \mu \)Spec DSL and how to write axioms.

? Permitted AND Observable
OR Permitted AND Unobservable
OR Forbidden AND Unobservable
Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /
\ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO").

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /
\ EdgeExists ((i1, Fetch), (i2, Fetch)) =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").

Check Inputs: Microarchitecture Spec + Litmus Tests
Microarchitecture Specification in μSpec DSL

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

Under TSO: Forbid r1=1, r2=0

Litmus Test

Exhaustive enumeration of all possible executions

Microarchitectural happens-before (µhb) graphs
In a nutshell, our tool philosophy...

- Automate specification, verification, and translation related to MCMs
- Comprehensive exploration of ordering possibilities
- Key Techniques: Happens-before Graphs and SMT solvers
- Bounded, litmus-test driven verification
  - We have other related techniques for whole-program, more comprehensive design analysis
  - And we have templates to assist in the automatic generation of “families” of litmus tests
- Verification conducted on and axiomatic model of hardware
  - We have tools for verifying this model is representative of real RTL
Outline

▪ Introduction
▪ Motivating Example
▪ Overview of Our Work
▪ MCM Background & Our Approach
  ▪ PipeCheck: Verifying Hardware Implementations against ISA Specs
    • Graph-based happens-before analysis of program executions on hardware
    • μspec DSL for specifying axiomatic models of hardware
  ▪ TriCheck: Expanding to HW/SW Stack Interface Issues
  ▪ Looking forward: Other uses of tools and techniques
    • CCICheck, COATCheck, SecurityCheck, ...
Overview

- Modelling simple microarchitectures (µarches) in µSpec
  - Give you a taste of what µSpec can model and reason about

- Begin by modelling a SC µarch
  - Partially completed µarch in VM, you will fill in remainder

- Post-coffee break, will look at expanding this SC µarch to TSO
  - Store buffers
  - Reading own write early (time permitting)
  - Fences (time permitting)
Specifying a Simple (SC) Microarch. in µSpec

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy
Specifying a Simple (SC) Microarch. in µSpec

- Core 0
  - Fetch
  - Execute
  - Writeback
  - 3-stage in-order pipelines

- Core 1
  - Fetch
  - Execute
  - Writeback

Memory Hierarchy
Specifying a Simple (SC) Microarch. in μSpec

Core 0
Fetch
Execute
Writeback

Core 1
Fetch
Execute
Writeback

Memory Hierarchy

Loads access Mem in Execute stage
Specifying a Simple (SC) Microarch. in µSpec

Stores sent to Memory after Writeback

Stores sent to Memory after Writeback

Memory Hierarchy
Specifying a Simple (SC) Microarch. in µSpec

1. Start VirtualBox VM

2. Open a Terminal

3. Partially completed SC uarch in
/home/check/pipecheck_tutorial/uarches/SC_fillable.uarch
(i.e. ~/pipecheck_tutorial/uarches/SC_fillable.uarch)
μSpec: A DSL for Specifying Microarchitectures

- Language has capabilities similar to first-order logic
  - forall, exists, AND (\(\land\)), OR (\(\lor\)), NOT (\(\neg\)), implication (\(\Rightarrow\))

- Microarchitecture spec is a set of **axioms**
  - Each axiom enforces a **partial ordering** on execution events…
  - …which correspond to individual smaller microarchitectural orderings!
  - Eg: Some axioms for pipeline stages, others for coherence…

- Job of PipeCheck is to ensure that axioms correctly work **together** to uphold the requirements of the ISA-level MCM

- Axiom writing is an iterative process
Specifying µSpec Nodes

- A node represents a particular event in a particular instruction’s execution
- Eg: \((i, \text{Fetch})\) represents the fetch stage of instruction \(i\)
- Sometimes the core of a node needs to be explicitly specified
- Eg: \((i, (0, \text{MemoryHierarchy}))\) represents \(i\) reaching the memory hierarchy, which is nominally on core 0
  - Reflects that there’s only one memory hierarchy, not one per core
Writing $\mu$Spec Axioms

- A microarchitecture spec has three components:
  - Stage identifier definitions
  - Macro definitions (optional) for axiom decomposition and reuse
  - Axiom definitions

- Axioms are comprised of FOL operators and built-in predicates

- Some predicates deal with nodes and edges
  - EdgeExists ((i1, Fetch), (i2, Fetch))
  - NodeExists ((i1, Execute))

- Other predicates represent architecture-level properties
  - SameCore <instr1> <instr2>
  - SamePhysicalAddress, SameData, IsAnyRead, ProgramOrder,...
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy
Finding Axioms

Stores go through the pipeline stages and reach memory in order

Memory Hierarchy
The Writes_Path Axiom

Axiom "Writes_Path":
for all microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
  ((i, Execute), (i, Writeback), "path");
  ((i, Writeback), (i, (0, MemoryHierarchy)), "path")].
The Writes_Path Axiom

Axiom name

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges[((i, Fetch), (i, Execute), "path");
((i, Execute), (i, Writeback), "path");
((i, Writeback), (i, (0, MemoryHierarchy)), "path")].

Memory Hierarchy
The Writes_Path Axiom

Axiom "Writes_Path":

forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path"));
((i, Execute), (i, Writeback), "path"));
((i, Writeback), (i,(0,MemoryHierarchy)),
"path")].

Microop: A single load/store op.
The Writes Path Axiom

**Axiom "Writes Path":**

For all writes (IsAnyWrite predicate)

```
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
  ((i, Execute), (i, Writeback), "path");o
  ((i, Writeback), (i,(0,MemoryHierarchy)), "path")].
```
The Writes_Path Axiom

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path")];
((i, Execute), (i, Writeback), "path");
((i, Writeback), (i,(0,MemoryHierarchy)), "path")].

Add edges from:
- Fetch to Execute
- Execute to Writeback
- Writeback to MemoryHierarchy
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy
Finding Axioms

Loads go through the pipeline stages in order

Memory Hierarchy
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Loads go through the pipeline stages in order

Reads Path Axiom is very similar to Writes Path axiom (no WB→MemHier edge)

SC_fillable.uarch, line 26
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Thread 0
i1: Store [x] ← 1
i2: Store [x] ← 2

Thread 1
i3: r1 = Load [x]
i4: r2 = Load [x]

SC Forbids: r1=2, r2=1, Mem[x] = 2

Initially, Mem[x] = 0
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

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

Fetch
Execute
Writeback
MemHier

(i1)
(i2)

(i3)
(i4)

Edges added according to Reads_Path axiom

Initially, Mem[x] = 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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

Edges added according to Writes_Path axiom

Initially, Mem[x] = 0
Finding Axioms

Core 0
- Fetch
  - Execute
  - Writeback

Core 1
- Fetch
  - Execute
  - Writeback

Memory Hierarchy
Finding Axioms

Core 0

Fetch

Execute

Writeback

All instructions on the same core go through Fetch in program order

Core 1

Execute

Writeback

Memory Hierarchy
The PO_Fetch Axiom

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").
The PO_Fetch Axiom

**Axiom** "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").

Predicates check that instrs are on the same core and in program order
The PO_Fetch Axiom

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").

Add edge from Fetch stage of earlier instruction \(i_1\) to Fetch stage of later instruction \(i_2\)
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

- **Fetch**
  - (i1)
  - (i2)

- **Execute**
  - (i1)
  - (i2)

- **Writeback**
  - (i1)
  - (i2)

- **MemHier**
  - (i1)
  - (i2)

Initially, Mem[x] = 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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms
Each column represents an instruction flowing through the pipeline.

