Transistency Models: Memory Ordering at the Hardware–OS Interface

Daniel Lustig
Princeton University

Geet Sethi
Abhishek Bhattacharjee
Rutgers University

Margaret Martonosi
Princeton University

This article introduces the transistency model, a set of memory ordering rules at the intersection of virtual-to-physical address translation and memory consistency models. Using their COATCheck tool, the authors show how to rigorously model, analyze, and verify the correctness of a given system's microarchitecture and software stack with respect to its transistency model specification.

Modern computer systems consist of heterogeneous processing elements (CPUs, GPUs, accelerators) running multiple distinct layers of software (user code, libraries, operating systems, hypervisors) on top of many distributed caches and memories. Fortunately, most of this complexity is hidden away underneath the virtual memory (VM) abstraction presented to the user code. However, one aspect of that complexity does pierce through: a typical memory subsystem will buffer, reorder, or coalesce memory requests in often unintuitive ways for the sake of performance. This results in essentially all real-world hardware today exposing a weak memory consistency model (MCM) to concurrent code that communicates through shared VM.

The responsibilities for maintaining the VM abstraction and for enforcing the memory consistency model are shared between the hardware and the operating system (OS) and require careful coordination between the two. Although MCMs at the instruction set architecture (ISA) and programming language levels are becoming increasingly well understood,[1–5] a key verification challenge is that events within system layers can behave differently than the "normal" accesses described by the ISA or programming language MCM. For example, on the x86-64 architecture, which implements the relatively strong total store ordering (TSO) memory model,[5] page table walks are automatically issued by hardware, can happen at any time, and often are not ordered even with respect to fences. Even worse is that while an ISA by design tends to remain stable across processor generations, microarchitectural phenomena often change dramatically from one generation to the next. For example, CPUs today are experimenting

Published by the IEEE Computer Society

0272-1732/17/$33.00 © 2017 IEEE

with features such as concurrent page table walkers and translation lookaside buffer (TLB) coalescing that improve performance at the cost of adding significant complexity.[6] Consequently, VM and MCM specifications and implementations tend to be bug-prone and are only becoming more complex as systems become increasingly heterogeneous and distributed.

Bogdan Romanescu and colleagues were the first to distinguish between MCMs meant for virtual addresses (VAMC) and those for physical addresses (PAMC).[7] They considered hardware to be responsible for the latter, and a combination of hardware and OS for the former. However, as we show in this article, not even VAMC and PAMC capture the full intersection of address translation and memory ordering. Even machines that implemented the strictest model they considered—virtual address sequential consistency (SC-for-VAMC)—may be prone to surprising ordering bugs related to the checking of metadata at a different virtual and physical address from the data being accessed. We therefore coin the term *memory transistency model* to refer to any set of memory ordering rules that explicitly account for these broader virtual-to-physical address translation issues.

To enable rigorous analysis of transistency models and their implementations, we developed a tool called COATCheck for verifying memory ordering enforcement in the context of virtual-to-physical address translation. (COAT stands for consistency ordering and address translation.) COATCheck lets users reason about the ordering implications of system calls, interrupts, microcode, and so on at the microarchitecture, architecture, and OS levels. System models are built in COATCheck using a domain-specific language (DSL) called *μspec* (pronounced "mu-spec"), within which each component in a system (for example, each pipeline stage, each cache, and each TLB) can independently specify its own contribution to memory ordering using the languages of first-order logic and microarchitecture-level happens-before (μhb) graphs.[8,9] This allows COATCheck verification to be modular and flexible enough to adapt to the fast-changing world of modern heterogeneous systems.

Our contributions are as follows. First, we developed a comprehensive methodology for specifying and statically verifying memory ordering enforcement at the hardware–OS interface. Second, we built a fast and general-purpose constraint solver that automates the analysis of μspec microarchitecture specifications. Third, as a case study, we built a sophisticated model of an Intel Sandy-Bridge-like processor running a Linux-like OS, and using that model we analyzed various translation-related memory ordering scenarios of interest. Finally, we identified cases in which transistency goes beyond the traditional scope of consistency: where even SC-for-VAMC[7] is insufficient. Overall, our work offers a rigorous yet practical framework for memory ordering verification, and it broadens the very scope of memory ordering as a field. The full toolset is open source.[10]

Enhanced Litmus Tests

Litmus tests are small stylized programs testing some aspect of a memory model. Each test proposes an outcome: the value returned by each load plus the final value at each memory location, or some relevant subset thereof. The rules of a memory model determine whether an outcome is permitted or forbidden. Consider Figure 1a: as written, x and y appear to be distinct addresses. Under that assumption, the proposed outcome is observable even under a model as strict as sequential consistency (SC),[11] because the event interleaving shown in Figure 1b produces that outcome. If instead `x` and `y` are actually synonyms (both map to the same physical address), as in Figure 1c, the test is forbidden by SC, because then no interleaving of the threads produces the proposed outcome. While simple, this example highlights how memory ordering verification is fundamentally incomplete unless it explicitly accounts for address translation.

