Full-Stack Memory Model Verification with TriCheck

Caroline Trippel and Yatin A. Manerkar
Princeton University

Daniel Lustig and Michael Pellauer
Nvidia

Margaret Martonosi
Princeton University

Memory consistency models (MCMs) govern inter-module interactions in a shared memory system and are defined at the various layers of the hardware-software stack. TriCheck is the first tool for full-stack MCM verification. Using TriCheck, we uncovered under-specifications in the draft RISC-V instruction set architecture (ISA) and identified flaws in previously "proven-correct" C11 compiler mappings.

Modern computer systems employ parallelism to achieve performance scaling at manageable power and thermal levels. At the hardware level and for many programming languages, the primary mechanism for communication is shared memory. MCMs are central to performance and correctness in shared memory systems. MCMs specify the values that can be legally returned when loads access shared memory and are defined at the various layers of the hardware-software stack. Properly designed MCMs enable programmers to synchronize and orchestrate the outcomes of concurrent code. Under-specified MCMs will result in synchronization code working incorrectly; this can in turn result in intermittent unreliability and unpredictability, as well as in difficult-to-diagnose bugs when compiled programs are executed.

Several categories of problems can arise when translating a program from a high-level language (HLL) such as C11 into correct, portable, and efficient assembly code. These include:

- ill-specified or difficult-to-support HLL requirements regarding memory ordering,
- incorrect compilation or mapping of instructions from the HLL to the target ISA,
- inadequate specification of the ISA, and
- incorrect microarchitectural implementation of the ISA.

This article describes an analysis methodology and associated tool, TriCheck, for verifying that a HLL, ISA, and microarchitectural implementation all align well on MCM requirements. TriCheck supports iteratively designing an ISA MCM that provides a precise and minimally constrained target for compiled HLL programs. Our verification methodology systematically compares permitted and forbidden language-level executions of HLL programs with their corresponding observable and unobservable ISA-level executions on microarchitectural imple-

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

mentations of the ISA in question. When any illegal executions are observable, TriCheck provides information that aids designers in determining if the cause is an incorrect compiler mapping, ISA specification, hardware implementation, or even HLL specification. As a demonstration of its power, we use TriCheck to characterize deficiencies in the original RISC-V ISA[1] memory model. We also produce two counterexamples invalidating a previously "proven-correct" compiler mapping from C11 to Power and ARMv7, leading to the discovery of a flaw in the C11 HLL memory model itself.[2]

We hope that system designers at all levels of the hardware-software stack will use our open-source TriCheck tool (publicly available at https://github.com/ctrippel/TriCheck) to develop new HLLs, ISAs, and microarchitectures with increased resilience to MCM bugs.

TRICHECK: A TOOL FOR FULL-STACK MCM VERIFICATION

TriCheck adds to the Check family of tools (see http://check.cs.princeton.edu). The Check tools feature a domain-specific language (DSL) called μspec that hardware designers can use to construct a microarchitecture specification by defining a set of "ordering axioms." These axioms are statements that provide information about the ordering behavior of individual hardware components and locations within the design (such as "the commit stage is in order" and "the store buffer is FIFO"). This specification, along with a collection of user-provided litmus tests, comprise the Check inputs. Litmus tests are small parallel programs used to showcase MCM features; MCMs are often described in terms of permitted and forbidden litmus test executions across a suite of such tests. (See the "Memory Model Features" sidebar for a litmus test example.)

Given the aforementioned inputs, the Check tools evaluate all of the possible ways in which the litmus tests could be executed on the given microarchitecture to deduce all of the observable litmus test outcomes. The set of observable outcomes is compared with the set of outcomes that are permitted by the processor architecture specification to verify whether the microarchitecture correctly implements its required MCM.

TriCheck is the first tool capable of full-stack MCM verification bridging the HLL, compiler, ISA, and microarchitecture levels. Each layer of the hardware-software stack may assume a different MCM; errors at any layer or in translating between layers can ultimately produce incorrect program results. Pre-existing verification techniques, including but not limited to prior Check tools, evaluate segments of the system stack in isolation. For example, some prove compiler mappings from an HLL to an ISA, and others prove validity of a microarchitectural implementation of an ISA. No other tool runs this top-to-bottom analysis, and TriCheck does so efficiently enough to find real bugs, as we describe at the end of this article.

