Verifying Correct Microarchitectural Enforcement of Memory Consistency Models

Memory consistency models define the rules and guarantees about the ordering and visibility of memory references on multithreaded CPUs and systems on chip. PipeCheck offers a methodology and automated tool for verifying that a particular microarchitecture correctly implements the consistency model required by its architectural specification.

Daniel Lustig
Princeton University

Michael Pellauer
Intel

Margaret Martonosi
Princeton University

Memory consistency models (MCMs) are notoriously difficult to work with. Although they are central to correct system operation, they are hard to build, verify, and even define. Weak memory models were originally developed in the 1980s to sacrifice the intuitive simplicity of sequential consistency in favor of a large performance boost. Most architects and programmers now consider this tradeoff to be worthwhile; few modern processors implement sequential consistency.

Unfortunately, architects have not converged on any single point within the performance versus simplicity spectrum, leaving a wide variety of MCMs in use today. Models such as total store ordering (TSO), used by Sparc and x86(-64), are more conservative but may leave some performance on the table. Power and ARM processors reorder liberally by default, but reasoning about how to enforce ordering (for example, via fences) in these models is difficult even by consistency model standards.

MCMs' complexity is exacerbated by the modern trend toward architectural heterogeneity. Systems no longer comprise CPUs sharing a single instruction set architecture (ISA). Instead, there could be as many as a half-dozen ISAs—and hence a half-dozen consistency models—on a modern mobile system on a chip (SoC), and this number is likely only to increase. Furthermore, memory-accessing elements such as fixed-function video decoders may not even have traditional ISAs at all; these elements rely solely on the memory consistency model to communicate. Thus, MCMs have become a central form of abstraction in an increasingly heterogeneous landscape. All of these problems motivate the need to pay increased attention to properly specifying and verifying the correct consistency model behaviors of the multitude of computation elements on chip.

This article describes an analysis methodology for verifying that a given microarchitecture meets the specifications of a given architectural consistency model, and it presents PipeCheck,

Published by the IEEE Computer Society

an automated tool for implementing this technique. PipeCheck brings axiomatic memory model analysis techniques to the microarchitecture level, defining "microarchitectural-level happens-before" graphs at the granularity of instructions passing through particular pipeline stages. Using statements about the reordering behavior of individual stages (such as "the decode stage is an in-order stage"), PipeCheck verifies that each ordering edge that must be preserved according to the architectural consistency model (for example, each Store→Store ordering for TSO) is in fact provably maintained by the microarchitecture. As a result, PipeCheck reduces the problem of verifying global consistency model implementation correctness to the more tractable problem of verifying local reordering properties at various points in the microarchitecture.

Our hope is that architects will use our open source PipeCheck tool (publicly available at github.com/daniellustig/pipecheck) and its analysis techniques to design chips with increased resilience against the kinds of consistency and memory system bugs that continue to haunt hardware even today.

PipeCheck: Microarchitecture-level analysis

Architecture-level memory consistency model specifications say nothing about the behavior of any individual microarchitectural implementation. On one hand, certain architecturally permitted behaviors might not be observable on a given microarchitecture. For example, a sequentially consistent (SC) pipeline is a valid implementation of the TSO memory model, although many executions that are legal under TSO will not be observable in such a pipeline—the microarchitectural memory model is stricter than the architectural model MCM requires. On the other hand, architecturally forbidden behaviors might be observable on a given microarchitecture, and this would mean that the implementation has a bug.

PipeCheck aims to formalize and automate this comparison of microarchitecture versus architecture. It extends axiomatic memory model analysis techniques to the microarchitecture space, creating microarchitecture-level happens-before graphs. Here, we describe how these graphs are created and used for verifying a microarchitecture's correctness with respect to a given memory model.

Microarchitecture-level happens-before graphs

Orderings between instructions are often too complicated to be captured by a single architecture-level happens-before edge. A single pair of instructions may fetch in order, issue out of order, execute in order, commit in order, and reach memory out of order. PipeCheck therefore defines microarchitecture-level happens-before (μhb) edges in terms of both instructions and particular locations within the pipeline:

Definition 1. Microarchitectural-Level Happens-Before: A μhb graph is a directed graph (V, E) in which each vertex (inst@loc) $\in V$ represents a memory instruction inst passing through a particular location loc, and each edge ($\text{inst}_i@\text{loc}_a$, $\text{inst}_j@\text{loc}_b$) represents a guarantee that instruction inst_i passes through location loc_a before instruction inst_j passes through location loc_b.