(i1) Fetch
(i2) Fetch
(i3) Execute
(i4) Execute
(i3) Writeback
(i4) Writeback
(i3) MemHier
(i4) MemHier

Edges Added Using PO_Fetch axiom

(i1): Store \([x]\) \(\leftarrow 1\)
(i2): Store \([x]\) \(\leftarrow 2\)
(i3): \(r1 = \text{Load} \[x\]\)
(i4): \(r2 = \text{Load} \[x\]\)

SC Forbids: \(r1=2, r2=1, \text{Mem}[x] = 2\)

Initially, \(\text{Mem}[x] = 0\)
Finding Axioms

Core 0
- Fetch_{i_1}
  - Execute_{i_1}
    - Writeback

Core 1
- Fetch_{i_2}
  - Execute_{i_2}
    - Writeback

Memory Hierarchy
If two instructions on the same core go through Fetch in order, they will go through Execute in the same order.
The Execute_State_Is_In_order Axiom

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").
Axiom "Execute_stage_is_in_order":
for all microops "i1",
for all microops "i2",
\[\text{SameCore } i1 \text{ i2} \land \text{EdgeExists } ((i1, \text{Fetch}), (i2, \text{Fetch}), "") \Rightarrow \text{AddEdge } ((i1, \text{Execute}), (i2, \text{Execute}), "PPO").\]

If instructions \(i1\) and \(i2\) on same core go through Fetch in order…
The Execute _Stage_Is_In_order Axiom

Axiom "Execute_stage_is_in_order":
for all microops "i1",
for all microops "i2",
SameCore i1 i2 \ /
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").

...then they go through Execute in the same order.
Finding Axioms

Core 0

Fetch_{i_1} --> Execute

Writeback_{i_1} --> Fetch_{i_2} --> Execute

Writeback_{i_2} --> Memory Hierarchy

Core 1

Fetch --> Execute

Writeback --> Memory Hierarchy
Finding Axioms

If two instructions on the same core go through Fetch in order, they will go through Writeback in the same order.
The WritebackStageIsInOrder Axiom

If two instructions on the same core go through Fetch in order, they will go through Writeback in the same order.

Axiom "Writeback_stage_is_in_order":
forall microops "i1",
forall microops "i2",
________ i1 i2 /\
EdgeExists (((i1, _____), (i2, _____), "")) =>
AddEdge (((i1, ___________), (i2, ___________), "PPO").
The Writeback Stage Is In Order Axiom

If two instructions on the same core go through Fetch in order, they will go through Writeback in the same order.

Axiom "Writeback_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Writeback), (i2, Writeback), "PPO").
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline.

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Thread 0
i1: Store [x] ← 1
i2: Store [x] ← 2

Thread 1
i3: r1 = Load [x]
i4: r2 = Load [x]

SC Forbids: r1=2, r2=1, Mem[x] = 2

Initially, Mem[x] = 0
µhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2)

(i3) (i4)

Thread 0

Thread 1

\begin{align*}
i1: & \text{Store } [x] \leftarrow 1 \\
i2: & \text{Store } [x] \leftarrow 2 \\
i3: & r1 = \text{Load } [x] \\
i4: & r2 = \text{Load } [x]
\end{align*}

SC Forbids: r1=2, r2=1, Mem[x] = 2

Initially, Mem[x] = 0
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Edges from
Execute_stage_is_in_order &
Writeback_stage_is_in_order axioms

Initially, Mem[x] = 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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback\textsubscript{i1}

Core 1
- Fetch
- Execute
- Writeback\textsubscript{i2}

Memory Hierarchy
Finding Axioms

All writes to the same address must be totally ordered at the Memory Hierarchy. (coherence order)
Finding Axioms

All writes to the same address must be totally ordered at the Memory Hierarchy. (coherence order)

Coherence order:

i1: Store y=1

i2: Store y=2

Memory Hierarchy
The WriteSerialization Axiom

Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
( ~(SameMicroop i1 i2) \ IsAnyWrite i1
  \ IsAnyWrite i2 \ SamePhysicalAddress i1 i2 ) =>
( EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \ EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier))).

SC_fillable.uarch, line 65
The WriteSerialization Axiom

Axiom "WriteSerialization":
for all microops "i1",
for all microops "i2",

( ~(SameMicroop i1 i2) \ IsAnyWrite i1 \\ IsAnyWrite i2 \ SamePhysicalAddress i1 i2) =>
(EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \ EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).

Two different writes to the same address

(SC_fillable.uarch, line 65)
The WriteSerialization Axiom

Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
( ~SameMicroop i1 i2 \ IsAnyWrite i1 \ IsAnyWrite i2 \ SamePhysicalAddress i1 i2) =>
(EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \ EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).

Either i1 is before i2 in coherence order, OR vice-versa.
μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

<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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

**WriteSerialization axiom**

Two solutions;
Each enumerated **separately**

<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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC **Forbids**: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions; Each enumerated separately

Thread 0  Thread 1

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

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions; Each enumerated separately

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store ([x] \leftarrow 1)</td>
<td>i3: (r_1 = \text{Load } [x])</td>
</tr>
<tr>
<td>i2: Store ([x] \leftarrow 2)</td>
<td>i4: (r_2 = \text{Load } [x])</td>
</tr>
</tbody>
</table>

SC Forbids: \(r_1 = 2, r_2 = 1, \text{Mem}[x] = 2\)
μhb Graphs for co-mp Using Axioms

Write Serialization axiom

Two solutions; Each enumerated separately

PipeCheck examines all cases
Will focus on left graph for clarity

<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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms
A write must reach the Memory Hierarchy before memory instructions on the same core that are after the write in program order.

(otherwise the write could be reordered with later writes or later reads)
A write must reach the Memory Hierarchy before execution of memory instructions that are after the write in program order.

Axiom "EnforceWriteOrdering":
for all microop "w",
for all microop "i",
(________ w /\ __________ w i) =>
AddEdge ((w, (0, ______________)), (i, _____)).
A write must reach the Memory Hierarchy before execution of memory instructions that are after the write in program order.

Axiom "EnforceWriteOrdering":
forall microop "w",
forall microop "i",
(IsAnyWrite w \ ProgramOrder w i) =>
AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)).
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Initially, Mem[x] = 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 [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Thread 0
i1: Store [x] ← 1
i2: Store [x] ← 2

Thread 1
i3: r1 = Load [x]
i4: r2 = Load [x]

Initially, Mem[x] = 0

Edge added by Enforce_Write_Ordering axiom

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback\textsubscript{w}

Core 1
- Fetch
- Execute\textsubscript{i}
- Writeback

Y = 0
Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

Y = 0
If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

\[ i: \text{Load } y=0 \]

\[ Y = 0 \]
Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

w: Store y=1

i: Load y=0

Y = 1
BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
    DataFromInitialStateAtPA i /
    forall microop "w", ( (IsAnyWrite w /\ SamePhysicalAddress w i /
        ~SameMicroop i w) => AddEdge ((i, _______), (w, (0, _______________))) ).

if a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

$w$: Store $y=1$

$i$: Load $y=0$

$Y = 0$
BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":

DataFromInitialStateAtPA i \( \forall \) microop "w", (IsAnyWrite w \( \forall \) SamePhysicalAddress w i \( \forall \) ~SameMicroop i w) =>
AddEdge ((i, _______), (w, (0, __________________)))).

Macro: A µSpec fragment that can be instantiated as part of a larger axiom
BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":

DataFromInitialStateAtPA i \/
forall microop "w", (IsAnyWrite w \/ SamePhysicalAddress w i \/
~SameMicroop i w) =>
AddEdge ((i, _______), (w, (0, _______________))).

Check that i reads its value from the initial state of the litmus test

SCfillable.uarch, line 107
BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
DataFromInitialStateAtPA i \/
forall microop "w", ( 
(IsAnyWrite w \/
SamePhysicalAddress w i \/
~SameMicroop i w) =>
AddEdge ((i, _______), (w, (0, _______________))))).

If a load reads the initial value of a memory location, it must execute before any write to that addr reaches Mem.
BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
DataFromInitialStateAtPA i \ /
forall microop "w", ( (IsAnyWrite w \ SamePhysicalAddress w i \\
\ ~SameMicroop i w) => AddEdge ((i, Execute), (w, (0, MemoryHierarchy))))).