Litmus Test Generation from HLL Templates

TriCheck is a litmus test-based verification framework. To get the best coverage, TriCheck must consider a variety of tests designed for their coverage. We created a litmus test generator capable of producing a suite of such tests from 1) litmus test templates containing placeholders that correspond to different types of memory or synchronization operations and 2) a set of HLL MCM primitives to insert into these placeholders.

The original conference article[3] gives an example C11 litmus test template for the Write-to-Read Causality (WRC) litmus test. C11 litmus test templates contain placeholders for memory reads, memory writes, and/or fences. Our litmus test generator produces all permutations of each test template, featuring all combinations of applicable C11 memory_order primitives: relaxed (*rlx*), acquire (*acq*), or sequentially consistent (*sc*) for reads; *rlx*, release (*rel*), or *sc* for writes; and *acq*, *rel*, or *sc* for fences. (C11 uses "atomic" to mean "memory accesses used for synchronization," not just for read-modify-write. Memory operations involving C11 atomics have "memory_order"

annotations that specify ordering requirements for said operations.) This type of litmus test synthesis enables us to verify ISA MCM functionality for all possible HLL-level interactions and synchronization scenarios for a given litmus test.

TriCheck Framework and Methodology

The ISA MCM serves as a contract between hardware and software. It defines ordering semantics of valid hardware implementations and provides ordering-enforcement mechanisms for compilers to leverage, such as fences. We identify four primary MCM-dependent system components: 1) an HLL MCM, 2) compiler mappings from the HLL to an ISA, 3) an ISA MCM, and 4) a microarchitectural implementation of the ISA.

Figure 1 illustrates the TriCheck toolflow and includes as inputs an HLL MCM, HLL→ISA compiler mappings, an implementation μspec model, and a suite of HLL litmus tests (generated from litmus test templates, like the WRC example in the original conference paper[3]). The ISA places constraints on both the compiler and the microarchitecture and is present in TriCheck through these two inputs. Given these inputs, TriCheck evaluates whether they can successfully work together to preserve MCM ordering semantics guaranteed to the programmer when HLL programs are compiled and run on target microarchitectures.

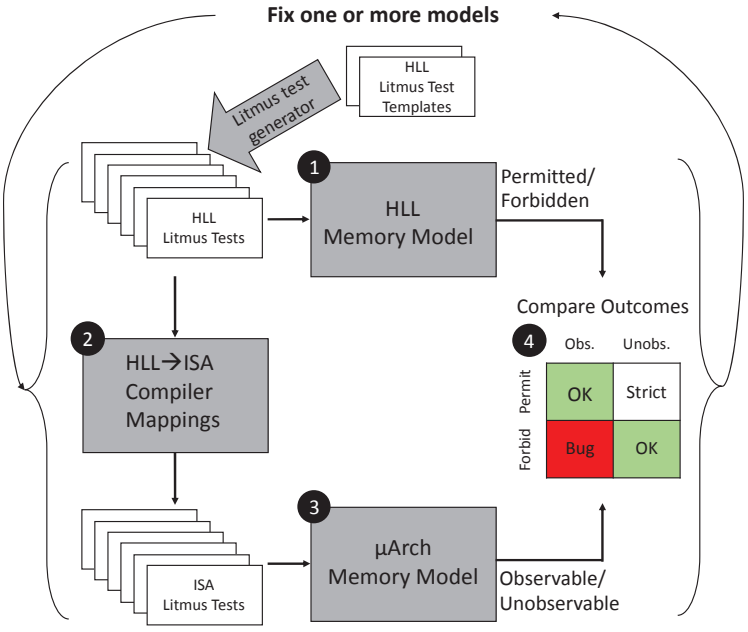

Figure 1. Overview of the TriCheck approach for full-stack MCM verification.

In describing the TriCheck toolflow, we discuss how the inputs are combined, evaluated, and refined to prohibit all illegal-according-to-HLL executions and to permit as many legal-according-to-HLL executions as possible. This results in correct but minimally constrained HLL programs. Given its inputs, the TriCheck toolflow proceeds as follows:

1. *HLL axiomatic evaluation*. The suite of HLL litmus tests is run on an HLL Herd model (such as the C11 Herd model) to determine the outcomes that the HLL MCM permits or forbids for each at the program level.
2. *HLL→ISA compilation*. Using the HLL→ISA compiler mappings, TriCheck translates HLL litmus tests to their assembly language equivalents.
3. *Microarchitecture evaluation*. The suite of assembly litmus tests is run on the Check model of the microarchitecture to determine the outcomes that are observable or unobservable on the microarchitecture.