We depict μhb graphs in a grid with instructions along the x-axis and microarchitectural locations along the y-axis. Not all instructions pass through all locations (for example, loads do not occupy the store buffer), and so some entries in the grid are left empty. Despite the grid depiction, only relationships depicted by arrows provide any ordering guarantee.

Figure 1 shows the μhb graph for the message passing (mp) litmus test (discussed in the "Memory Model Analysis" sidebar) executing on a processor with standard five-stage in-order pipelines. The four memory operations—i1, i2, i3, and i4—are depicted from left to right, and various locations in the microarchitecture are shown from top to bottom. Each vertex represents an instruction at a particular location within the microarchitecture. Each row of vertices captures the ordering of instructions at a particular location within the pipeline, and each column of vertices therefore corresponds to an instruction progressing through various locations in the microarchitecture.

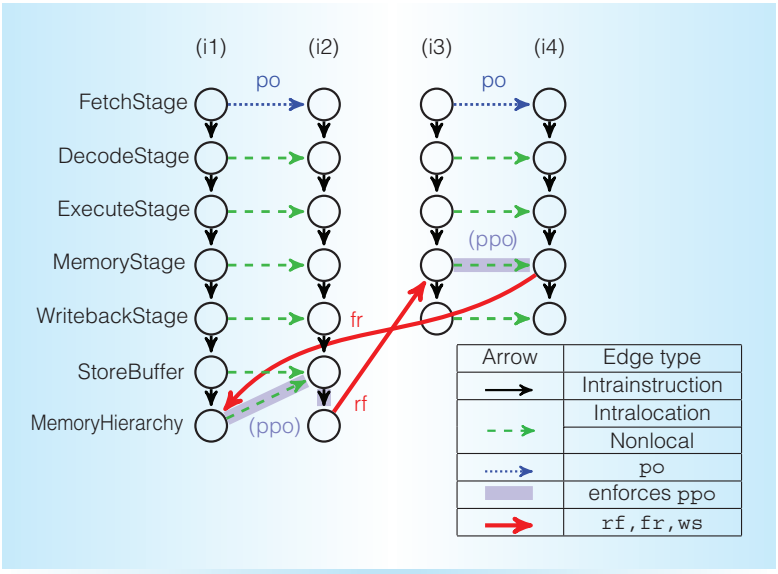

Figure 1. PipeCheck microarchitecture-level happens-before (μhb) graph. The depicted execution of the message passing litmus test (see the "Memory Model Analysis" sidebar) has a cycle and hence is not observable on this pipeline.

Microarchitecture definition

In PipeCheck, a microarchitecture is defined by

- a list of locations;
- legal path(s) per instruction type;
- performing locations within each path;
- a local ordering guarantee at each location; and
- nonlocal edges, or edges that are both interinstruction and interlocation.

We will define these terms in more detail in this article.

Running example. Table 1 shows the PipeCheck definition of the classic five-stage pipeline depicted in Figure 2. The rows of the table are microarchitectural locations. The two middle columns define the possible paths each class of instructions can take through the pipeline. Note that, in general, instructions can have more than one choice of path through the microarchitecture. The last column defines the local ordering guarantees at each location. The footnotes specify the performing locations for each type of instruction as well as a set of nonlocal edges specific to the store buffer.

Instruction paths. During execution, as instructions flow through the pipeline, they pass through a specified set of locations along some well-defined path. A memory instruction can have more than one legal path through a pipeline. For example, a read can take a different path depending on whether it forwards from the store buffer, reads from the cache via a cache hit, or reads from the cache after a cache miss.

Performing locations. Each path also defines the set of locations at which each instruction can perform. Traditionally, a store has performed when a (potentially hypothetical) load may read the value, and a load has performed when a (potentially hypothetical) store may not change the value returned.[1] The notion of performing is in turn used to define the behavior of properties such as the cumulativity of fences on some weak architectures. This classical definition of performing is fundamentally hypothetical and thus difficult to work with, because happens-before relationships are made to inherently depend on loads and stores that do not actually exist in a program and hence cannot easily be referenced during analysis. This difficulty is reflected in the wide variety of definitions of cumulativity used in the literature.

