Expanding to HW/SW Interface Issues: From High-Level Languages Through ISAs and down to Hardware
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/
What can go wrong?
- Ill-specified HLL memory model
- Incorrect HLL → ISA compilation
- Inadequate ISA specification
- Incorrect hardware implementation

TriCheck: Full-stack verification
- Compiler mappings from HLL to ISA
- Validity of hardware implementation

Memory Models in the Hardware-Software Stack

- High-level Language (HLL) Memory Model
- Compiler mappings from HLL to ISA
- Validity of hardware implementation
- Microarchitecture
- ISA Memory Model
- Hardware Implementation
Why is TriCheck Necessary?

- Memory model bugs are real and problematic!
  - ARM Read-after-Read Hazard [Alglave et al. TOPLAS14]
  - RISC-V ISA is currently incompatible with C11
  - C11 $\rightarrow$ POWER/ARMv7 “trailing-sync” compiler mapping [Batty et al. POPL ‘12]
  - C11 $\rightarrow$ 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

Write a HLL litmus test template

Define a set of HLL → ISA compiler mappings

Relax the TSO μspec model to permit R→R reordering

Litmus Test Generator

HLL Memory Model

HLL → ISA Compiler Mappings

ISA μSpec Model

Refine inputs if necessary/desired

Refined HLL Memory Model

Refined HLL → ISA Compiler Mappings

Refined ISA μSpec Model

Auto-generated HLL litmus tests

HLL Memory Model Simulator (Herd)

HLL litmus test templates

HLL Litmus Test Template

User-defined inputs

HLL Memory Model

HLL → ISA Compiler Mappings

ISA μSpec Model

TriCheck

BUG.txt

Strict.txt

Auto-generated HLL litmus tests

HLL Litmus Test

Refine if bugs
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:

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

TriCheck

HLL Memory Model Simulator (Herd)

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)

<table>
<thead>
<tr>
<th>Message Passing (MP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>W x ← 1</td>
</tr>
<tr>
<td>W y ← 1</td>
</tr>
</tbody>
</table>

Non-SC Outcome Forbidden

Processor/Core ID
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 (  )

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>0</td>
</tr>
<tr>
<td>R y</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Non-SC Outcome Permitted
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)

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W x</td>
<td>↓ 1</td>
<td>W y</td>
</tr>
<tr>
<td>R y</td>
<td>↓ 0</td>
<td>R x</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 → ISA compiler mappings
  - Hardware model (μspec DSL)
User-defined input #1: HLL memory model

- 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/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”

<table>
<thead>
<tr>
<th>Axiom &quot;Execute_stage_is_in_order&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall microops &quot;i1&quot;,</td>
</tr>
<tr>
<td>forall microops &quot;i2&quot;,</td>
</tr>
<tr>
<td>SameCore i1 i2 (\land) (\neg) (____________ (\land)</td>
</tr>
<tr>
<td>EdgeExists ((i1, Fetch), (i2, Fetch), &quot;&quot;)) (\Rightarrow)</td>
</tr>
<tr>
<td>AddEdge ((i1, Execute), (i2, Execute), &quot;PPO&quot;,</td>
</tr>
<tr>
<td>&quot;darkgreen&quot;).</td>
</tr>
</tbody>
</table>
1. Modify \texttt{Execute\_stage\_is\_in\_order} axiom

- Modify axiom to permit Ld-Ld reordering:

  "Execute stage is in order for all pairs of operations \textit{except} two reads"

\hspace{1cm}

\texttt{Axiom "Execute\_stage\_is\_in\_order":}
\texttt{forall microops "i1",}
\texttt{forall microops "i2",}
\texttt{SameCore i1 i2 /\ \sim(\texttt{IsAnyRead }i1 \ /\ \texttt{IsAnyRead }i2) /\}
\texttt{EdgeExists ((i1, Fetch), (i2, Fetch), "" ) =>}
\texttt{AddEdge ((i1, Execute), (i2, Execute), "PPO",}
\texttt{ "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":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
\|
\|
\|
______________________ /
______________________ /
\|
\|
\|
HasDependency addr i1 i2 =>
AddEdge ((i1, _______), (i2, _______), "addr_rr_dependency").

Axiom "CtrlIsb_Read_Read_Dependencies":
forall microops "i1",
forall 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"

---

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").
User-defined input #2: HLL → ISA compiler mappings

- 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
User-defined input #2: HLL → ISA compiler mappings

- 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 → ISA compiler mappings

**Exercise:** 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>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 $\neq$ 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.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

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>

Forbidden outcome observable on Cortex-A9

Two loads of the same address

<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>

<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>
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>
</tr>
</thead>
<tbody>
<tr>
<td>Read relaxed</td>
<td>NA</td>
</tr>
<tr>
<td>Write relaxed</td>
<td>NA</td>
</tr>
<tr>
<td>Read acquire</td>
<td>NA</td>
</tr>
<tr>
<td>Write release</td>
<td>NA</td>
</tr>
<tr>
<td>Read seq_cst</td>
<td>NA</td>
</tr>
<tr>
<td>Write seq_cst</td>
<td>NA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>suffix;suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MMFENCE</td>
</tr>
<tr>
<td></td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>MMFENCE</td>
</tr>
<tr>
<td></td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>MMFENCE</td>
</tr>
<tr>
<td></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) microarchitectural relaxations

- Impossible to compile C11 for RISC-V as 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:

Setting flag1 causes setting flag2

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
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

Takeaway: Current RISC-V cannot serve as a compiler target for C11.

Next Steps: There is a RISC-V Memory Model Working Group open to members of the organization. We are a part of this working group, that is working to formalize a RISC-V memory model that meets the needs of RISC-V users and supports C11.
Evaluating Compiler Mappings with TriCheck

- 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