If a load reads the initial value of a memory location, it must execute before any write to that addr reaches Mem.
DefineMacro "BeforeAllWrites":
DataFromInitialStateAtPA i \ 
forall microop "w", ( 
(IsAnyWrite w \ SamePhysicalAddress w i \\ ~SameMicroop i w) => 
AddEdge ((i, Execute), (w, (0, MemoryHierarchy)))).

If a load reads the initial value of a memory location, it must execute before any write to that addr reaches Mem.

Enforce that the load executes before all writes to its address in the test.
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback\(_w\)

Core 1
- Fetch
- Execute\(_i\)
- Writeback

Memory Hierarchy
A load must execute either before or after any write to its address reaches memory.
Finding Axioms

A load must execute either before or after any write to its address reaches memory.
A load must execute either before or after any write to its address reaches memory.
The Before Or After Every SameAddrWrite Macro

A load must execute either before or after any write to its address.

DefineMacro "Before Or After Every SameAddrWrite":
forall microop "w", (  
(IsAnyWrite w /\ SamePhysicalAddress w i) =>  
(AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \/
 AddEdge ((i, Execute), (w, (0, MemoryHierarchy))))).
DefineMacro "Before_Or_After_Every_SameAddrWrite":
forall microop "w", (IsAnyWrite w \ SamePhysicalAddress w i) =>
(AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \/
AddEdge ((i, Execute), (w, (0, MemoryHierarchy)))).
Finding Axioms

Y = 0
Finding Axioms

A load must read from the latest write to that address to reach memory.
Alternatively:
1) The load must execute after the write it reads from
2) No writes to that address between the source write and the read

\[ Y = 0 \]
Alternatively:
1) The load must execute after the write it reads from
2) No writes to that address between the source write and the read
Finding Axioms

Alternatively:
1) The load must **execute after the write it reads from**
2) No writes to that address between the source write and the read

\[ w: \text{St } y=1 \]
\[ i: \text{Load } y=1 \]

\[ Y = 1 \]
Alternatively:
1) The load must **execute after the write it reads from**
2) No writes to that address **between the source write and the read**
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":  
exists microop "w", ( 
   IsAnyWrite w \ /
   AddEdge ((w, (0, MemHier)), (i, Execute)) \/
   ~(exists microop "w'", 
      IsAnyWrite w' \ /
      EdgesExist [((w, (0, MemHier)), (w', (0, MemHier))); 
                        ((w', (0, MemHier)), (i, Execute))])).

1) The load must **execute after the write it reads from**
2) **No writes to that address between the source write and the read**
The No_SameAddrWrites_Btwn_Src_And_Read Macro

```c
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
exists microop "w", (  
    IsAnyWrite w \ SamePhysicalAddress w i \SameData w i \ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \ ~(exists microop "w'", 
    IsAnyWrite w' \ SamePhysicalAddress i w' \ ~SameMicroop w w' \
    \ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier))); ((w', (0,MemHier)), (i, Execute))])].
```
The No_SameAddrWrites_Btwn_Src_And_Read Macro

DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
exists microop "w", (IsAnyWrite w \ SamePhysicalAddress w i \ SameData w i \ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \ ~(exists microop "w'", IsAnyWrite w' \ SamePhysicalAddress i w' \ ~SameMicroop w w' \ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier))); ((w', (0,MemHier)), (i, Execute))]]).
The No_SameAddrWrites_Btwn_Src_And_Read Macro

Alternatively:

DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read": exists microop "w", ( 
    IsAnyWrite w \ SamePhysicalAddress w i \ SameData w i \
    \ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \ 
~(exists microop "w'", 
    IsAnyWrite w' \ SamePhysicalAddress i w' \ 
~SameMicroop w w' \
    \ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier))); 
                    ((w', (0,MemHier)), (i, Execute))]]).

...and there are no writes \ w' to that addr between the source write w and the read i.
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites /
 ( ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read /
   ExpandMacro Before_Or_After_Every_SameAddrWrite )).
Putting the Macros together: the Read_Values axiom

Alternatively:

Axiom "Read Values":

For all reads \( i \) (same identifier used in the macros)...

\[
\forall \text{microops } i, \text{IsAnyRead } i \Rightarrow \\
(\text{ExpandMacro} \text{ BeforeAllWrites} \parallel \\
(\text{ExpandMacro} \text{ No_SameAddrWrites_Btwn_Src_And_Read} \\
\parallel \\
\text{ExpandMacro} \text{ Before_Or_After_Every_SameAddrWrite})).
\]
Putting the Macros together: the Read_Values axiom

Alternatively:

Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read \/
    ExpandMacro Before_Or_After_Every_SameAddrWrite )))

…either the read executes before all writes (expand macro defined earlier)…

SC_fillable.uarch, line 149
Putting the Macros together: the Read\_Values axiom

Alternatively:

Axiom "Read\_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites /
 (ExpandMacro No\_SameAddrWrites\_Btwn\_Src\_And\_Read /
 ExpandMacro Before\_Or\_After\_Every\_SameAddrWrite)
));

...or the read reads from the latest write to that address

SC\_fillable\_uarch, line 149
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Thread 0
- i1: Store [x] ← 1
- i2: Store [x] ← 2

Thread 1
- i3: r1 = Load [x]
- i4: r2 = Load [x]

Initially, Mem[x] = 0

SC Forbids: r1=2, r2=1, Mem[x] = 2
µhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline.

- **Fetch**
- **Execute**
- **Writeback**
- **MemHier**

### Thread 0
- i1: Store [x] ← 1
- i2: Store [x] ← 2

### Thread 1
- i3: r1 = Load [x]
- i4: r2 = Load [x]

**SC Forbids:** r1=2, r2=1, Mem[x] = 2

Initially, Mem[x] = 0

- i3 must be sourced from the write i2
- No intervening writes; **constraint satisfied**
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

\[
\begin{align*}
(i1) & \quad (i2) \\
\text{Fetch} & \quad \text{Execute} & \quad \text{Writeback} & \quad \text{MemHier} \\
\text{Thread 0} & \quad \text{Thread 1} \\
\text{i1: Store \([x] \leftarrow 1)} & \quad \text{i3: } r1 = \text{Load \([x])} \\
\text{i2: Store \([x] \leftarrow 2)} & \quad \text{i4: } r2 = \text{Load \([x])} \\
\end{align*}
\]

- i4 must be sourced from i1
- But i2 intervenes!

\[\Rightarrow \text{Constraint unsatisfiable}\]

Initially, Mem[\(x\)] = 0

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

SC Forbids: r1=2, r2=1, Mem[\(x\)] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

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

Thread 0

i1: Store [x] ← 1
i2: Store [x] ← 2

Thread 1

i3: r1 = Load [x]
i4: r2 = Load [x]

Initially, Mem[x] = 0

SC Forbids: r1=2, r2=1, Mem[x] = 2

Cannot find an **acyclic** graph that **satisfies** all constraints => **Forbidden** Execution of co-mp is **NOT observable** on μarch!
Test your completed SC uarch!

# Assuming you are in ~/pipecheck_tutorial/uarches/
$ check -i ../tests/SC_tests/co-mp.test -m SC_fillable.uarch

# If your uarch is valid, the above will create co-mp.pdf in your current directory (open pdfs from command line with evince)
# To run the solution version of the SC uarch on this test:
# (Note: this will overwrite the co-mp.pdf in your current folder)
$ check -i ../tests/SC_tests/co-mp.test -m SC.uarch -d solutions/

# If you get an error (cannot parse uarch, ps2pdf crashes, etc),
# examine your syntax or ask for help.
# If the outcome is observable (“BUG”), compare the graphs generated by the solution uarch to those of your uarch.

# To compare the uarches themselves:
$ diff SC_fillable.uarch solutions/SC.uarch
Run the entire suite of SC litmus tests!