PipeCheck defines the point at which an instruction has performed in terms of location rather than the traditional notion of potential visibility:

Definition 2. Performing Location: A location l is a performing location with respect to core c if a load at location l can read the value written by a store from core c, and the data being written by a store at location l is visible to core c. A location l is a global performing location if it is a performing location with respect to all cores.

In PipeCheck, the transitivity of edges makes it straightforward to check whether one instruction performs before another. One simply checks whether there are one or more μhb edges that connect the performing locations of the two instructions.

Local ordering guarantees. To more precisely define in-order and out-of-order, we define a

Memory Model Analysis

Axiomatic memory models represent programs as graphs. Vertices represent instructions; an edge from a node *s* to another node *d* indicates that *s* happens before *d* in a formal sense defined by the model. A cycle in an axiomatic memory model graph indicates that a proposed execution is disallowed, with important exceptions made to account for certain weak memory behavior.[1] This reflects the intuition that an instruction cannot happen before itself. Acyclic graphs correspond to permitted executions.

Figure A depicts the standard axiomatic analysis of the message passing (`mp`) litmus test, a program written specifically to test a consistency model. This particular test asks whether some execution of the two threads produces the result r1 = 1 and r2 = 0 on a processor implementing the total store ordering (TSO) consistency model (see Table A), used by SPARC and x86(-64). All memory locations are assumed to hold the value 0 originally. Working backwards, since *r*1 receives the value 1, *i*2 must have happened before *i*3. Similarly, *i*4 must have happened before *i*1, because otherwise *i*4 would also have returned the value 1. As Table A indicates, TSO itself guarantees the Load→Load and Store→Store orderings within each thread; these constraints are called the preserved program order (`ppo`). As Figure B shows, these four edges form a cycle, indicating that the outcome is forbidden under TSO.

Table A. Preserved program order (`ppo`) in the total store ordering (TSO) memory model. Must an access of the type in the row heading maintain its ordering with respect to a subsequent instruction of the type in the column heading?

	Load	Store
Load	√	√
Store	–(`mfence`)	√

* √: *enforced by default.* –: *not enforced by default.* (`mfence`) : *enforced by* `mfence`.

Core 0	Core 1
(i1) [x] <-- 1	(i3) r1 <-- [y]
(i2) [y] <-- 1	(i4) r2 <-- [x]
Proposed outcome at core 1: r1 = 1, r2 = 0. Outcome forbidden under TSO	

Figure A. Code for message passing (`mp`) litmus test.

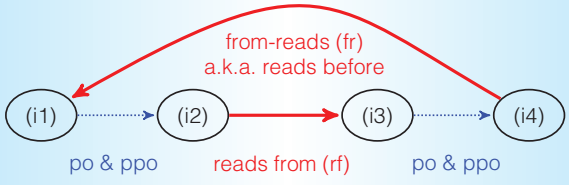

Figure B. Architecture-level analysis of Load→Load and Store→Store ordering litmus test `iwp2.1/amd1/mp`. The cycle indicates that this execution is forbidden under the rules of TSO.

Reference

1. J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Programming Languages and Systems* (TOPLAS), vol. 36, no. 2, 2014, article 7.

local ordering guarantee at each location. This specifies the reorderings that location does or does not permit on instructions passing through it. At one extreme, a first-in, first-out (*FIFO*) local ordering specifies that all interinstruction orderings guaranteed at entry into a location will also be guaranteed leaving that location. At the other extreme, a *NoGuarantees* local ordering specifies that no orderings are guaranteed for instructions leaving the location. Other guarantees may lie in between. The specific guarantees of each pipeline stage will vary from processor to processor.

Nonlocal edges. Nonlocal μhb edges model any ordering guarantees implemented by the pipeline across multiple instructions and locations. Such nonlocal μhb edges are relatively rare; they correspond to nonlocal wires and/or communication across a chip, making them expensive in practice.

Table 1. PipeCheck definition of a classic five-stage pipeline

#	Access type: Loads	Access type: Stores	Local ordering guarantee
0	Fetch	Fetch	FIFO
1	Decode	Decode	FIFO
2	Execute	Execute	FIFO
3	Memory	Memory	FIFO
4	Writeback	Writeback	FIFO
5		Store Buffer	FIFO
6		Memory Hierarchy	NoGuarantees

*Loads perform globally at the memory stage. Stores perform locally (that is, enter the store buffer) at the memory stage, and they perform remotely when reaching the memory hierarchy. Only one store can be outstanding from the store buffer at a time: for all stores s, for the immediately subsequent store s', (s@MemHierarchy)→(s'@StoreBuffer).