4. *HLL-microarchitecture equivalence check*. The results of Steps 1 and 3 are compared for each test to determine if the microarchitecturally realizable outcomes are stronger than, weaker than, or equivalent to the outcomes required by the HLL model. A "stronger than" outcome corresponds to an HLL program that is permitted by the HLL MCM, yet unobservable on the microarchitectural implementation of the ISA. If Step 4 concludes that the microarchitecturally realizable outcomes are more restrictive than what the HLL requires, the designer might wish to relax the ISA or microarchitecture for performance reasons. On the other hand, when outcomes forbidden by the HLL are observable on the microarchitecture, TriCheck flags the presence of a bug that must be corrected through refinement.

After running TriCheck on a combination of inputs, a subsequent refinement step is possible. This step corresponds to modifying any combination of the HLL MCM, compiler mappings, and microarchitectural implementation in response to a microarchitectural execution that differs from the HLL-specified outcome for a given set of program executions. Discrepancies between the two outcomes result from one or more inputs containing bugs or being overly constrained. The purpose of refinement is to have a better match between HLL requirements and execution outcomes, whether this is to eliminate bugs or improve performance by avoiding overly constraining the implementation.

When TriCheck identifies an error, the cause may be debated. Regardless of where blame is assigned, designers may propose a solution that affects other layers of the hardware-software stack out of convenience or necessity. In the case outlined in the "MCM Bugs in the Wild" sidebar, the ARM ISA memory model contained a bug that permitted reordering of same-address loads, a behavior that is forbidden for C11 atomics. Due to the existence of buggy microarchitectures in the wild, ARM suggested a workaround to address the problem with compilers that stipulates that additional fence instructions should be added to executables to prevent the illegal reordering.

FINDING REAL BUGS WITH TRICHECK

Many industry MCMs are still evolving or being refined in pursuit of correctness or performance (such as C11, RISC-V, Nvidia, and ARMv8). Any changes to HLLs, compilers, ISAs, or microarchitectures merit verifying that all still align on memory model requirements. TriCheck is the first tool capable of full-stack MCM verification bridging the HLL, compiler, ISA, and microarchitecture levels.

Identifying and Characterizing Bugs in the RISC-V ISA Specification

The RISC-V ISA is an open-source ISA. A widely utilized, free, and open ISA offers some key advantages, such as well-maintained compiler and software tool-chains and even open-source hardware designs. However, a clearly defined ISA MCM is crucial in achieving this vision. To demonstrate the applicability of our framework to modern ISA, compiler, and processor design, we conducted a case study that applies TriCheck to the version of the RISC-V ISA specification available at the time the conference article[3] associated with this article was submitted.[1] The RISC-V memory model has since been corrected by a task group that was formed in part as a response to our findings.

In our case study, we used TriCheck to identify and characterize a series of deficiencies in the RISC-V v2.1 MCM specification.[1] Such issues made it possible to build legal RISC-V implementations incapable of supporting compiled C11 programs. Following the design of the RISC-V ISA, we divide our case study into two halves: one for the RISC-V memory model as it applies to the Baseline (or "Base") RISC-V ISA and one as it applies with the addition of the Atomics ISA extension (or "Base+A"). Compiler mappings from C11 to both subsets of the ISA are another key component of our study. For both the Base and Base+A ISAs, we derived these mappings manually from the RISC-V documentation.

For the microarchitecture component of our analysis, we started with a μspec model of the RISC-V Rocket Chip,[4] a six-stage in-order pipeline that supports the Base RISC-V ISA and some optional extensions, including the Atomics extension. Across a series of TriCheck runs, we explored a range of microarchitectures featuring increasing amounts of MCM relaxation, all within the bounds of what the RISC-V MCM specification permitted. For the first hardware specification input to TriCheck, we "minimally relaxed" the Rocket Chip model from a sequentially consistent (SC) design with no reordering to a design with write→read reordering enabled by adding a store buffer. This design is referred to as "μarch 1" in Figures 2 and 3.

By incrementally exploring weaker and weaker microarchitectures through gradual relaxations on our baseline design, we push the bounds of what the RISC-V MCMs allow.