The basic unit of testing in COATCheck is the enhanced litmus test (ELT). ELTs extend traditional litmus tests by adding address translation, memory (re)mappings, interrupts, and other system-level operations relevant to memory ordering. In addition, just as a traditional litmus test outcome specifies the values returned by loads, ELTs also

Initially: [x] = 0, [y] = 0	
Thread 0	**Thread 1**
St [x] ← 1	St [y] ← 2
Ld [y] → r1	Ld [x] → r2
Proposed outcome: r1 = 2, r2 = 1	

(a)

Initially: [x] = 0, [y] = 0	
Thread 0	**Thread 1**
St PA1 ← 1	St PA2 ← 2
Ld PA2 → r1	Ld PA1 → r2
Outcome: r1 = 2, r2 = 1 permitted	

(b)

Initially: [x] = 0, [y] = 0	
Thread 0	**Thread 1**
St PA1 ← 1	St PA1 ← 2
Ld PA1 → r1	Ld PA1 → r2
Outcome r1 = 2, r2 = 1 forbidden	

(c)

Figure 1. A litmus test showing how virtual memory interacts with memory ordering. (a) Litmus test code. (b) A possible execution showing how the proposed outcome is observable if x and y point to different physical addresses. (c) The outcome is forbidden if *x* and *y* point to the same physical address (only one possible interleaving among many is shown).

consider the physical addresses used by each VM access to be part of the outcome. Finally, ELTs include "ghost instructions" that model lower-than-ISA operations (such as microcode and page table walks) executed by hardware, even if these instructions are not fetched, decoded, or issued as part of the normal ISA-level instruction stream. These features give ELTs sufficient expressive power to test all aspects of memory ordering enforcement as it relates to address translation.

The COATCheck toolflow provides automated methods to create ELTs from user-provided litmus tests plus other system-level annotation guidance. We describe this conversion process below.

OS Synopses

OS activities such as TLB shootdowns and memory (re)mappings are captured within ELTs as sequences of loads, stores, system calls, and/or interrupts. An OS synopsis specifies a mapping from each system call into a sequence of micro-ops that capture the effects of that system call on ordering and address translation. When the system call contains an interprocessor interrupt (IPI), the OS synopsis also instantiates predefined interrupt handler threads on interrupt-receiving cores.

For example, an OS synopsis might expand the `mprotect` call of Figure 2a into the shaded instructions of Figure 2b. The call itself expands into four instructions: one updates the page table entry, one invalidates the local TLB, one sends an IPI, and one waits for the IPI to be acknowledged. The OS synopsis also produces the interrupt handler (Thread 1b), which performs its own local TLB invalidation before responding to the initiating thread.

Microarchitecture Synopses

As with the OS synopses, microarchitecture synopses map each instruction onto a microcode sequence that includes ghost instructions such as page table walks. Not every instruction actually triggers a page table walk, so these ghost instructions are instantiated only as needed during the analysis.

For example, Figure 2b is transformed into the ELT of Figure 2c by the addition of the gray-shaded ghost instructions. In this example, Thread 0's store to [x] requires a page table walk, because the TLB entry for that virtual address would have been invalidated by the preceding `invlpg` instruction. Furthermore, because the page was originally clean, ghost instructions also model how hardware marks the page dirty at that point. Finally, the microarchitecture synopsis adds to Thread 1b a microcode preamble containing ghost instructions to receive the interrupt, save state, and disable nested interrupts. In this example, hardware is responsible for saving state, but software is responsible for restoring it. This again highlights the degree of collective responsibility between hardware and OS for ensuring ordering correctness.