However, they often serve to enforce critical ordering guarantees. An example of such a nonlocal edge is a store buffer that enforces that "after issuing a request to memory, the store buffer must await an acknowledgment from memory before issuing a subsequent request"—a property that is often critical to the enforcement of Store→Store orderings in TSO.

Generating μhb graphs

Given a microarchitecture definition and a program, PipeCheck automatically enumerates the set of all μhb graphs representing all possible executions of the program. This process is broken into two steps: enumeration of static edges, or those which are true in every execution of a program, and enumeration of observed edges, or those inferred during a particular execution of that program.

Static edges. We begin by adding a set of intrainstruction μhb edges between consecutive locations along the path for that instruction. For example, an instruction being in the fetch stage will "microarchitecturally happen before" the point when that same instruction is in the decode stage. These are represented by the solid vertical arrows in Figure 1.

Next, each location observes instructions passing through in some order. We assume program order to be the ordering of instructions at the fetch stage of the pipeline. Some subsequent pipeline stages also guarantee to maintain intralocation ordering guarantees propagated from previous stages. We add intralocation μhb edges to represent these per-location guarantees. These are represented by the dashed horizontal arrows in Figure 1.

Finally, we add the nonlocal edges defined by the pipeline specification. For example, the definition of the five-stage pipeline (see Table 1) contains a nonlocal edge to describe the store buffer's behavior. This is drawn as the diagonal dashed edge from (i1@MemoryHierarchy) to (i2@StoreBuffer) in Figure 1.

Observed edges. PipeCheck enumerates three types of observed edges. The "Memory Model Analysis" sidebar discusses two examples: "reads from" (`rf`) and "from reads" (`fr`). The third type is "write serialization" (`ws`), also known as "coherence," which places a total order on all stores made to each address.

PipeCheck defines the endpoints of observed edges to be at the performing location(s) of each instruction's path. When there is more than one possibility (for example, a load can read from either the store buffer or memory), PipeCheck analyzes each independently. The cross product of the set of `rf`, `ws`, and path choices forms the set of graphs that need to be evaluated.

Properties of μhb graphs

PipeCheck μhb graphs have a number of properties that make them particularly suitable for use in verification. We discuss these below.

Transitivity of μhb edges. Axiomatic memory models capture the complexity of weak ordering behavior in one of two ways. Many models place the complexity into the edges. In such models, graphs are smaller, but execution-forbidding cycles can be found only within carefully chosen subsets of edges, and the transitive closure of happens-before edges is itself not always a happens-before edge.[2] Other models, including PipeCheck, define larger graphs in which each node represents an instruction plus metadata (that is, in PipeCheck, a pipeline location). The extra information in nodes (and hence also in edges) means that edges can be transitively composed and that any cycle serves to forbid an execution. This simplifies the analysis and restores the intuitive one-to-one correspondence between cycles and forbidden executions.

Graph size and tractability. Although PipeCheck μhb graphs are larger than those created by many other axiomatic models, they nevertheless remain very tractable to analyze. Each graph's size is roughly proportional to the number of instructions being analyzed times the depth of the pipeline. As such, μhb graphs typically have no more than a hundred nodes. Furthermore, although each analysis generally produces more than one graph, these graphs can be analyzed entirely independently in parallel. Nevertheless, our results show that even naive sequential analysis remains tractable, generally running to completion within just a few minutes.

Verification methodology

Here, we describe the high-level verification approach, as well as the design and usage of our PipeCheck tool, which automates the process.

Verification types

PipeCheck verifies pipeline correctness using two techniques: direct satisfaction tests and litmus tests.

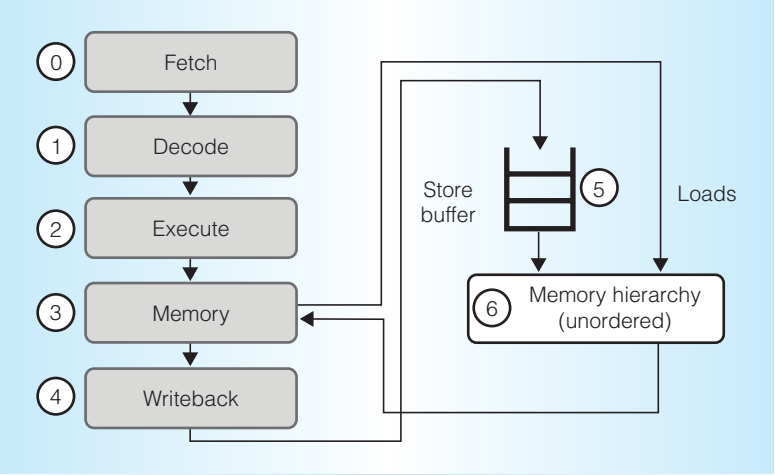

Figure 2. Graphical representation of a classic five-stage pipeline plus a store buffer and an unordered memory system. We use this relatively simple pipeline as a running example throughout the article.