# Assuming you are in ~/pipecheck_tutorial/uarches/
$ run_tests -v 2 -t ../tests/SC_tests/ -m SC_fillable.uarch

# The above will generate *.gv files in ~/pipecheck_tutorial/out/ for all SC tests, and output overall statistics at the end. If the count for “Buggy” is non-zero, your uarch is faulty. Look for the tests that output “BUG” to find out which tests fail.

# You can use gen_graph to convert gv files into PDFs:
$ gen_graph -i <test_gv_file>

# Compare your uarch with the solution SC uarch using diff to find discrepancies:
$ diff SC_fillable.uarch solutions/SC.uarch
Coffee Break!

After the break: Extending SC uarch. to TSO
PipeCheck Hands-On Continued:

Extending SC uarch. to TSO
Hands-on: Moving from SC to TSO

- Reads must currently wait for prior writes to reach memory
  - **EnforceWriteOrdering** axiom
  - Low performance!

- Main motivation for TSO: store buffers to hide write latency
  - Allow reads to be *reordered* with writes

- Also want to allow reads to bypass value from store buffer (before value made visible to other cores)
  - Known as “read your own write early”

- **How to model this in µSpec?**
Moving from SC to TSO

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy
Moving from SC to TSO

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy

Store Buffer

Store Buffer
Moving from SC to TSO

Memory Hierarchy

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Loads can bypass from SB

Store Buffer

Store Buffer

Memory Hierarchy
Moving from SC to TSO

Partially completed TSO uarch in /home/check/pipecheck_tutorial/uarches/TSO_fillable.uarch (i.e. ~/pipecheck_tutorial/uarches/TSO_fillable.uarch)

Some axioms remain the same from SC.uarch
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
Add StoreBuffer Stage

- “StoreBuffer” stage is between Writeback and MemoryHierarchy
- Solution:

  StageName _ "__________".
  StageName _ "MemoryHierarchy".
Add StoreBuffer Stage

- “StoreBuffer” stage is between Writeback and MemoryHierarchy
- Solution:

<table>
<thead>
<tr>
<th>StageName 3</th>
<th>&quot;StoreBuffer&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>StageName 4</td>
<td>&quot;MemoryHierarchy&quot;</td>
</tr>
</tbody>
</table>
Hands-on: Moving from SC to TSO

7 changes needed to SC.uarch:

1. Add store buffer stage

2. Make writes go through SB before memory
Writes Go Through SB

- Modify **Writes_Path** axiom so stores go WB → SB → MemHier

Solution:

Axiom "Writes_Path":
for all microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
((i, Execute), (i, Writeback), "path");
((i, _________), (i, ___________), "path");
((i, ___________), (i, (0, ____________)), "path")
].
Writes Go Through SB

- Modify **Writes_Path** axiom so stores go WB $\rightarrow$ SB $\rightarrow$ MemHier

- Solution:

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path"));
    ((i, Execute), (i, Writeback), "path"));
    ((i, Writeback), (i, StoreBuffer), "path"));
    ((i, StoreBuffer), (i, (0, MemoryHierarchy)), "path")
].

*TSO_fillable.uarch, line 55*
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
Same-Core Writes Go Through SB in order

- If same-core writes go through WB in order, they should go through SB in order too

- **Hint:** Use `Writeback.stage_is_in_order` axiom as a starting point

- **Solution:**

```plaintext
Axiom "StoreBuffer.stage_is_in_order":
forall microops "i1",
forall microops "i2",
IsAnyWrite i1 \ IsAnyWrite i2 \ _______ i1 i2 =>
EdgeExists ((i1, _______), (i2, _______), "") =>
AddEdge ((i1, ___________), (i2, ___________), "PPO", "darkgreen").
```
Same-Core Writes Go Through SB in order

- If same-core writes go through WB in order, they should go through SB in order too

  **Hint:** Use `Writeback_stage_is_in_order` axiom as a starting point

- **Solution:**

  Axiom "StoreBuffer_stage_is_in_order":
  
  ```
  forall microops "i1",
  forall microops "i2",
  IsAnyWrite i1 \ IsAnyWrite i2 \ SameCore i1 i2 =>
  EdgeExists ((i1, Writeback), (i2, Writeback), "") =>
  AddEdge ((i1, StoreBuffer), (i2, StoreBuffer), "PPO", "darkgreen").
  ```
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
Same-Core Writes Reach Memory In Order

- For two same-core writes in program order, first write must reach memory before second can leave store buffer
- **Hint:** Axiom should only apply to pairs of writes!
- **Solution:**

Axiom "EnforceWriteOrdering":

\[
\begin{align*}
&\forall \text{microop } w, \forall \text{microop } w', \\
&(\text{IsAnyWrite } w \land \text{__________ } w' \land \text{__________ } w \land w') \Rightarrow \\
&\text{AddEdge } ((w, (0, \text{__________})), (w', \text{__________}), \\
&\text{"one_at_a_time", "green"}).
\end{align*}
\]
For two same-core writes in program order, first write must reach memory before second can leave store buffer

Hint: Axiom should only apply to pairs of writes!

Solution:

Axiom "EnforceWriteOrdering":
forall microop "w",
forall microop "w'",
(IsAnyWrite w \ IsAnyWrite w' \ ProgramOrder w w') =>
   AddEdge (w, (0, MemoryHierarchy)), (w', StoreBuffer),
   "one_at_a_time", "green").
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
Create a **macro** enforcing that all writes before instr "i" in program order to address of "i" have reached mem before "i" **Executes**

**Solution:**

```plaintext
DefineMacro "STBEmpty":
% Store buffer is empty for the address we want to read.
forall microop "w", ( (__________ w \ w i / __________ w i) =>
AddEdge ((w, (0, ______________)), (i, ______), "STBEmpty", "purple").
```
Only read from Mem if SB has no same addr writes

- Create a macro enforcing that all writes before instr “i” in program order to address of “i” have reached mem before “i” executes

- Solution:

```c
DefineMacro "STBEmpty":
  % Store buffer is empty for the address we want to read.
  forall microop "w", (
    (IsAnyWrite w \ SamePhysicalAddress w i \ ProgramOrder w i) =>
    AddEdge ((w, (0, MemoryHierarchy)), (i, Execute),
    "STBEmpty", "purple")){
```
Only read from Mem if SB has no same addr writes

Now expand the macro in Read_Values axiom to ensure that SB has no entries for a load’s address if it is reading from memory

TSO_fillable.uarch, line 226
Only read from Mem if SB has no same addr writes

Axiom "Read_Values":
forall microops "i",
IsAnyRead i => (
% Uncomment the commented lines if you add the (advanced) store buff forwarding.
% ExpandMacro ______ \/
% (   
  ExpandMacro ______ \/
  (   ExpandMacro BeforeAllWrites \/
  (       ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read \/
  ExpandMacro Before_Or_After_Every_SameAddrWrite
  )
  )
% )
).
Axiom "Read_Values":
forall microops "i",
IsAnyRead i => (  
% Uncomment the commented lines if you add the (advanced) store buff forwarding.  
% ExpandMacro ______ \/
% (  
    ExpandMacro STBEmpty \/
    (  
        ExpandMacro BeforeAllWrites \/
        (  
            ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read \/
            ExpandMacro Before_Or_After_Every_SameAddrWrite
        )
    )
)  
).