We apply the iterative design and refinement methodology of Figure 1 to these inputs. When bugs are encountered, we first identify the root cause of the problem. We then propose a solution that involves modifying one or more of the TriCheck inputs and re-running TriCheck to confirm the fix is successful. As an example, Run 4 in Figures 2 and 3 produces bugs resulting from the reordering of same-address loads, a behavior that is prohibited by the C11 MCM for all atomic memory operations, yet permitted by the RISC-V v2.1 specifications for normal assembly loads. One solution to this problem could be refinement of the compiler mappings to insert fences between every pair of same-address load operations to preserve the C11-required ordering. Another solution, which is deployed in the "Refinement of μarch 4" microarchitecture of Figures 2 and 3, is to modify the ISA (and consequently the microarchitecture) to preserve the order of same-address loads by default.

By incrementally exploring weaker and weaker microarchitectures through gradual relaxations on our baseline design, we push the bounds of what the RISC-V MCMs allow. Our analysis shows that parts of the RISC-V v2.1 MCMs are too weak and others are too strong to implement C11 atomics correctly and efficiently.

Figures 2 and 3 illustrate the verification and refinement that we conducted for the Base and Base+A ISAs, respectively, highlighting bugs TriCheck discovered along the way. We evaluated seven microarchitectures, labeled μarch 1 to μarch 7. Each of these abided by the RISC-V specifications, but increasingly pushed the bounds of the RISC-V MCM specifications in terms of ordering relaxations, with some exposing correctness issues for us to address. Short descriptions of the microarchitectures are given to the left of each in the figures. (Please refer to the original paper for more details on the hardware optimizations used to create these designs.[3]) Red X's signify that bugs were produced in a given TriCheck run. Runs immediately following bug-producing runs feature some refinement of the inputs in response to the error.

In addition to the bugs highlighted in Figures 2 and 3, we observed a potential inefficiency in the Base+A ISA. Informally, the C11 memory model requires that *sc* atomic loads and stores need only enforce acquire and release orderings respectively, in addition to appearing in a total order observed by all cores. *Sc* loads do not need to implement release semantics, and *sc* stores do not need to implement acquire semantics. This allows *acq* loads and *rel* stores to function as one-way barriers, allowing more reordering of memory operations and theoretically improved performance. (This type of reordering past *sc* atomics to "expand the critical section" has been dubbed "Roach-Motel movement.") RISC-V v2.1 required *sc* loads and *sc* stores to act as two-way barriers, thereby precluding some potential performance optimizations.

In response to the flaws we identified in the earlier versions of the RISC-V ISA MCM, the RISC-V Foundation initiated a Memory Model Task Group to repair the RISC-V ISA MCM. Through that group, we are participating with other experts in specifying and formalizing a correct memory model, and a corrected MCM is (at the time of publication of this article) nearing final ratification. The original article[3] shows quantitative results from our study.