Direct satisfaction tests. The first approach is to directly check whether each required architecture-level happens-before (hb) edge requirement is enforced by one or more μhb edges. A given architectural memory model can therefore generate direct satisfaction tests to check preserved program order (`ppo`), program order accesses to the same address (`po-addr`), dependency orderings, fence orderings, and so on. PipeCheck ensures that the microarchitectural interpretation of each hb edge is in fact present in the transitive closure of each μhb graph. As an example, the highlighted edges in Figure 1 represent μhb edges found to enforce the hb requirements of `ppo` for TSO.

Litmus tests. PipeCheck also evaluates each microarchitecture using a suite of litmus tests built up from existing repositories.[3] Architectural analysis determines whether the outcome specified by each litmus test is permitted or forbidden; PipeCheck calculates whether the outcome for each test is observable or not on the given microarchitecture. A permitted but unobserved outcome means that the pipeline is stronger than strictly necessary. A forbidden but observed outcome, however, indicates either a pipeline bug or an incorrect microarchitecture specification.

PipeCheck automated tool

Figure 3 shows the PipeCheck tool flow. PipeCheck is written using Coq,[4] an interactive

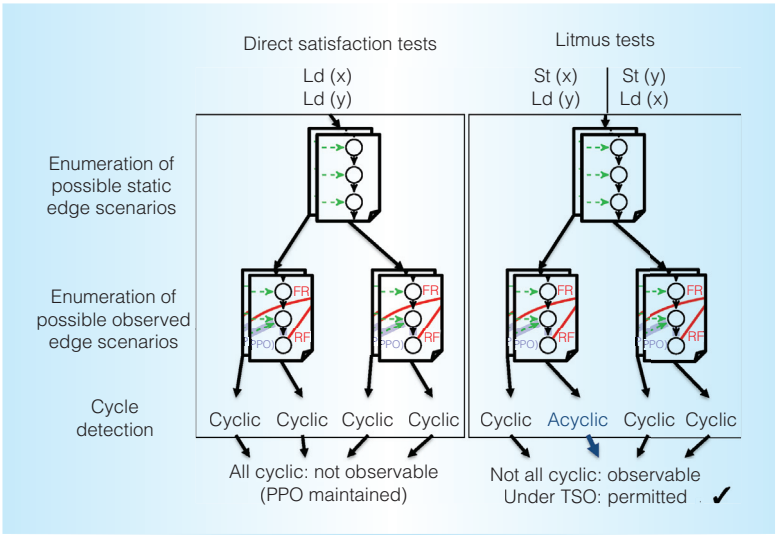

Figure 3. PipeCheck verification flow. The enumeration of graphs and the process of checking graphs for cycles are entirely automated.

theorem prover, to make the code amenable to formal analysis and integration with existing open source frameworks also using Coq.[5] To speed up the analysis, we use built-in functionality within Coq to export the code into OCaml and then compile this extracted code into a stand-alone binary. We then measured the runtime of this binary's execution on an Intel Xeon E3-1230v2 processor.

We evaluated PipeCheck by verifying processors implementing the TSO consistency model. TSO imposes nontrivial `ppo` ordering requirements on all memory operations and is in widespread use. Both facts make it a particularly interesting target.

We analyzed four pipelines. The first two are the five-stage pipeline in Figure 2, both without and with a store buffer. The former is effectively sequentially consistent, meaning that some litmus test outcomes permitted under TSO might (legally) not be observable. These two microarchitectures reflect pipelines that might be used in classrooms or as embedded cores. The third is the O3 (out-of-order) pipeline from the gem5 simulator.[6] This represents a medium-sized core and demonstrates how simulated cores are also amenable to analysis. Finally, we describe the OpenSPARC T2 pipeline, representing a well-documented industry-strength microarchitecture.[7]

We analyzed each litmus test on a four-core version of each pipeline. We also analyzed the set of `ppo` and `po-addr` direct satisfaction tests for each pipeline.

