A Memory Consistency Model For RISC-V Formally Evaluated with TriCheck

Caroline Trippel
Princeton University
November 29, 2016

Role of the Instruction Set Architecture (ISA)

- Introduced in 1964 by IBM
  - 1 set of software
  - >1 hardware implementations
- Definitive spec. of hardware as seen by software:
  - Specification of what hardware must implement
  - Target for compiler translation

**Weak PPO (e.g., ARM, POWER)**
- Fewer ordering primitives (e.g., fences/barriers) inserted by compiler

**Strong PPO (e.g., SC, TSO)**
- More ordering primitives (e.g., fences/barriers) inserted by compiler
Our Work: Memory Consistency Model Verification

Software/HLL Memory Model

Compilation

Operating System

ISA Memory Model

Microarchitectural Implementation

Hardware Memory Model

ArMOR [Lustig et al. ISCA ‘15]

PipeCheck [Lustig et al. MICRO-47]

COATCheck [Lustig et al. ASPLOS ‘16]

TriCheck [Trippel et al. ASPLOS ‘17]

CCICheck [Manerkar et al. MICRO-48]
Memory Models Bugs Observed in Practice

ARM Read-after-Read Hazard [Alglave et al. TOPLAS ‘14]
- Ambiguous ISA spec. regarding same-address Ld→Ld ordering
  - ARM compilers did not insert synchronization primitives (e.g., fences/barriers)
  - Some ARM implementations relaxed same-address Ld→Ld ordering (e.g., Cortex-A9, Snapdragon 805)
- C/C++ atomics require same-address Ld→Ld ordering
  - ARM issued errata¹: Rewrite compilers to insert fences (with performance penalties)

We’ve identified and characterized flaws in the current RISC-V memory model (i.e., the memory model defined in the current manual) [Trippel et al. ASPLOS ‘17]

Note that the modifications to fix these issues will be mostly compatible with current implementations.
Outline

- Role of Memory Models in ISAs
- **What Should We Require From the Hardware?**
- What Fences/Barriers Do We Need to Support C/C++?
- TriCheck Framework for Full-Stack Memory Model Verification
- On-Going Work & Conclusions
Sequential Consistency

• Memory models specify the allowed behavior of a multithreaded program executing with shared memory

• First 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>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>r1=y</td>
</tr>
<tr>
<td>r1=y</td>
<td>r1=y</td>
</tr>
<tr>
<td>r2=x</td>
<td>r2=x</td>
</tr>
<tr>
<td>r2=x</td>
<td>r2=x</td>
</tr>
<tr>
<td>r2=x</td>
<td>y=1</td>
</tr>
<tr>
<td>y=1</td>
<td>y=1</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>r1=y</td>
<td>r1=y</td>
</tr>
<tr>
<td>r1=y</td>
<td>r1=y</td>
</tr>
<tr>
<td>r1=y</td>
<td>r1=y</td>
</tr>
</tbody>
</table>
Two Categories of Memory Model Relaxation

**Preserved Program Order:** Defines program orderings that hardware must preserve by default

**Store Atomicity:** Defines order in which stores become visible to cores

- **Multiple-copy atomic:** E.g., monolithic memory
  - All cores see store simultaneously

- **Read-Own-Write-Early-multiple-copy atomic:** E.g., private store buffer
  - Storing core can read its own store before other cores
  - Stores made visible to all remote cores simultaneously

- **Non-multiple-copy atomic:** E.g., shared store buffer
  - Storing core can read its own store before other cores
  - Store is made visible to some remote cores before others
RISC-V Proposed Preserved Program Order and Store Atomicity

Preserved Program Order:

<table>
<thead>
<tr>
<th>Before</th>
<th>PPO</th>
<th>R(SA)</th>
<th>R(DA)</th>
<th>After</th>
<th>W(SA)</th>
<th>W(DA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>W</td>
<td>✓_L</td>
<td>✓</td>
<td>✓_N</td>
<td>✓</td>
<td>✓</td>
<td>✓_N</td>
</tr>
</tbody>
</table>

Store Atomicity:

Non-multiple-copy atomic:

- Storing core can read its own store before other cores
- Store is made visible to some remote cores before others
Effects of Non-Multiple-Copy Atomic Stores

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>st ([x] \leftarrow 1)</td>
<td>st ([y] \leftarrow 1)</td>
<td>ld (x \rightarrow [r0])</td>
<td>ld (y \rightarrow [r2])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F R, R</td>
<td>F R, R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld (y \rightarrow [r1])</td>
<td>ld (x \rightarrow [r3])</td>
</tr>
</tbody>
</table>

Non-SC Outcome: \(r0=1\), \(r1=0\), \(r2=1\), \(r3=0\)

This outcome corresponds to the case in which the stores on threads T0 and T1 arrive to threads T2 and T3 in different orders.
Why Allow Non-Multiple-Copy Atomic Stores?