μspec: A DSL for Specifying Memory Orderings

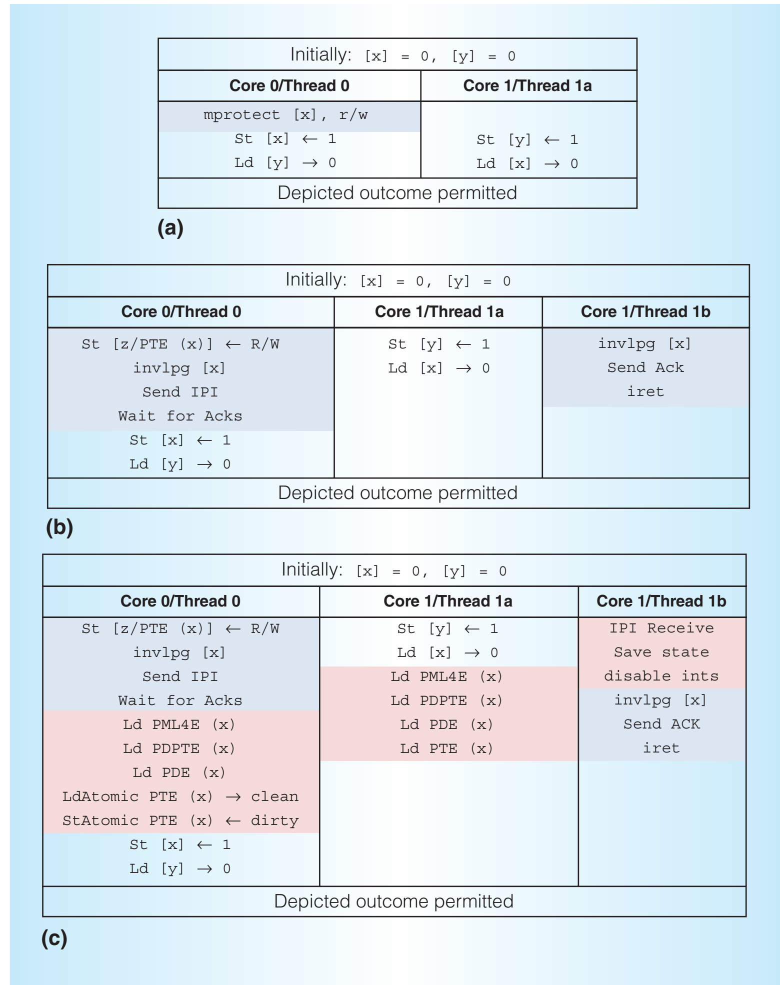
μspec is a domain-specific language for specifying memory ordering enforcement in the form of μhb graphs[8,9] (see Figure 3). Nodes in a μhb graph represent events corresponding

to a particular instruction (column) and some particular microarchitectural event (row). Edges represent happens-before orderings guaranteed by some property of the microarchitecture: an instruction flowing through a pipeline, a structure maintaining first-in, first-out (FIFO) ordering, the passage of a message, and so on. Although previous work derived μhb graphs using hard-coded notions of pipelines[8] and caches,[9] μspec models provide a completely general-purpose language for drawing μhb graphs tailored to any arbitrary system design. We provide a detailed example of the μspec syntax in the next section.

Hardware memory models today tend to be either axiomatic, where an outcome is permitted if and only if it simultaneously satisfies all of the axioms of the model, or operational, where an outcome is permitted only if it matches the outcome of some series of execution steps on an abstract "golden hardware model." μspec models are axiomatic: a μhb graph represents a legal test execution if and only if it is acyclic and satisfies all of the constraints in the model. Each hardware or software component designer provides an independent set of μhb graph axioms which that component guarantees to maintain. The conjunction of these axioms forms the overall μspec model. This modularity means that components can be added, changed, or removed as necessary without affecting any of the other components.

Although they are inherently axiomatic, μspec models capture the best of the operational approach as well. A total ordering of the nodes in an acyclic μhb graph is also analogous to the sequence of execution steps in an operational model. This analogy lets μhb graphs retain much of the intuitiveness of operational models while simultaneously retaining the scalability of axiomatic models. As such, μhb graphs are useful not only for transistency models but also more generally for software and hardware memory models.

The COATCheck constraint solver is inspired by SAT and SMT solvers. It searches to find any μhb graph that satisfies all of the constraints of a given μspec model applied to some ELT. If one can be found, the proposed ELT outcome is observable. If not, the proposed outcome is forbidden. This result is then checked against the architecture-level specification[1] to ensure correctness.



Initially: `[x] = 0, [y] = 0`

Core 0/Thread 0	Core 1/Thread 1a
mprotect [x], r/w	
St [x] ← 1	St [y] ← 1
Ld [y] → 0	Ld [x] → 0

Depicted outcome permitted

(a)

Initially: `[x] = 0, [y] = 0`

Core 0/Thread 0	Core 1/Thread 1a	Core 1/Thread 1b
St [z/PTE (x)] ← R/W	St [y] ← 1	invlpg [x]
invlpg [x]	Ld [x] → 0	Send Ack
Send IPI		iret
Wait for Acks		
St [x] ← 1		
Ld [y] → 0		

Depicted outcome permitted

(b)

Initially: `[x] = 0, [y] = 0`

Core 0/Thread 0	Core 1/Thread 1a	Core 1/Thread 1b
St [z/PTE (x)] ← R/W	St [y] ← 1	IPI Receive
invlpg [x]	Ld [x] → 0	Save state
Send IPI	Ld PML4E (x)	disable ints
Wait for Acks	Ld PDPTE (x)	invlpg [x]
Ld PML4E (x)	Ld PDE (x)	Send ACK
Ld PDPTE (x)	Ld PTE (x)	iret
Ld PDE (x)		
LdAtomic PTE (x) → clean		
StAtomic PTE (x) ← dirty		
St [x] ← 1		
Ld [y] → 0		