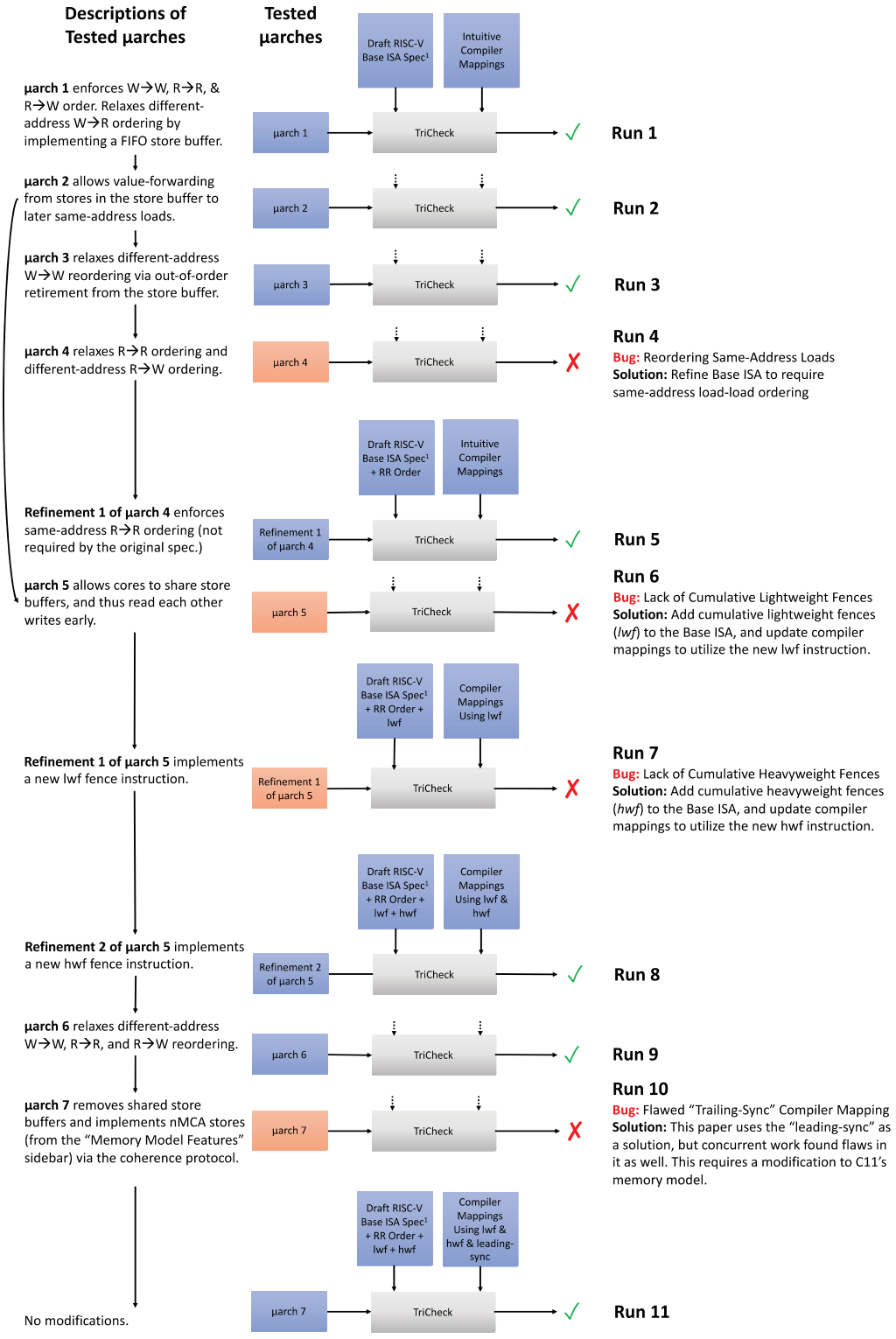

Figure 2. TriCheck verification of the Base RISC-V ISA. Arrows connecting the descriptions of two microarchitectures illustrate that all of the features of the first (tail of the arrow) are included in the second (head of the arrow), along with any new features unique to the second. Dashed (or "ghost") arrow inputs indicate that they are the same as the inputs for the previous run. See text for a discussion of Run 10's surprising result.