Results across litmus tests

Table 2 shows the results of verifying the suite of litmus tests on each modeled pipeline. Individual litmus tests are depicted as rows. For each row, the table shows whether TSO forbids or permits the outcome proposed by the test, and then shows its observability on the four microarchitectures. The microarchitecturally observable behaviors correspond with the architecturally specified behaviors in almost all cases. For the five-stage pipeline without a store buffer, six of the proposed results require non-SC behavior, and these results are confirmed as not being observable on the SC pipeline. On the other hand, test results for the gem5 pipeline indicate the presence of a bug.

Figure 4 shows the time taken to complete the verification process for each pipeline. The entire suite completes in just 10 minutes for each pipeline, demonstrating that even with code optimized for verifiability rather than performance, PipeCheck analysis is very practical.

Advanced microarchitectural optimizations

Many processors deliver improved performance through microarchitectural optimizations such as out-of-order execution, speculative load reordering, and value prediction. The desire to include such optimizations was the key motivation for building weak memory models at all. However, optimizations must make sure to follow the rules of the architectural memory model within which they are implemented. PipeCheck now provides a rigorous framework within which such verification can take place.

An interesting complication arises with microarchitectural optimizations that maintain the appearance of following the rules even while technically violating them. Much as pipelines are permitted to perform out-of-order execution as long as in-order semantics are maintained, pipelines can (and do) implement features, such as speculative load reordering, which violate the letter of the memory model specification but which nevertheless maintain the external appearance of correct behavior. PipeCheck supports

Table 2. Summary of litmus test results.

Litmus test	Total store ordering (expected)	Modeled pipelines: 5-stage (no store buffer)	Modeled pipelines: 5-stage (with store buffer)	gem5 O3	OpenSPARC
iwp2.1/amd1/mp	F	=	=	O[2]	=
iwp2.2/amd2/lb	F	=	=	=	=
iwp2.3a/amd4/sb	P	N[1]	=	=	=
iwp2.3b	P	=	=	=	=
iwp2.4/amd9	P	N[1]	=	=	=
iwp2.5/amd8/wrc	F	=	=	O[2]	=
iwp2.6	F	=	=	=	=
amd3	P	N[1]	=	=	=
amd6/iriw	F	=	=	O[2]	=
n1	P	N[1]	=	=	=
n2	F	=	=	O[2]	=
n4	F	=	=	=	=
n5	F	=	=	=	=
n6	P	=	=	=	=
n7	P	N[1]	=	=	=
rwc	P	N[1]	=	=	=

*"F": Forbid. "P": Permit. "=": Matches expected TSO outcome. "O": Observable. "N": Not observable. [1]: Implementation more restrictive than TSO requires. [2]: Indicates the presence of a bug.

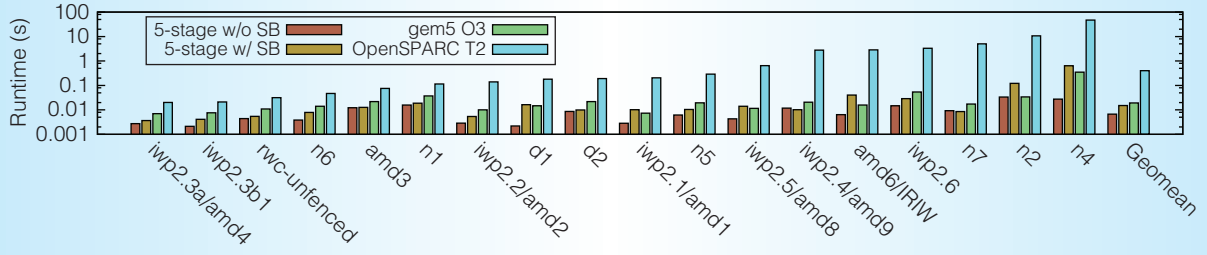

Figure 4. Verification time results (computed using extracted OCaml). Even in the worst case, the analysis is fast, making verification very tractable in practice.

verification of these features as well. In such cases, a literal interpretation of architecture-level requirements such as Load→Load ordering might not be verifiable, but in these cases correctness should nevertheless be enforced by replacement μhb edges.