Depicted outcome permitted

(c)

Figure 2. Traditional litmus tests are expanded into enhanced litmus tests (ELTs). (a) A traditional litmus test with an `mprotect` system call added. (b) The user+kernel version of the litmus test. On core 1, threads 1a and 1b will be interleaved dynamically. "R/W" indicates that the page table entry (PTE) R/W bit will be set. (c) The ELT. Page table accesses for `[y]`, accessed bit updates, and so forth are not depicted but will be included in the analysis.

System Model Case Study

In this section, we present an in-depth case study of how hardware and software designers can use COATCheck and μspec to model a high-performance out-of-order processor and OS. Our case study has three parts. The first is a μspec model called *SandyBridge* that describes an out-of-order processor based on public documentation of and patents relating to Intel's Sandy Bridge microarchitecture.

The second is the microarchitecture synopsis, which specifies how ghost instructions such as page table walks behave on SandyBridge. The third is an OS synopsis inspired by Linux's implementations of system calls and interrupt handlers. We offer in-depth model highlights in this article; see our full paper for additional detail.[12]

Memory Dependency Prediction and Disambiguation

SandyBridge uses a sophisticated, high-performance virtually and physically addressed store buffer (SB). This decision was intentional: a virtual-only SB would be unable to detect virtual address synonyms, whereas a physical-only SB would place the TLB onto the critical path for SB forwarding. The *SandyBridge* SB instead splits the forwarding process into two parts: a prediction stage tries to preemptively anticipate physical address matches, and a disambiguation stage later ensures that all predictions were correct. This pairing keeps the TLB off the critical path without giving up the ability to detect synonyms.

The mechanism works as follows. All stores write their virtual address and data into the SB in parallel with accessing the TLB. Once the TLB provides it, the physical address is written into the SB as well. Each load, in parallel with accessing the TLB, writes the lower 12 bits (the "index bits") of its virtual address into a CAM-based load buffer storing uncommitted loads. The load then compares its index bits against those of all older stores in the SB. If an index match is found, the load then compares its virtual tag, and potentially its physical tag, against the stores. If there is a tag match, the youngest matching store forwards its data to the load. If no match is found, the load proceeds to the cache. If there is an empty slot because the load executed out of order before an earlier store, then the load predicts that there is no dependency. This prediction is later checked during disambiguation: before each store commits, it checks the load buffer to see if any younger loads matching the same physical address have speculatively executed before it. If so, it squashes and replays those mispredicted loads.

The following μspec snippet shows a portion of the SandyBridge μspec model capturing a case in which a load has an index match, a virtual tag miss, and a physical tag match with a previous store.

```
DefineMacro "StoreBufferForwardPTag":
exists microop "w", (
        SameCore w i /\IsAnyWrite w /\ProgramOrder w i /\
        SameIndex w i /\~(SameVirtualTag w i) /\
        SamePhysicalTag w i /\SameData w i/\
        EdgesExist [
                ((w, SB-VTag/Index/Data), (i, LB-SB-IndexCompare),
                        "SBEntryIndexPresent");
                ((w, SB-PTag), (i, LB-SB-PTagCompare), "SBEntryPTagPresent");
                ((i, SB-LB-DataForward), (w, (0, MemoryHierarchy)),
                        "BeforeSBEntryLeaves");
                ((i, LB-SB-IndexCompare), (i, LB-SB-VTagCompare), "path");
                ((i, LB-SB-VTagCompare), (i, LB-SB-PTagCompare), "path");
                ((i, LB-PTag), (i, LB-SB-PTagCompare), "path");
                ((i, LB-SB-PTagCompare), (i, SB-LB-DataForward), "path");
                ((i, SB-LB-DataForward), (i, WriteBack), "path")
        ] /\
        ExpandMacro STBNoOtherMatchesBetweenSrcAndRead

).
```

The first set of predicates narrows the axiom down to apply to the scenario we described earlier. The edges listed in the `EdgesExist` predicate then describe the associated memory ordering constraints. The first three ensure that write `w` is still in the SB when load `i` searches for it, and the rest describe the path that `i` itself takes through the microarchitecture. Finally, the axiom also checks (using a macro defined elsewhere) that the store is in fact the youngest matching store in the SB.