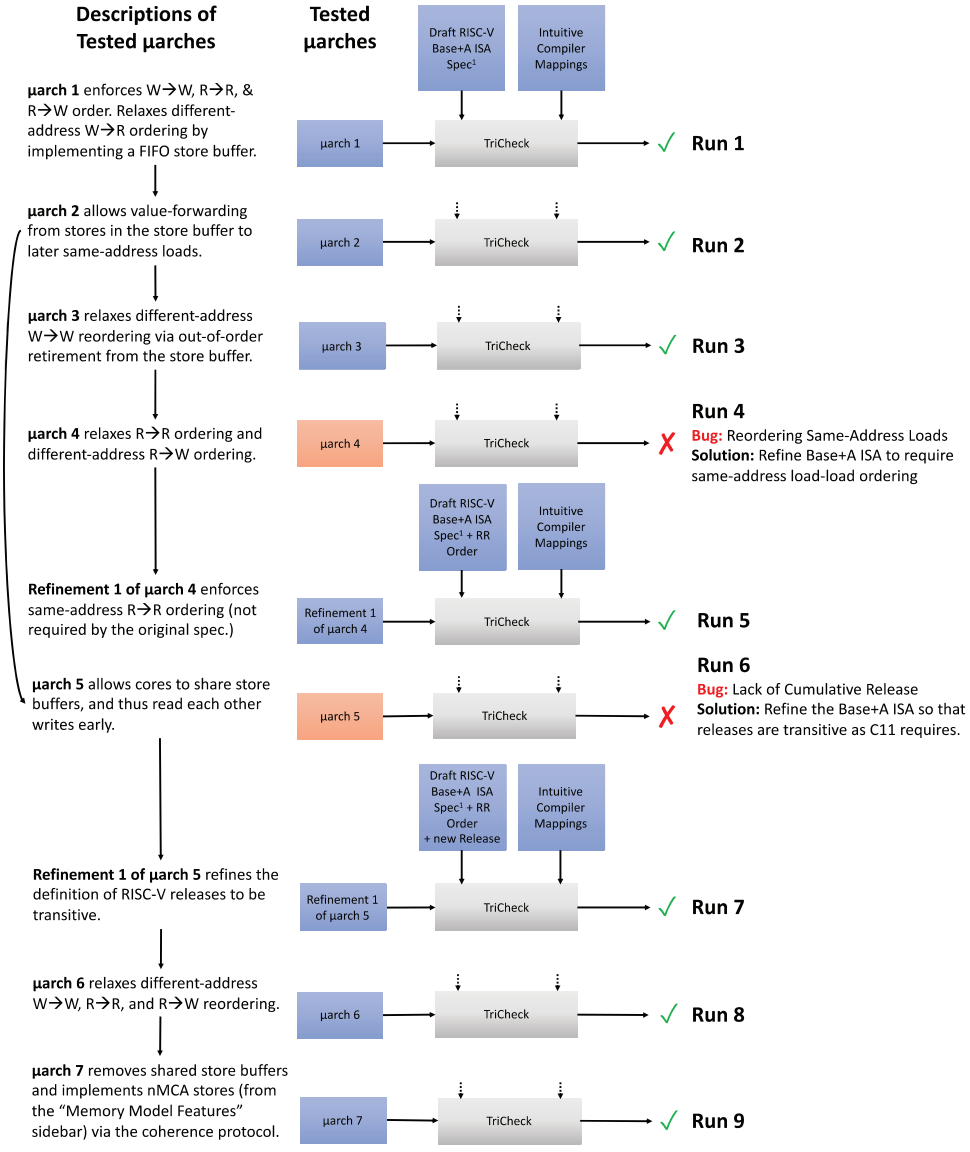

Figure 3. TriCheck verification of the Base+A ISA. Arrows connecting the descriptions of two microarchitectures illustrate that all of the features of the first (tail of the arrow) are included in the second (head of the arrow), along with any new features unique to the second. Dashed (or "ghost") arrow inputs indicate that they are the same as the inputs for the previous run.

Identifying Flaws in "Proven-Correct" Compiler Mappings from C11 to Power and ARMv7

TriCheck uses compiler mappings to get from HLLs to ISA. We intended to rely on existing proven-correct compiler mappings for our case studies. Thus, we were surprised to find that one of our first TriCheck results was to identify two counterexamples to previously proven-correct compiler mappings from C11 to Power and ARMv7; it turns out the proofs in question relied on an unproven and invalid axiom. These counterexamples are also noted in Figure 2, Run 10. This unexpected counterexample further supports our claims that full-stack verification is important and that verifying segments of the system stack in isolation can miss bugs. Our style of test generation produces comprehensive families of tests, exercising many interleavings and C11 feature

combinations. Additionally, TriCheck allows us to run a whole family of tests efficiently, considering all possible interleavings in seconds or minutes. This allowed us to identify counterexamples showcasing variants of well-known litmus tests that themselves were not well studied. These counterexamples, along with a flaw discovered in the leading-sync mapping in concurrent work,[5] render the C11 HLL memory model broken and in need of revision. As with the RISC-V work, the ISO C++ Concurrency Committee responded by making the necessary changes to the C++ MCM to correct the problem.[14]

CONCLUSION

Based on its success identifying shortcomings of the RISC-V and C++ MCMs, we argue that TriCheck is a highly efficient mechanism for full-stack memory model verification that can produce real counterexamples when they exist. As such, we envision architects using TriCheck early in the ISA or microarchitecture design process. While architects are selecting hardware optimizations for improved performance or simplifications for ease of verification, TriCheck can be used to simultaneously study the effects of these choices on the ISA-visible MCM and the ability of their designs to accurately and efficiently support HLL programs.

While architects are selecting hardware optimizations for improved performance or simplifications for ease of verification, TriCheck can be used to simultaneously study the effects of these choices on the ISA-visible MCM and the ability of their designs to accurately and efficiently support HLL programs.

TriCheck is not limited to new or evolving ISA designs. Furthermore, there are cases when other elements (such as the compiler) are modified in response to ISA or microarchitecture MCM bugs out of convenience or necessity. The "MCM Bugs in the Wild" sidebar provides such an example. When a workaround is proposed for an MCM bug—such as fence insertion in ARM's case—TriCheck can be used to verify that adding the fence did indeed prohibit the forbidden outcome across relevant litmus tests. Our RISC-V case study showcases TriCheck's applicability to ISA design by focusing on the time in the design process when the ISA MCM can be modified, as well as its applicability to HLL design by describing how TriCheck was able to find a flaw in an HLL→ISA compiler mapping for C11 and the Power and ARMv7 ISAs.