TSO_fillable.uarch, line 226
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. *(Advanced)* Allow a core to read value of a write from its store buffer before write is made visible to other cores
Forward Value from SB (Advanced)

- Create a **macro** that checks for a write on the same core to forward from (Execute stage -> Execute stage), and ensures the forwarding occurs **before** the write reaches memory
- Macro must also check that forwarding occurs from the latest write in program order (no intervening writes)
- Solution:
DefineMacro "STBFwd":

% Forward from the store buffer
exists microop "w", (  
    __________ w /
    _______ w i /
    ____________ w i /
    _______ w i /
    AddEdges [((w, Execute), (i, Execute), "STBFwd", "red");
      ((i, Execute), (w, (0, MemoryHierarchy)), "STBFwd", "purple")]] /

% Ensure the STB entry is the latest one.
~exists microop "w'",
    __________ w' /
    ______________ w w' /
    ______________ w w' /
    _____________ w' i.
DefineMacro "STBFwd":
% Forward from the store buffer
exists microop "w", (
    IsAnyWrite w /
    SameCore w i /
    SamePhysicalAddress w i /
    SameData w i /
    AddEdges [((w, Execute), (i, Execute), "STBFwd", "red");
        ((i, Execute), (w, (0, MemoryHierarchy)), "STBFwd",
            "purple")]) /
% Ensure the STB entry is the latest one.
~exists microop "w'",
    IsAnyWrite w' /
    SamePhysicalAddress w w' /
    ProgramOrder w w' /
    ProgramOrder w' i.
Forward Value from SB

- Expand the macro in the Read_Values axiom so that forwarding from the SB is an *alternative* choice to reading from memory.
- Remember to uncomment lines 231-232, and line 243!
- Solution:

TSO_fillable.uarch, line 226
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
( ExpandMacro ______ \ /
  ( ExpandMacro STBEmpty \/
    ( ExpandMacro BeforeAllWrites
      \/
        ( ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
          \/
            ExpandMacro Before_Or_After_Every_SameAddrWrite
          )
        )
      )
  )
).

TSO_fillable.uarch, line 226
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(
  ExpandMacro STBFwd \/
  (ExpandMacro STBEmpty \/
    (ExpandMacro BeforeAllWrites \/
      (ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read \/
        ExpandMacro Before_Or_After_Every_SameAddrWrite
      )
    )
  )
).

TSO_fillable.uarch, line 226
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. **Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform**
Fence Instruction Orders Write-Read pairs

- Add a fence instruction that flushes all prior writes in program order to memory before the fence's execute stage
- Solution:

```c
TSOfillable.uarch, line 300
```
Fence Instruction Orders Write-Read pairs

TSO_fillable.uarch, line 300

Axiom "Fence_Ordering":
for all microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch), (f, Execute), "path");
          ((f, Execute), (f, Writeback), "path")]
/
(  
  for all microops "w",
   (________ w \_\_ ___________ w f) =>
    AddEdge ((w, (0, ____________)), (f, _______),
         "fence", "orange")
).

Fence Instruction Orders Write-Read pairs

Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch), (f, Execute), "path"));
((f, Execute), (f, Writeback), "path")]
/
(
  forall microops "w",
  (IsAnyWrite w \ ProgramOrder w f) =>
  AddEdge ((w, (0, MemoryHierarchy)), (f, Execute), "fence", "orange")
).
Initially, Mem[x] = Mem[y] = 0

μhb Graph for sb On TSO μarch.

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) StoreBuffer

Thread 0
- i1: Store [x] ← 1
- i2: r1 = Load [y]

Thread 1
- i3: Store [y] ← 1
- i4: r2 = Load [x]

SC Forbids: r1=0, r2=0
Initially, \( \text{Mem}[x] = \text{Mem}[y] = 0 \)

\[ \mu hb \] Graph for \( sb \) On TSO \( \mu \)arch.

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i1: \text{Store} [x] \leftarrow 1 )</td>
<td>( i3: \text{Store} [y] \leftarrow 1 )</td>
</tr>
<tr>
<td>( i2: r1 = \text{Load} [y] )</td>
<td>( i4: r2 = \text{Load} [x] )</td>
</tr>
</tbody>
</table>

\( \text{SC} \) \text{ Forbids: } r1=0, r2=0

Dotted green edges order writes before later reads in our SC \( \mu \)arch.

These edges are not present in our TSO \( \mu \)arch!
Initially, Mem\[x\] = Mem\[y\] = 0

\[\mu h b\] Graph for sb On TSO \(\mu\)arch.

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) StoreBuffer

Thread 0
- i1: Store \[x\] \(\leftarrow\) 1
- i2: r1 = Load \[y\]

Thread 1
- i3: Store \[y\] \(\leftarrow\) 1
- i4: r2 = Load \[x\]

SC Forbids: r1=0, r2=0

Dotted green edges order writes before later reads in our SC \(\mu\)arch.

These edges are not present in our TSO \(\mu\)arch!
Initially, Mem\[x\] = Mem\[y\] = 0

\[\mu hb\] Graph for sb On TSO \(\mu\)arch.

\[
\begin{align*}
\text{Thread 0} & \\
i1: & \text{Store } [x] \leftarrow 1 \\
i2: & r1 = \text{Load } [y] \\
i3: & \text{Store } [y] \leftarrow 1 \\
i4: & r2 = \text{Load } [x] \\
\text{SC Forbids: } & r1=0, r2=0
\end{align*}
\]

Dotted green edges order writes before later reads in our SC \(\mu\)arch. These edges are not present in our TSO \(\mu\)arch!

Loads no longer need to wait for prior writes to reach memory => **acyclic graph**

sb is **observable** on TSO \(\mu\)arch!
Test your completed TSO uarch!

# Assuming you are in ~/pipecheck_tutorial/uarches/
$ check -i ../tests/TSO_tests/sb.test -m TSO_fillable.uarch

# If your uarch is valid, the above will create sb.pdf in your current directory (open pdfs from command line with evince)
# To run the solution version of the TSO uarch on this test:
# (Note: this will overwrite the sb.pdf in your current folder)
$ check -i ../tests/TSO_tests/sb.test -m TSO.uarch -d solutions/

# If you get an error (cannot parse uarch, ps2pdf crashes, etc),
# examine your syntax or ask for help.
# If the outcome is not observable (“Strict”), compare the graphs generated by the solution uarch to those of your uarch.

# To compare the uarches themselves:
$ diff TSO_fillable.uarch solutions/TSO.uarch
Run the entire suite of TSO litmus tests!

# Assuming you are in ~/pipecheck_tutorial/uarches/
$ run_tests -v 2 -t ../tests/TSO_tests/ -m TSO_fillable.uarch

# The above will generate *.gv files in ~/pipecheck_tutorial/out/
# for all TSO tests, and output overall statistics at the end. If
# the count for “Buggy” is non-zero, your uarch is faulty. Look for
# the tests that output “BUG” to find out which tests fail.

# You can use gen_graph to convert gv files into PDFs:
$ gen_graph -i <test_gv_file>

# Compare your uarch with the solution TSO uarch using diff to find
# discrepancies:
$ diff TSO_fillable.uarch solutions/TSO.uarch
Covered the basics of what PipeCheck can do…

- But there’s more!

- PipeCheck can handle heterogeneous pipelines:
Covered the basics of what PipeCheck can do…

- …and microarchitectural optimizations…

**Left:** Speculative Load Reordering

**Right:** Speculative Fence Retirement
Covered the basics of what PipeCheck can do…

- …and the methodology is extensible to other types of orderings!

**CCICheck:** Coherence orderings that affect consistency

**COATCheck:** Addr Translation/Virtual Memory orderings that affect consistency
Does the µspec model match hardware?

- **RTLCheck**: Validate that hardware supports µspec axioms!

**Axiomatic Microarch. Verification**

**Abstract nodes and happens-before edges**

**Temporal RTL Verification (SVA, etc)**

**Concrete signals and clock cycles**
PipeCheck Summary

- Fast, automated verification
- Check hardware implementation against ISA spec
- Decompose HW verification into smaller per-axiom sub-problems
  - Each axiom can then be each validated w.r.t RTL independently
- Open-Sourced:
  [https://github.com/daniellustig/coatcheck](https://github.com/daniellustig/coatcheck)

Repo from this tutorial:
[https://github.com/ymanerka/pipecheck_tutorial](https://github.com/ymanerka/pipecheck_tutorial)
Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, ...
TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA

Caroline Trippel, Yatin A. Manerkar, Daniel Lustig*, Michael Pellauer*, Margaret Martonosi

Princeton University *NVIDIA

ASPLOS 2017

http://check.cs.princeton.edu/
Why is TriCheck Necessary?