Case study: Speculative load reordering

The key principle behind speculative load reordering is that two loads l_1 and l_2 in program order can be speculatively reordered (that is, l_2 can perform before l_1) as long as the value read speculatively by l_2 is the same as it would have been had l_2 in fact performed nonspeculatively (that is, after l_1).[8] The implementation used by the gem5 O3 pipeline snoops for cache line invalidations. Specifically, if a cache line has not been overwritten or invalidated (due to cache replacement or external request) since an earlier speculative read of that line, the core can safely assert that a subsequent read of that

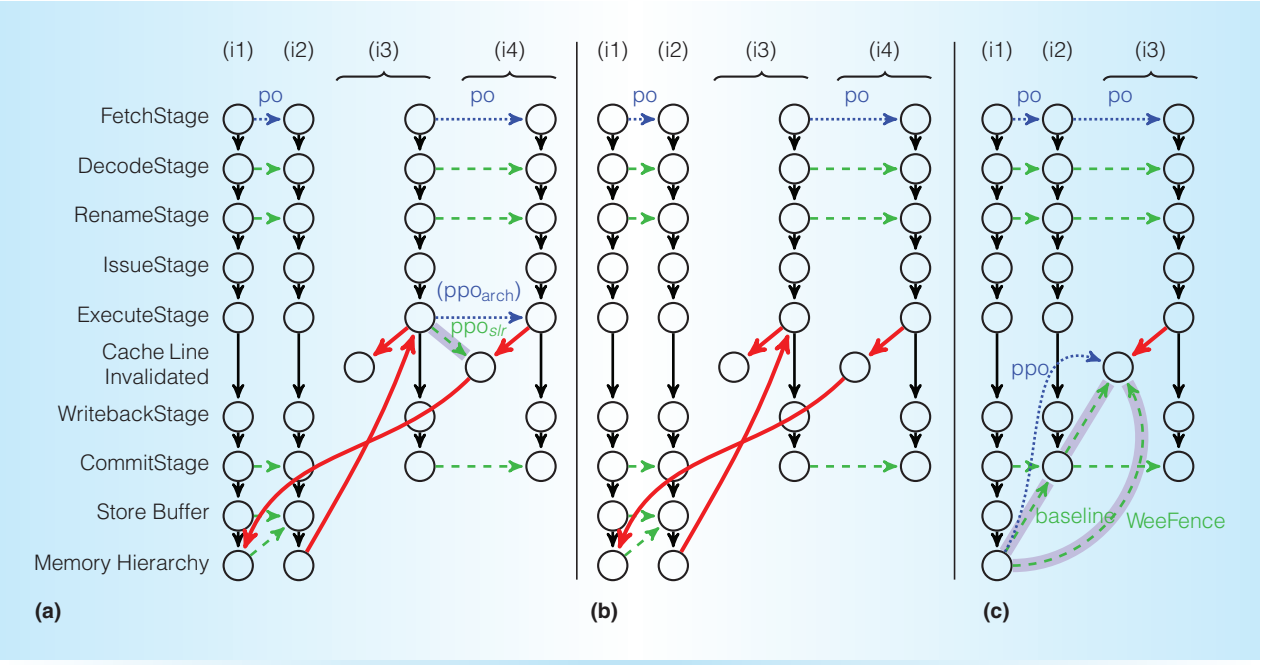

Figure 5. Case studies on the gem5 O3 pipeline. (a) Speculative load reordering. Although $\mathtt{ppo}_{\mathtt{arch}}$ is not enforced, a legal replacement $\mathtt{ppo}_{\mathtt{slr}}$ is enforced, and it completes the cycle. (b) Pipeline bug shown via the `iwp2.1/amd1/mp` litmus test. The lack of a cycle indicates that the behavior is (erroneously) observable. (c) WeeFence eliminates the slow baseline dependency while maintaining the necessary ordering.[9]

line would return the same value. On the other hand, if the cache line is modified or invalidated, the core is conservative and assumes that the invalidation indicates a failed speculation.

We can model this implementation of speculative load reordering in PipeCheck by including cache line invalidation as an extra location within the instruction path. Figure 5a shows an example within the gem5 O3 pipeline model for the `mp` litmus test. Extra vertices represent the invalidations of the cache lines that i3 and i4 read from, and the observed edges in the graph have been adjusted to account for these new vertices. In particular, the cache line that i4 reads from must have been invalidated before i1 wrote to memory to observe the proposed result.