SIDEBAR: MEMORY MODEL FEATURES

Frequently regarded as the most intuitive MCM, SC[6] requires that the result of a program execution be the same as if all cores execute their own instructions in program order (PO) and a total global order exists on all instructions from all cores such that each load returns the value written by the most recent store to the same address. Unfortunately, common microarchitectural optimizations violate SC, resulting in low performance for naive SC implementations. Despite attempts at mitigating SC's performance cost, most manufacturers have nevertheless elected to build hardware with MCMs that relax SC. Various issues can arise when the effects of relaxing memory orderings are not carefully considered at ISA design time. Here we provide some examples of MCM features that are relevant to our case study and results.

Coherence, Same-Address Ordering, and Dependencies

Coherence ensures that 1) all stores are eventually made visible to all cores and 2) there exists a single total order that all threads agree on for all stores to the same address.[7,8]

A dependency relates a load with a load or store that is later in PO. An address dependency results when the address accessed by a load or store depends syntactically on the value returned by a PO-prior load. A data dependency exists between a load and a PO-later store when the store's value depends syntactically on the loaded value. A control dependency occurs when the control flow decision of whether to execute a load or store depends syntactically on the value returned by a PO-prior load.

Store Atomicity, Cumulativity, and C11 Atomics

A store is multiple-copy atomic (MCA) if all cores in the system, including the performing core, conceptually see the updated value at the same instant.[9] As a performance optimization, some architectures allow a core to read its own writes prior to their being made visible to other cores; we refer to this as read-own-write-early-multiple-copy atomic (rMCA).[10] (As a side-effect, this generally also relaxes write→read ordering.) However, rMCA writes must be made visible at the same time to all cores other than the performing core.

Weaker models, such as ARMv7 and Power, feature non-multiple-copy atomic (nMCA) stores that may become visible to some remote cores before they become visible to others. The litmus test in Figure 4 demonstrates the often counter-intuitive effects of nMCA stores. The specified non-SC outcome (the outcome that would be forbidden under the SC memory model) corresponds to a causality chain where T0 sets a flag by writing 1 to *x*, and T1 reads the updated value of *x*, subsequently setting its own flag by writing 1 to *y*. T2 then sees the update of *y*, reading 1; however, it has still not observed the update of *x* and reads its value as 0. If this C11 program is compiled down to regular assembly loads and stores on an nMCA system, the forbidden outcome will (perhaps surprisingly) be observable.

Intial conditions: x=0, y=0		
T0	T1	T2
a: st(x,1,rlx)	b: r0 = ld(x, rlx) c: st(y,1,rel)	d: r1 = ld(y, acq) e: r2 = ld(x, rlx)
Forbidden C11 Outcome: r0=1, r1=1, r2=0		

Figure 4. C11 variant of the WRC litmus test. T0, T1, and T2 are three threads. The *st* and *ld* of *y* achieve release-acquire synchronization.

An nMCA architecture must include cumulative fences to support C11-style cross-thread synchronization. Fences order specified accesses in the fence's predecessor set (accesses before the fence) with specified accesses in the fence's successor set (accesses after the fence). Cumulative fences additionally include accesses performed by threads other than the fencing thread in the predecessor and successor sets. Recursively, memory operations (from any thread) that have performed prior to an access in the predecessor set are also members of the predecessor set. Also recursively, memory operations (from any thread) that perform after a load that returns the value of a store in the successor set are also in the successor set.

SIDEBAR: MCM BUGS IN THE WILD

Mis- and under-specification of MCMs in modern hardware is a real problem that leads to processors producing incorrect or counter-intuitive outcomes. Consider the standard compilation approach today, in which compilers map each C++ operation into one or more processor instructions that perform the necessary operation and enforce the necessary memory ordering (or stronger). When the program in Figure 5 is compiled by Clang++ v3.8, the resulting program intermittently produces a result that is illegal according to the C11 specification[11] when run on some ARM hardware platforms. This behavior was first reported by J. Alglave, L. Maranget, and M. Tautschnig.[12] We have observed the phenomenon on a Galaxy Nexus (ARM Cortex-A9) and a Nexus 6 (Qualcomm Snapdragon 805).

In this particular example, the illegal outcome occurs because hardware does not preserve PO (the original thread ordering) for reads of the same address. This behavior was formally acknowledged by ARM as a bug in 2011[13] and is referred to in this article as the "ARM load→load hazard."