- Memory model bugs are real and problematic!
  - ARM Read-after-Read Hazard [Alglave et al. TOPLAS14]
  - RISC-V ISA draft specification was incompatible with C11
  - C11→POWER/ARMv7 “trailing-sync” compiler mapping [Batty et al. POPL ‘12]
  - C11→POWER/ARMv7 “leading-sync” compiler mapping [Lahav et al. PLDI17]

- ISAs are an important and still-fluid design point!
  - Often, ISAs designed in light of desired HW optimizations
  - ISA places some constraints on hardware and some on compiler
  - Many industry memory models are still evolving: C11, ARMv7 vs. ARMv8
  - New ISAs are designed, e.g., RISC-V CPUs, specialized accelerators

- Correctness requires cooperation of the whole stack
TriCheck Key Ideas

▪ First tool capable of full stack memory model verification
  • Any layer can introduce real bugs

▪ Litmus Tests + Auto-generators
  • Comprehensive families of tests across HLL ordering options, compiler mapping variations, ISA options

▪ Happens-before, graph-based analysis
  • Nodes are memory accesses & ordering primitives
  • Edges are event orders discerned via memory model relations

▪ Efficient top-to-bottom analysis: Runtime in seconds or minutes
  • Fast enough to find real bugs; Interactive design process
TriCheck Overview

1. Define a set of HLL → ISA compiler mappings
2. Relax the TSO μspec model to permit R→R reordering
3. Write a HLL litmus test template
4. Refine inputs if necessary/desired

**Diagram:**
- HLL litmus test templates
  - HLL Litmus Test Template
  - User-defined inputs
- Auto-generated HLL litmus tests
  - Litmus Test Generator
  - HLL Memory Model Simulator (Herd)
- TriCheck
  - HLL Memory Model
  - HLL → ISA Compiler Mappings
  - ISA μSpec Model
- Refined HLL Memory Model
- Refined HLL → ISA Compiler Mappings
- Refined ISA μSpec Model
- BUG.txt
- Strict.txt
Outline

▪ TriCheck Introduction
▪ Auto-generating HLL litmus tests
▪ User-defined TriCheck inputs
▪ Iterative ISA design example
▪ Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
▪ Ongoing Work & Conclusions

NOTE: Before running TriCheck, define the $TRICHECK_HOME environment variable and install the parallel utility:

```
sudo apt-get update
export TRICHECK_HOME=/home/check/TriCheck
sudo apt-get install parallel
```
Auto-generating HLL litmus tests

- Auto-generated TriCheck inputs
  - HLL litmus test suite from templates

HLL litmus test templates

User-defined inputs

- HLL Memory Model
- HLL → ISA Compiler Mappings
- ISA μSpec Model

HLL Litmus Test Template

Litmus Test Generator

HLL Litmus Test

Auto-generated HLL litmus tests

HLL Memory Model Simulator (Herd)

Refine inputs if necessary/desired

TriCheck

BUGS.txt
Strict.txt

Refined HLL → ISA Compiler Mappings

Refined ISA μSpec Model
Litmus test templates

- HLL is generally meant to compile/map to a variety of ISAs
  - For a given litmus test, we want to evaluate all possible HLL-level formulations and ordering options
  - Translates to evaluating a variety of compiler mapping and ISA options

- HLL litmus tests with placeholders for HLL-specific memory model ordering primitives

- E.g., C11 features the *atomic* type and allows programmers to place ordering constraints on memory accesses to *atomic* variables
  - Stores to atomic variables can be specified as *relaxed*, *release*, or *seq_cst*
  - Loads of atomic variables can be specified as *relaxed*, *acquire* *, or *seq_cst*

- Litmus test templates path: $TRICHECK_HOME/tests/templates
C <TEST>
{
[x] = 0;
[y] = 0;
}

P0 (atomic_int* y, atomic_int* x) {
    atomic_store_explicit(x,1,memory_order_<ORDER_STORE>);
    atomic_store_explicit(y,1,memory_order_<ORDER_STORE>);
}

P1 (atomic_int* y, atomic_int* x) {
    int r0 = atomic_load_explicit(y,memory_order_<ORDER_LOAD>);
    int r1 = atomic_load_explicit(x,memory_order_<ORDER_LOAD>);
}

exists (0:r0=1 / 1:r1=0)

Message Passing (MP)

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Wy</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Non-SC Outcome Forbidden
Exercise: $TRICHECK_HOME/tests/templates/sb.litmus

C <TEST>
{
[x] = 0;
[y] = 0;
}
P0 (atomic_int* y, atomic_int* x) {
    // store to x
    int r0 = // load of y
}
P1 (atomic_int* y, atomic_int* x) {
    // store to y
    int r1 = // load of x
}
exists ( )

<table>
<thead>
<tr>
<th>Store Buffering (SB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
</tr>
<tr>
<td>W x ← 1</td>
</tr>
<tr>
<td>R y ← 0</td>
</tr>
</tbody>
</table>

Non-SC Outcome Permitted
Solution: $TRICHECK\_HOME/tests/templates/sb.litmus

C <TEST>
{
[x] = 0;
[y] = 0;
}

P0 (atomic_int* y, atomic_int* x) {
    atomic_store_explicit(x,1,memory_order_<_ORDER_STORE>);
    int r0 = atomic_load_explicit(y,memory_order_<_ORDER_LOAD>);
}

P1 (atomic_int* y, atomic_int* x) {
    atomic_store_explicit(y,1,memory_order_<_ORDER_STORE>);
    int r1 = atomic_load_explicit(x,memory_order_<_ORDER_LOAD>);
}

exists (0:r0=0 /\ 1:r1=0)

Store Buffering (SB)

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W x</td>
<td>← 1</td>
<td></td>
</tr>
<tr>
<td>R y</td>
<td></td>
<td>← 0</td>
</tr>
<tr>
<td>W y</td>
<td></td>
<td>← 1</td>
</tr>
<tr>
<td>R x</td>
<td></td>
<td>← 0</td>
</tr>
</tbody>
</table>

Non-SC Outcome Permitted
Outline

▪ TriCheck Introduction
▪ Auto-generating HLL litmus tests
▪ User-defined TriCheck inputs
▪ Iterative ISA design example
▪ Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
▪ Ongoing Work & Conclusions
User-defined Inputs

- Auto-generated TriCheck inputs
  - HLL litmus test suite from templates

- User-defined TriCheck inputs
  - HLL memory model (Herd [Alglave et al. TOPLAS14])
    - C11 Herd model [Batty et al. POPL16]
  - HLL\(\rightarrow\)ISA compiler mappings
  - Hardware model (\(\mu\)spec DSL)
For this tutorial, we will use the C11 HLL memory model, written in herd syntax from [Batty et al., POPL16]

C11 herd model path: $TRICHECK_HOME/util/herd/c11 PartialSC.cat
User-defined inputs #2 & #3: ISA

- ISA is a contract between hardware and software
- Sliding lever between what is required by compiler and what is required by microarchitecture
- TriCheck represents ISA as an input through:
  - Compiler mappings
  - Hardware model
User-defined input #3: Hardware model

- Hardware model so we know primitives to use in compiler mappings

- Default TriCheck uarches path: $TRICHECK_HOME/uarches

**Exercise:** open $TRICHECK_HOME/uarches /TSO-RR.uarch

- Relax Ld-Ld order
- Enforce Ld-Ld order only for dependent operations
  - Address dependencies – affect Ld-Ld, Ld-St
  - Data dependencies – affect Ld-St
  - Control dependencies – affect Ld-Ld, Ld-St
1. Modify Execute-stage-is-in-order axiom

- Modify axiom to permit Ld-Ld reordering:

  "Execute stage is in order for all pairs of operations except two reads"

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ∼ (____________ \ ∨ ____________) \ ∧
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "darkgreen").
1. Modify `Execute_stage_is_in_order` axiom

- Modify axiom to permit Ld-Ld reordering:

  "Execute stage is in order for all pairs of operations except two reads"

```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ~ (IsAnyRead i1 \ IsAnyRead i2) \ 
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "darkgreen").
```
2. Enforce dependency order by default

- Relaxing Ld-Ld order requires axioms for address (addr) and control (ctrlisb) dependencies
  - Make use of HasDependency <addr|data|ctrl|ctrlisb> <i1> <i2> predicate
    
    "If two reads are related by a dependency of type <addr|ctrlisb>, they must execute in order"

Axiom "Addr_Read_Read_Dependencies":
for all microops "i1",
for all microops "i2",
SameCore i1 i2 \∧ \__________\∧ \___________\∧ HasDependency addr i1 i2 ⇒
AddEdge ((i1, _______), (i2, _______), "addr_rr_dependency").