Other Model Details

A second component of our SandyBridge model reflects the functionality of system calls and interrupts as they relate to memory mapping and remapping functions. Although x86 TLB lookups and page table walks are performed by the hardware, x86 TLB coherence is OS-managed. To support this, x86 provides the privileged `invlpg` instruction, which invalidates the local TLB entry at a given address, along with support for interprocessor interrupts (IPIs). As a serializing instruction, `invlpg` forces all previous instructions to commit and drains the SB before fetching the following instruction. `invlpg` also ensures that the next access to the virtual page invalidated will be a TLB miss, thus forcing the latest version of the corresponding page table entry to be brought into the TLB.

To capture IPIs and `invlpg` instructions, our Linux OS synopsis expands the system call `mprotect` into code snippets that update the page table, invalidate the now-stale TLB entry on the current core, and send TLB shootdowns to other cores via IPIs and interrupt handlers that execute `invlpg` operations on the remote cores. The *SandyBridge* microarchitecture synopsis captures interrupts by adding ghost instructions that represent the reception of the interrupt and the hardware disabling of nested interrupts before each interrupt handler. All possible interleavings of the interrupt handlers and the threads' code are considered. Figures 2b and 2c depict the effects of both of these synopses.

To model TLB occupancy, the *SandyBridge* μspec model adds two nodes to the μhb graph to represent TLB entry creation and invalidation, respectively. These are then

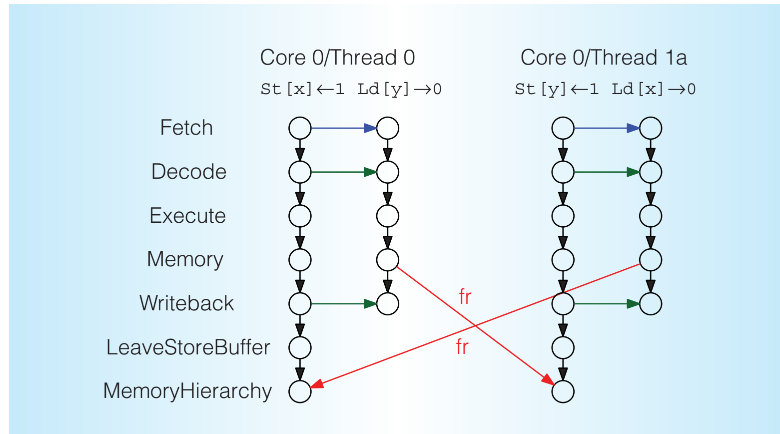
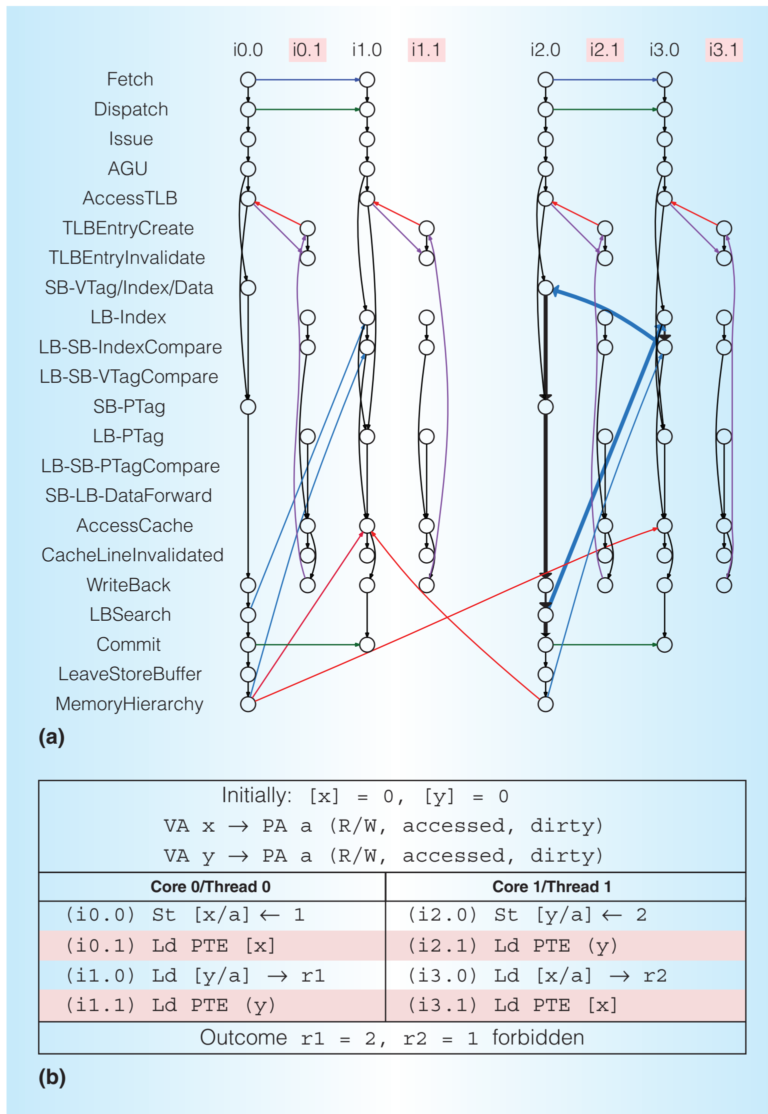