The ARM load→load hazard is the result of imprecision in the coherence specification. ARM acknowledged that, due to the vast number of load instructions in programs, binary patching in the linker is infeasible; they instead suggest that compilers be rewritten to issue a data memory barrier (*dmb*) fence instruction immediately following atomic loads. To evaluate the performance cost resulting from imprecise ISA MCM specifications, we conducted a measurement study in the original paper[3] to estimate the performance overhead of this proposed workaround, finding it to be approximately 15.3 percent for the tested workload.

```
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);
    assert(r0 != r1);
}

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

```
Test CO-RSDWI Allowed
Histogram (3 states)
5010437:>0:R0=0; 0:R2=1; 0:R3=1; z
4989500:>0:R0=0; 0:R2=1; 0:R3=1; z
63    *>0:R0=0; 0:R2=2; 0:R3=1; z=
Ok

Witnesses
Positive: 63, Negative: 9999937
```

Figure 5. A C11 program that intermittently produces results disallowed by the C11 MCM when compiled by Clang++v3.8 and run on modern ARM Android hardware.

ACKNOWLEDGMENTS

This work was supported in part by the Center for Future Architectures Research (C-FAR) under the grant HR0011-13-3-0002, the Semiconductor Research Corporation (SRC) STARnet program (sponsored by Microelectronics Advanced Research Corporation (MARCO) and DARPA), and the NSF under grants CCF-1117147 and CCF-1253700.

REFERENCES

1. A. Waterman et al., *The RISC-V instruction set manual, volume I: User-level ISA, version 2.1*, technical report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, 2016.
2. Y. A. Manerkar et al., "Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings," *CoRR*, 2016.
3. C. Trippel et al., "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 119–133.
4. K. Asanovic et al., *The Rocket Chip generator*, technical report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.
5. O. Lahav et al., *Repairing sequential consistency in C/C++11*, technical report MPI-SWS-2016-011, 2016.
6. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computing*, vol. 28, no. 9, 1979, pp. 690–691.
7. K. Gharachorloo, *Memory Consistency Models for Shared-memory Multiprocessors*, thesis, Stanford University, 1996.

8. K. Gharachorloo et al., "Memory consistency and event ordering in scalable shared-memory multiprocessors," *17th International Symposium on Computer Architecture* (ISCA), 1990.
9. W.W. Collier, *Reasoning About Parallel Architectures*, Prentice Hall, 1992.
10. S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, 1996, pp. 66–76.
11. H.J. Boehm and S. Adve, "Foundations of the C++ concurrency memory model," *29th Conference on Programming Language Design and Implementation* (PLDI), 2008.
12. J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 2, 2014.
13. *Cortex-A9 MPCore Programmer Advice Notice, Read-after-Read Hazards*, ARM, 2011; http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf.
14. H.J. Boehm et al., *P0668R0: Revising the C++ memory model*, 2017; www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0668r0.html.

ABOUT THE AUTHORS

Caroline Trippel is a PhD candidate in the Computer Science department at Princeton University. Her research focuses on computer architecture, with a particular emphasis on concurrency and security verification in heterogeneously parallel systems. Trippel has a master's degree in computer science from Princeton University. She is a student member of the ACM. Contact her at ctrippel@princeton.edu.

Yatin A. Manerkar is a PhD candidate in the Computer Science department at Princeton University. His research focuses on MCMs and formal verification of their implementation. Manerkar has a master's degree in computer science and engineering from the University of Michigan. He is a student member of the ACM. Contact him at manerkar@princeton.edu.

Daniel Lustig is a senior research scientist at Nvidia. His research focuses on architecting efficient and correct memory systems, with a particular focus on MCM design and verification. Lustig has a PhD in electrical engineering from Princeton University. He is an ACM and IEEE member. Contact him at dlustig@nvidia.com.

Michael Pellauer is a senior research scientist at Nvidia. His research focuses on computer architecture, with emphasis on nonstandard accelerator architectures using spatial programming. Pellauer has a PhD in computer science from the Massachusetts Institute of Technology (MIT). Contact him at mpellauer@nvidia.com.

Margaret Martonosi is the Hugh Trumbull Adams '35 Professor of Computer Science at Princeton University. Her research focuses on computer architecture and mobile systems, with a particular emphasis on verification, performance, and power efficiency in heterogeneous systems. Martonosi has a PhD in electrical engineering from Stanford University. She is a Fellow of IEEE and the ACM. Contact her at mrm@princeton.edu.