- Commercial ISAs allow non-multiple-copy atomic stores (e.g. ARM, POWER)
- RISC-V is intended to be integrated with other vendor ISAs
- Potential deployment in non-multiple-copy atomic memory systems
- If sharing memory system, awareness that stores may be observed in orders that differ from other cores
Outline

• Role of Memory Models in ISAs
• What Should We Require From the Hardware?

• What Fences/Barriers Do We Need to Support C/C++?
• TriCheck Framework for Full-Stack Memory Model Verification
• On-Going Work & Conclusions
## Fences to Restore Multiple-Copy Atomicity

**Initial conditions:** x=0, y=0

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>st [x]</td>
<td>1</td>
<td></td>
<td>ld x</td>
<td>[r0]</td>
</tr>
<tr>
<td>st [y]</td>
<td>1</td>
<td></td>
<td>ld y</td>
<td>[r2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pscF RW, RW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pscF RW, RW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ld y</td>
<td>[r1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ld x</td>
<td>[r3]</td>
</tr>
</tbody>
</table>

**Non-SC Outcome:** r0=1, r1=0, r2=1, r3=0

Predecessor-/Successor- Cumulative Fence: Necessary to Restore SC for Non-Multiple-Copy Atomic Memory Systems

---

PRINCETON UNIVERSITY
Other Fences/Barriers/Ordering Primitives

• Baseline Memory Model
  • PPO requires same-address R-R order to be maintained
  • PPO requires order to be maintained between most dependent instructions
  • Predecessor-/Successor-Cumulative F RW, RW; F IO, IO; F IORW, IORW

• Baseline + Atomics Extension
  • Predecessor-Cumulative F RW, W
Outline

• Role of Memory Models in ISAs
• What Should We Require From the Hardware?
• What Fences/Barriers Do We Need to Support C/C++?
  • TriCheck Framework for Full-Stack Memory Model Verification
• On-Going Work & Conclusions
TriCheck Full-Stack Verification Framework

Suite of C/C++ Litmus Tests → Compiler Mappings from C/C++ to RISC-V → Suite of Small C/C++ Programs

C/C++ Herd Model

TriCheck compares HLL outcomes to ISA-level outcomes for a spectrum of legal ISA microarchitectures.

C/C++ Outcome Forbidden → implies → ISA Level Outcome Forbidden

ISA Level Outcome Forbidden
TriCheck Full-Stack Verification Framework

- Suite of C/C++ Litmus Tests
- Compiler Mappings from C/C++ to RISC-V
  - ISA DOES NOT ALLOW outcomes prohibited by the ISA
  - C/C++ Outcome Forbidden
  - implies
  - ISA Level Outcome Forbidden
- Suite of Small C/C++ Programs
- RISC-V Check Model
TriCheck Full-Stack Verification Framework

Suite of C/C++ Litmus Tests → C/C++ Herd Model → C/C++ Outcome Forbidden

Compiler Mappings from C/C++ to RISC-V

ISA ALLOWS outcomes prohibited by the ISA

implies

RISC-V Check Model → ISA Level Outcome Forbidden

Suite of Small C/C++ Programs
RISC-V Base: Lack of Cumulative Fences

- Base RISC-V ISA lacks cumulative fences
  - Minimally, the ISA requires a Predecessor-/Successor Cumulative F RW, RW
  - Cannot fix bugs by modifying compiler currently

Our current RISC-V proposal requires only a P-/S-Cumulative F RW, RW in the RISC-V Base ISA, and includes a weaker P-Cumulative F RW, W Fence in the Base+Atomics extension.
Outline