Figure 3. A μhb graph for the litmus test in Figure 1 (minus the `mprotect`), executing on a simple five-stage out-of-order pipeline. Because the graph is acyclic, the execution is observable.

constrained following the value-in-cache-line (ViCL) mechanism.[9] All loads and stores (including ghost instructions) are constrained by the model to access the TLB within the lifetime of some matching TLB entry.

Page table walks are also instantiated by the microarchitecture synopsis as a set of ghost instruction loads of the page table entry. Because these are generated by dedicated hardware, the SandyBridge μspec model does not draw nodes such as Fetch and Dispatch for these instructions, because they do not pass through the pipeline. Furthermore, because the page table walk loads are not TSO-ordered, they do not search the load buffer. They are, however, ordered with respect to `invlpg`.

Our SandyBridge model also captures the accessed and dirty bits present in the page table and TLB. When an accessed or dirty bit needs to be updated, the pipeline waits until the triggering instruction reaches the head of the reorder buffer. At that point, the processor injects microcode (modeled via ghost LOCKed read-modify-write [RMW] instructions) implementing the update. The ghost instructions in a status bit update do traverse the Dispatch, Issue, and Commit stages, unlike the ghost page table walks, because the status bit updates do propagate through most of the pipeline and affect architectural state. The model also uses μhb edges to ensure that the update is ordered against all other instructions.



Initially: `[x] = 0, [y] = 0`
`VA x → PA a (R/W, accessed, dirty)`
`VA y → PA a (R/W, accessed, dirty)`	

Core 0/Thread 0	**Core 1/Thread 1**
`(i0.0) St [x/a] ← 1`	`(i2.0) St [y/a] ← 2`
`(i0.1) Ld PTE [x]`	`(i2.1) Ld PTE (y)`
`(i1.0) Ld [y/a] → r1`	`(i3.0) Ld [x/a] → r2`
`(i1.1) Ld PTE (y)`	`(i3.1) Ld PTE [x]`
Outcome `r1 = 2, r2 = 1` forbidden	

Figure 4. Analyzing litmus test `n5` with COATCheck. (a) The μhb graph, with the cycle shown with thicker edges. (b) The ELT code.

Analysis and Verification Examples

In this section, we present three test cases for our SandyBridge model.

Store Buffer Forwarding

Test `n5` (see Figure 4) checks the SB's ability to detect synonyms. If a synonym is misdetected, one of the loads (`i1.0` or `i3.0`) might be allowed to bypass the store (`i0.0` or `i2.0`) before it, leading to an illegal outcome. Also pictured are the TLB access ghost instructions associated with each ISA-level instruction. Figure 4a shows one of the μhb graphs COATCheck uses to rule out such a situation on SandyBridge. Figure 4b shows the code itself. If load (`i3.0`) executes out of order, it finds that the SB contains no previous entries with the same index; this is captured by a μhb edge between (`i3.0`, `LB-SB-IndexCompare`) and (`i2.0`, `SB-VTag/Index/Data`). However, when the store (`i2.0`) does eventually execute, it will squash (`i3.0`) unless the load buffer has no index matches—that is, if (`i3.0`) has not yet entered the load buffer. The μhb edge from (`i2.0`, `LBSearch`) back to (`i3.0`, `LB-Index`) completes the cycle, which rules out the execution.