Axiom "CtrlIsb_Read_Read_Dependencies":
for all microops "i1",
for all microops "i2",
SameCore i1 i2 \∧ \__________\∧ \___________\∧ HasDependency ctrlisb i1 i2 ⇒
AddEdge ((i1, _______), (i2, _______), "ctrlisb").
2. Enforce dependency order by default

- Relaxing Ld-Ld order requires axioms for address (addr) and control (ctrlisb) dependencies
  - Make use of `HasDependency <addr|data|ctrl|ctrlisb> <i1> <i2>` predicate
    
    “If two reads are related by a dependency of type <addr|ctrlisb>, they must execute in order”

```plaintext
Axiom "Addr_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ / IsAnyRead i1 \ / IsAnyRead i2 \ / HasDependency addr i1 i2 =>
AddEdge ((i1, Execute), (i2, Execute), "addr_rr_dependency").

Axiom "CtrlIsb_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ / IsAnyRead i1 \ / IsAnyRead i2 \ / HasDependency ctrlisb i1 i2 =>
AddEdge ((i1, Execute), (i2, Execute), "ctrlisb").
```

TSO-RR.uarch, line 196
Compiler mappings have been proven correct for C11 to x86-TSO

<table>
<thead>
<tr>
<th>C/C++11 Operation</th>
<th>X86-TSO implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Relaxed:</td>
<td>MOV</td>
</tr>
<tr>
<td>Load Acquire:</td>
<td>MOV</td>
</tr>
<tr>
<td>Load Seq_Cst:</td>
<td>MOV</td>
</tr>
<tr>
<td>Store Relaxed:</td>
<td>MOV</td>
</tr>
<tr>
<td>Store Release:</td>
<td>MOV</td>
</tr>
<tr>
<td>Store Seq Cst:</td>
<td>MOV, MFENCE</td>
</tr>
</tbody>
</table>

Path to compiler mappings file: $TRICHECK_HOME/util/compile.txt
This is how we would specify the C11 to TSO.uarch compiler mappings in compile.txt:

<table>
<thead>
<tr>
<th>C11/C++11 op</th>
<th>prefix;prefix</th>
<th>suffix;suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read acquire</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write release</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read seq_cst</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write seq_cst</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
</tbody>
</table>
User-defined input #2: HLL $\rightarrow$ ISA compiler mappings

- **Exercise:** Modify these mappings for our new TSO-RR.uarch that relaxes Read $\rightarrow$ Read ordering.

  - **Hint:** Load Acquire and Load Seq_Cst require Read $\rightarrow$ Read order.

<table>
<thead>
<tr>
<th>C11/C++11 op</th>
<th>prefix;prefix</th>
<th>suffix;suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read acquire</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write release</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read seq_cst</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write seq_cst</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
</tbody>
</table>
Solution: Modify these mappings for our new TSO-RR.uarch that relaxes Read→Read ordering.

• Hint: Load Acquire and Load Seq_Cst require Read→Read order.

<table>
<thead>
<tr>
<th>C11/C++11 op</th>
<th>prefix;prefix</th>
<th>suffix;suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Write relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read acquire</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
<tr>
<td>Write release</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read seq_cst</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
<tr>
<td>Write seq_cst</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
</tbody>
</table>
Outline

- TriCheck Introduction
- Auto-generating HLL litmus tests
- User-defined TriCheck inputs
- Iterative ISA design example
- Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
- Ongoing Work & Conclusions
Run TriCheck On Inputs

- cd $TRICHECK_HOME/util
- ./release-generate-tests.py --all --fences
- ./release-run-all.py --pipecheck=/home/check/pipecheck_tutorial/src

Path to litmus test generator: $TRICHECK_HOME/util/release-generate-tests.py
Path to TriCheck: $TRICHECK_HOME/util/release-run-all.py
User-defined Inputs

- Each iteration: bugs analyzed to identify cause
  - Compiler bug, hardware implementation bug, ISA bug
  - Blame may be debated
  - Blame != Fix
Create BUG.txt and Strict.txt

- cd $TRICHECK_HOME/util
- ./release-parse-results.py
- cat $TRICHECK_HOME/util/results/TSO-RR.uarch/BUG.txt

Path to TriCheck output parser: $TRICHECK_HOME/util/release-parse-results.py
Create BUG.txt and Strict.txt

- cd $TRICHECK_HOME/util
- ./release-parse-results.py
- cat $TRICHECK_HOME/util/results/TSO-RR.uarch/BUG.txt

Bugs exist, so we must refine some combination of inputs and rerun...
Analyzing a bug

- `cd $TRICHECK_HOME/util/results/TSO-RR.uarch/corr`
- `gen_graph -i corr_R_relaxed_fence_acquire_fence_W_relaxed_fence_relaxed_fence.test.gv`
- `evince corr_R_relaxed_fence_acquire_fence_W_relaxed_fence_relaxed_fence.test.pdf`

C11 requires that all same-address reads of atomic locations execute in order.
ARM Read-Read Hazard

Software Memory Model

Compilation

ISA Memory Model

Hardware Implementation

Microarchitecture

Initial conditions: data=0, atomic *ptr=&data
Forbidden by C11: r1=2, r2=1

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>st</td>
<td>(data,1,rlx)</td>
<td>(data,2,rlx)</td>
</tr>
<tr>
<td>r1</td>
<td>=ld(*ptr,rlx)</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>=ld(data,rlx)</td>
<td></td>
</tr>
</tbody>
</table>

C11/C++11      | ARMv7              |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>st(rlx)</td>
<td>STR</td>
</tr>
<tr>
<td>ld(rlx)</td>
<td>LDR</td>
</tr>
<tr>
<td>ld(acq)</td>
<td>LDR; DMB</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C0</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>[data] →1</td>
<td>[data] →2</td>
</tr>
<tr>
<td>LD</td>
<td>[ptr] →r0</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>[r0] →r1</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>[data] →r2</td>
<td></td>
</tr>
</tbody>
</table>

Two loads of the same address
Forbidden outcome observable on Cortex-A9

ARM Cortex-A9
Fixing the bug...