• Role of Memory Models in ISAs
• What Should We Require From the Hardware?
• What Fences/Barriers Do We Need to Support C/C++?
• TriCheck Framework for Full-Stack Memory Model Verification
• On-Going Work & Conclusions
On-Going Work & Conclusions

• We have formulated an English language diff. of the current spec. with our proposed changes
• Currently we are constructing a formal model in Herd [Alglave et al., TOPLAS ‘14] of our proposed memory model modifications
• Memory model design choices are complicated and involve reasoning about the subtle interplay between many diverse features
• Defining an ISA specification in light of the evaluation of a single microarchitecture is not sufficient
• TriCheck is generalizable to any ISA and uncovered/quantified flaws in the RISC-V memory mode.
ctrippel@princeton.edu
http://check.cs.princeton.edu/
RISC-V Base+A: Lack of Transitive Releases

- Base+A RISC-V ISA lacks transitive releases
  - i.e., RISC-V acquires do not synchronize with RISC-V releases as required by C/C++
  - AMO.rl and stronger AMO.aq.rl are both insufficient
  - Cannot fix bugs by modifying compiler
- Our solution: redefine release operations in the Base+A RISC-V ISA to be transitive

<table>
<thead>
<tr>
<th>Variation</th>
<th>Litmus test:</th>
<th>μSpec Model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR</td>
<td>wrc</td>
<td>riscv-curr</td>
</tr>
<tr>
<td>rWR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rWM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rMM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nWR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nMM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9like</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Test Variations:
- Base+A: 256 variations
- μSpec: 96 true, 96 false, 72 don't care
RISC-V Base+A: No Roach-Motel Movement for SC Atomics

- RISC-V SC loads and stores require both aq and rl bits set on AMOs
  - Operation has acquire and release semantics
  - Prohibits roach-motel movement
- Our solution: add an sc bit for implementing AMO.aq.sc and AMO.rl.sc instructions which are capable of implementing C/C++ SC loads and stores

<table>
<thead>
<tr>
<th>Initial conditions: x=0, y=0</th>
<th></th>
<th>Initial conditions: x=0, y=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: amoswap.w.aq.rl x1, x0, (x4)</td>
<td>b: sw x1, (x5)</td>
<td>c: amoadd.w.aq.rl x0, x2, (x5)</td>
</tr>
<tr>
<td>Allowed Non-SC Outcome: x1=1, x2=1, x3=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>μSpec Model:</th>
<th>Variation:</th>
<th>Litmus test:</th>
<th>ISA:</th>
</tr>
</thead>
<tbody>
<tr>
<td>riscv-curr</td>
<td>mp</td>
<td>riscv-curr</td>
<td>sb</td>
</tr>
</tbody>
</table>

RISC-V Baseline + Atomics (Base+A)
RISC-V Base: Same Address Ld→Ld Re-Ordering

• Base RISC-V ISA includes F R, R
  • Possible to fix bugs by modifying compiler with potential performance penalty
  • 20.3% preliminary estimate of fence insertion performance penalty for ARM
• Our solution: modify Base RISC-V memory model to require same-address Ld→Ld ordering

Initial conditions: x=0, y=0

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>sw x1, (x5)</td>
<td>c: lw x3, (x5)</td>
</tr>
<tr>
<td>b</td>
<td>sw x2, (x5)</td>
<td>d: lw x4, (x5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our current RISC-V proposal eliminates F R, R from the RISC-V Base ISA, and requires hardware to enforce same-address Ld→Ld order by default.
Re-ordering Dependent Operations

• RISC-V does not require ordering for dependent instructions
• Many commercial ISAs – x86, ARM, Power – respect dependencies
  • Can also be used as lightweight synchronization
• Explicit synchronization/fences needed when dependency ordering is required but not enforced by default, e.g., Linux
  • Macro read_barrier_depends() optionally inserts a barrier
    • Inserts a fence for Alpha, which does not respect dependencies\(^1\)
    • Inserts nothing for RISC-V, which does not respect dependencies\(^2\)
• Our solution: modify Base RISC-V memory model to require the preservation of dependency orderings.

\(^1\)Linus Torvalds et al. Linux kernel, 2016. https://github.com/torvalds/linux/blob/master/arch/alpha/include/asm/barrier.h
ctrippel@princeton.edu
http://check.cs.princeton.edu/