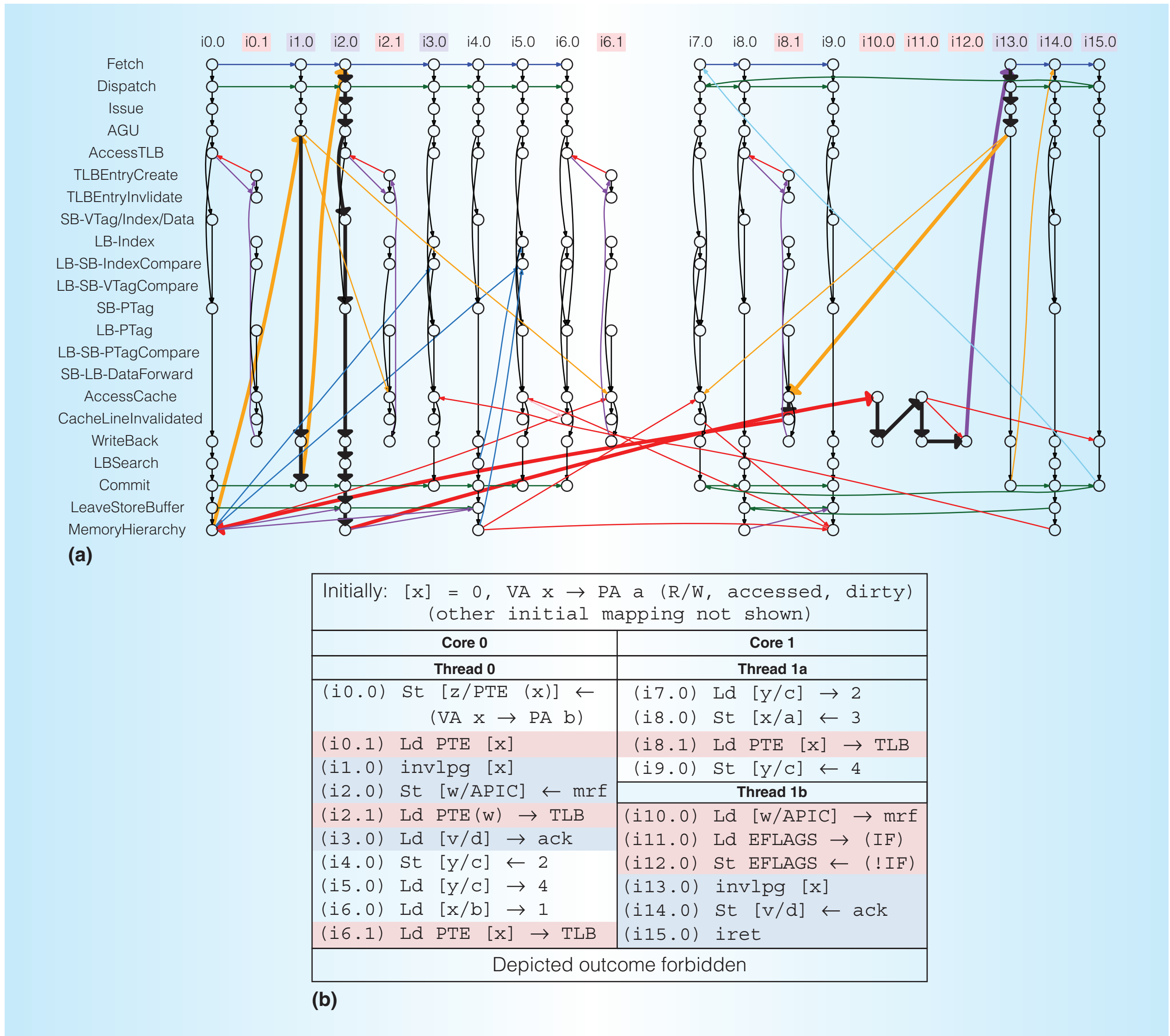
Page Remappings

Figure 5 reproduces and extends the key example studied by Bogdan Romanescu and colleagues:[7] thread 0 changes the mapping for `x` (`i0.0`), triggers a TLB shootdown (`i2.0`), and sends a message to thread 1 (`i4.0`). Thread 1 receives the message (`i7.0`) and is hypothesized to write to `x` (`i8.0`) using the old, stale mapping (a situation COATCheck should be expected to rule out). Thread 1 (`i9.0`) sends a message back to thread 0 (`i5.0`), which checks (`i6.0`) that the value at `x` (according to the new mapping) was not overwritten by the thread 1 store (`i8.0`), which used the old mapping. The μhb graph generated for this scenario (Figure 5a) is also cyclic, showing how COATCheck does in fact rule out the execution of Figure 5b. The graph also simultaneously demonstrates many COATCheck features, such as IPIs, handlers, microcode, and fences, and it shows COATCheck's ability to scale up to large and highly nontrivial problems.

Transistency versus Consistency

Our third example focuses on status bits and synonyms. Status bits are tracked per virtual-to-physical mapping rather than per physical page, and so the OS is responsible for tracking the status of synonyms. In this example, suppose the OS intends to swap out to disk a clean page that is a synonym of some dirty page. If it fails to check the status bits for that synonym, it might think that the page is clean and hence that it can be safely swapped out without being first written back.