Case study: gem5 pipeline bug

For the gem5 O3 pipeline, our `ppo` direct satisfaction tests indicated that Load→Load `ppo` ordering was not guaranteed, and four litmus tests that relied on such ordering failed validation. As an example of a failed test, Figure 5b shows the μhb graph for `mp` executing on this pipeline. To analyze further, we wrote a microbenchmark to execute `mp` in a tight loop. With this test, software observed the forbidden result, confirming the presence of the bug as well as its cause. This bug was fixed by a third party in revision 10149.

Case study: WeeFence

WeeFence is a recent optimization proposal that allows post-fence loads to perform and retire before stores prior to the fence.[9] WeeFence buffers or bounces invalidation requests to cache lines relevant to a pending fence, thereby allowing post-fence reads to safely retire nonspeculatively even before prefence stores have written back to memory. Although this violates the letter of the fence semantics, it maintains the appearance of correct execution.

Figure 5c demonstrates the use of PipeCheck to validate the WeeFence optimization (within the corrected gem5 O3 pipeline). Both the baseline (non-WeeFence) and the WeeFence approaches enforce the (i1@MemoryHierarchy)→(i3@CacheLineInvalidate) ordering, but WeeFence does so without the slow intermediate step of (i2@CommitStage), thereby saving latency over the

baseline. This analysis demonstrates how PipeCheck can be used to specify and demonstrate the correctness of a new microarchitectural optimization proposal.

We hope that techniques such as PipeCheck can help bring attention to both the need and the opportunity to verify new microarchitectural optimization proposals. Although performance is the primary motivation for most such proposals, performance results should only be considered meaningful once correctness has been established. Incorrect (or even nearly correct) microarchitectures may (even unintentionally) benefit from artificially inflated performance, thereby placing correct proposals at an unfair disadvantage. Litmus tests such as `iriw` arose after long discussions in the community about the performance costs of implementing strong ordering semantics for programming idioms that are widely considered esoteric. Nevertheless, models such as TSO require strong semantics despite their cost. Proposals that claim to implement TSO should be expected to demonstrate the correctness of `iriw` before presenting performance numbers. The time has come for microarchitects to accept the burden of establishing correctness in a rigorous manner.

Fortunately, analysis techniques and tools are quickly approaching a point at which automated, systematic verification is possible. Precise formal models of many architectures now exist, as do large, well-established suites of litmus tests for popular ISAs, including x86(-64), Power, and ARM. We hope that PipeCheck is useful in extending rigorous analysis techniques into the microarchitecture space, thereby providing researchers with a straightforward and reliable way to demonstrate their proposals' correctness.

MICRO

Acknowledgments

We thank Jade Alglave, Lennart Beringer, James Bornholt, Doug Clark, Nirav Dave, and Kathryn McKinley for their helpful feedback. Daniel Lustig was supported in part by an Intel PhD Fellowship. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was also supported in part by NSF under grant CCF-1117147.

References

1. M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l Symp. Computer Architecture*, 1986, pp. 434–442.
2. J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Programming Languages and Systems* (TOPLAS), vol. 36, no. 2, 2014, article 7.
3. S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: x86-TSO," *Proc. 22nd Int'l Conf. Theorem Proving in Higher Order Logics*, 2009, pp. 391–407.
4. *The Coq Proof Assistant Reference Manual, Version 8.0*, LogiCal, Apr. 2004.
5. J. Alglave, "A Formal Hierarchy of Weak Memory Models," *Formal Methods in System Design*, vol. 41, no. 2, 2012, pp. 178–210.
6. N. Binkert et al., "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011, pp. 1–7.
7. *OpenSPARC T2 Core Microarchitecture Specification, Revision A*, Sun, 2007; www.oracle.com/technetwork/systems/opensparc/t2-06-opensparct2-core-microarch-1537749.html.
8. K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. 29th Int'l Conf. Parallel Processing*, 1991, pp. 355–364.
9. Y. Duan, A. Muzahid, and J. Torrellas, "WeeFence: Toward Making Fences Free in TSO," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 213–224.

Daniel Lustig is a PhD candidate in the Department of Electrical Engineering at Princeton University. His research focuses on the design and verification of memory systems for heterogeneous computing platforms. Lustig has an MA in electrical

engineering from Princeton University. He is a student member of IEEE and the ACM. Contact him at dlustig@princeton.edu.

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. He performed the research for this article while at Intel. 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 computing, with emphasis on power-efficient 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.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*