- ARM fixed the bug by modifying the compiler, so we’ll do the same thing here...
- Modify compiler mapping in $TRICHECK_HOME/util/compile.txt

<table>
<thead>
<tr>
<th>C/C++11 Operation</th>
<th>TSO-RR implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Relaxed:</td>
<td>Read, MMFENCE</td>
</tr>
<tr>
<td>Load Acquire:</td>
<td>Read, MMFENCE</td>
</tr>
<tr>
<td>Load Seq_Cst:</td>
<td>Read, MMFENCE</td>
</tr>
<tr>
<td>Store Relaxed:</td>
<td>Write</td>
</tr>
<tr>
<td>Store Release:</td>
<td>Write</td>
</tr>
<tr>
<td>Store Seq Cst:</td>
<td>Write, MMFENCE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C11/C++11 op</th>
<th>prefix;prefix</th>
<th>suffix;suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read relaxed</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
<tr>
<td>Write relaxed</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read acquire</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
<tr>
<td>Write release</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Read seq_cst</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
<tr>
<td>Write seq_cst</td>
<td>NA</td>
<td>MMFENCE</td>
</tr>
</tbody>
</table>
Run TriCheck On Refined Inputs

- cd $TRICHECK_HOME/util
- rm –r $TRICHECK_HOME/util/tests/ctests/*/pipecheck
- ./release-generate-tests.py --all --fences
- ./release-run-all.py --pipecheck=/home/check/pipecheck_tutorial/src
- ./release-parse-results.py
- cat $TRICHECK_HOME/util/results/TSO-RR.uarch/BUG.txt
Outline

▪ TriCheck Introduction
▪ Auto-generating HLL litmus tests
▪ User-defined TriCheck inputs
▪ Iterative ISA design example
▪ Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
▪ Ongoing Work & Conclusions
RISC-V Case Study

- Create μspec models for 7 distinct RISC-V implementation possibilities:
  - All abide by current RISC-V spec
  - Vary in preserved program order and store atomicity

- Started with stricter-than-spec microarchitecture: RISC-V Rocket Chip
  - TriCheck detects bugs: refine for correctness
  - TriCheck detects over-strictness: Performed legal (per RISC-V spec) hardware relaxations

- Impossible to compile C11 for RISC-V as originally specified

- Out of 1,701 tested C11 programs:
  - RISC-V-Base-compliant design allows 144 buggy outcomes
  - RISC-V-Base+A-compliant design allows 221 buggy outcomes
RISC-V Base: Lack of Cumulative Fences

C11 acquire/release synchronization is transitive: accesses before a release write in program order, and observed by the releasing core prior to the release write must be ordered before the release from the viewpoint of an acquire read that reads from the release write.

<table>
<thead>
<tr>
<th>Initial conditions: x=0, y=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>a: sw x1, (x5)</td>
</tr>
<tr>
<td>c: fence rw, w</td>
</tr>
<tr>
<td>d: sw x2, (x6)</td>
</tr>
</tbody>
</table>

Forbidden HLL Outcome: x1=1, x2=1, x3=1, x4=0

μSpec Model:
Variation:
Litmus test:
ISA:

C0

C1

C2

Setting flag1 causes setting flag2

FENCE[LD,ST,ST]  
if (LD flag1==1)  
ST flag1\(\leftarrow1\)

Setting flag2

FENCE[LD,LD,ST]  
if (LD flag2==1)  
ST flag2\(\leftarrow1\)

LD flag1\(\rightarrow\)test

flag1=0  Main Memory  flag2=1

RISC-V Baseline (Base)
RISC-V Base: Lack of Cumulative Fences

Base RISC-V ISA lacks cumulative fences

- Cumulative fence needed to enforce order between different-thread accesses
- Cannot fix bugs by modifying compiler

Our solution: add cumulative fences to the Base RISC-V ISA

<table>
<thead>
<tr>
<th>Initial conditions: x=0, y=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>a:</td>
</tr>
<tr>
<td>sw x1, (x5)</td>
</tr>
<tr>
<td>d:</td>
</tr>
<tr>
<td>sw x2, (x6)</td>
</tr>
<tr>
<td>g:</td>
</tr>
<tr>
<td>lw x4, (x5)</td>
</tr>
</tbody>
</table>

Forbidden HLL Outcome: x1=1, x2=1, x3=1, x4=0
More results in the paper:

- Both Base and Base+A:
  - Lack of cumulative lightweight fences
  - Lack of cumulative heavyweight fences
  - Re-ordering of same-address loads
  - No dependency ordering, but Linux port assumes it

- Base+A only:
  - Lack of cumulative releases; no acquire-release synchronization
  - No roach-motel movement

Since publishing these results, a RISC-V Memory Model Working Group was formed to design a robust MCM specification for the RISC-V ISA that meets the needs of RISC-V users and supports C11.

As of a few days ago, the new MCM proposal passed the 45 day ratification period.
During RISC-V analysis, we discovered two counter-examples while using the “proven-correct” trailing-sync mappings for compiling C11 to POWER/ARMv7

Also incorrect: the proof for the C11 to POWER/ARMv7 trailing-sync compiler mappings [Manerkar et al., CoRR ‘16]
TriCheck Conclusions

▪ Memory model design choices are complicated =>
  • Verification calls for automated analysis to comprehensively tackle subtle interplay between many diverse features.

▪ TriCheck uncovered flaws in the RISC-V memory model...
  • But more generally, TriCheck can be used on any ISA.

▪ Languages and Compilers matter too...
  • TriCheck uncovered bugs in the trailing-sync compiler mapping from C11 to POWER/ARMv7
Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, ...
COATCheck: Verifying Memory Ordering at the Hardware-OS Interface

Daniel Lustig, Geet Sethi⁺, Michael Pellauer*, Margaret Martonosi, Abhishek Bhattacharjee⁺

Princeton University ⁺ Rutgers University  * NVIDIA

ASPLOS 2016

http://check.cs.princeton.edu/
**Simple Motivating Example**

<table>
<thead>
<tr>
<th>Initially:</th>
<th>[x]=0, [y]=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>St [x] ← 1</td>
<td>St [y] ← 2</td>
</tr>
<tr>
<td>Ld [y] → r1</td>
<td>Ld [x] → r2</td>
</tr>
<tr>
<td>Proposed outcome: r1=2, r2=1</td>
<td></td>
</tr>
</tbody>
</table>

Permitted if x and y are different addresses

<table>
<thead>
<tr>
<th>Initially:</th>
<th>[x]=0, [y]=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>St PA1←1</td>
<td>St PA2←2</td>
</tr>
<tr>
<td>Ld PA1→r2</td>
<td>Ld PA1→r2</td>
</tr>
<tr>
<td>Outcome r1=2, r2=1 permitted</td>
<td></td>
</tr>
</tbody>
</table>

Forbidden if x and y are synonyms

<table>
<thead>
<tr>
<th>Initially:</th>
<th>[x]=0, [y]=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>St PA1←1</td>
<td>St PA1←2</td>
</tr>
<tr>
<td>Ld PA1→r1</td>
<td>Ld PA1→r2</td>
</tr>
<tr>
<td>Outcome r1=2, r2=1 forbidden</td>
<td></td>
</tr>
</tbody>
</table>
“Transistency Model”

- Memory ordering verification is fundamentally incomplete unless it explicitly accounts for address translation
- Superset of consistency which captures all address translation-aware sets of ordering rules
- Most prior techniques ignore the implications of virtual-to-physical address translation on memory ordering
  - E.g., synonyms, and page permission updates
- Microarchitectural events and OS behavior can affect memory ordering in ways for which standard memory model analysis can be fundamentally insufficient
Ongoing Work

- We’ve seen how memory model bugs can result in incorrect program outcomes that are intermittent/unpredictable.

- Currently, we are applying our techniques of exhaustive enumeration and checking of event orderings to other domains:
  - Security: Is a hardware design susceptible to a given class of security exploits?
    - Hardware-aware exploit program synthesis
    - We auto-synthesized programs representative of Meltdown & Spectre
    - We also synthesized 2 new exploits related to Meltdown & Spectre but distinct
  - IoT: how do we reason about many concurrently acting IoT devices?
Takeaways

- Memory consistency modes matter
  - Reliability, correctness, and portability
  - Performance
  - Security

- Intuitive “checking” through automated verification

- Move memory model verification earlier in the design processes

- Evaluate across interfaces and design boundaries
  - If interfaces are often source of bugs

- Speed of approach enables new opportunities
  - Comprehensieve and fast verification for iterative design
http://check.cs.princeton.edu/