Notably, in this example, the bug may be observable even when there is no reordering



Initially: [x] = 0, VA x → PA a (R/W, accessed, dirty)
(other initial mapping not shown)

Core 0	Core 1
Thread 0	**Thread 1a**
(i0.0) St [z/PTE (x)] ← (VA x → PA b)	(i7.0) Ld [y/c] → 2
	(i8.0) St [x/a] ← 3
(i0.1) Ld PTE [x]	(i8.1) Ld PTE [x] → TLB
(i1.0) invlpg [x]	(i9.0) St [y/c] ← 4
(i2.0) St [w/APIC] ← mrf	**Thread 1b**
(i2.1) Ld PTE(w) → TLB	(i10.0) Ld [w/APIC] → mrf
(i3.0) Ld [v/d] → ack	(i11.0) Ld EFLAGS → (IF)
(i4.0) St [y/c] ← 2	(i12.0) St EFLAGS ← (!IF)
(i5.0) Ld [y/c] → 4	(i13.0) invlpg [x]
(i6.0) Ld [x/b] → 1	(i14.0) St [v/d] ← ack
(i6.1) Ld PTE [x] → TLB	(i15.0) iret

Depicted outcome forbidden

Figure 5. Litmus test `ipi8`.[7] (a) Because the graph is cyclic (thick edges), the outcome is forbidden. In this case, the cycle was found before the PTEs for *y* were even enumerated. (b) The ELT code.

of any kind taking place, even under virtual- and/or physical-address sequential consistency.[7] Because the checks of the two synonym page mappings are to different virtual and physical addresses, the necessary ordering cannot even be described by VAMC. This example highlights a key way in which transistency models are inherently broader in scope than consistency models.

We tested COATCheck on 118 litmus tests, many of which come from Intel and AMD manuals and from prior work,[1] and others that are handwritten to stress the SandyBridge model (including the case studies discussed earlier). On an Intel Xeon E5-2667-v3 CPU, all 118 tests completed in fewer than 100 seconds, and many were even faster. Although these μhb graphs are often an order of magnitude larger than those studied by prior tools analyzing μhb graphs,[8,9] the runtimes are similar. This demonstrates the benefits of combining the μspec DSL with an efficient dedicated solver. It also points to the feasibility of providing transistency verification fast enough to support interactive design and debugging.

With COATCheck, we were able to successfully identify, model, and verify a number of interesting scenarios at the intersection of memory consistency models and address translation. However, many important challenges remain; COATCheck only scratches the surface of the complete set of phenomena that can arise at the OS and microarchitecture layers. For example, a natural next step might be to extend COATCheck to model virtual machines and hypervisors of arbitrary depth. Generally, we hope and expect that future work in the area will build on top of COATCheck to create more complete and more rigorous transistency models that can capture an ever-growing set of system-level behaviors and bugs.

We also envision COATCheck becoming more integrated with top-to-bottom memory ordering verification tools. We hope that one day verification tools will cohesively span the full computing stack, from programming languages all the way down to register transfer level, thereby giving programmers and architects much more confidence in the correctness of their code and systems. These goals will only become more challenging as systems grow more heterogeneous and more complex over time. However, COATCheck provides a rigorous and scalable roadmap for understanding how such systems can be understood rigorously, and as such we hope that future work finds COATCheck and its μspec modeling language to be useful building blocks for continued research into the area. MICRO

References

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*, vol. 36, no. 2, 2014; doi:10.1145/2627752.
2. M. Batty et al., "Clarifying and Compiling C/C++ Concurrency: From C++ 11 to POWER," *Proc. 39th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, 2012, pp. 509–520.
3. H.-J. Boehm and S.V. Adve, "Foundations of the C++ Concurrency Memory Model," *Proc. 29th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2008, pp. 68–78.
4. S. Sarkar et al., "Understanding POWER Multiprocessors," *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2011, pp. 175–186.
5. P. Sewell et al., "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Comm. ACM*, vol. 53, no. 7, 2010, pp. 89–97.
6. M. Clark, "A New, High Performance x86 Core Design from AMD," Hot Chips 28 Symp., 2016; www.hotchips.org/archives/2010s/hc28.
7. B. Romanescu, A. Lebeck, and D.J. Sorin, "Address Translation Aware Memory Consistency," *IEEE Micro*, vol. 31, no. 1, 2011, pp. 109–118.
8. D. Lustig, M. Pellauer, and M. Martonosi, "Verifying Correct Microarchitectural Enforcement of Memory Consistency Models," *IEEE Micro*, vol. 35, no. 3, 2015, pp. 72–82.
9. Y. Manerkar et al., "CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface," *Proc. 48th Int'l Symp. Microarchitecture*, 2015, pp. 26–37.
10. Check Verification Tool Suite; http://check.cs.princeton.edu.
11. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, vol. 28, no. 9, 1979, pp. 690–691.
12. D. Lustig et al., "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," *Proc. 21st Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 233–247.

Daniel Lustig is a research scientist at Nvidia. His research interests include computer architecture and memory consistency models. Lustig received a PhD in electrical engineering from Princeton University, where he performed the work for this article. He is a member of IEEE and ACM. Contact him at dlustig@nvidia.com.

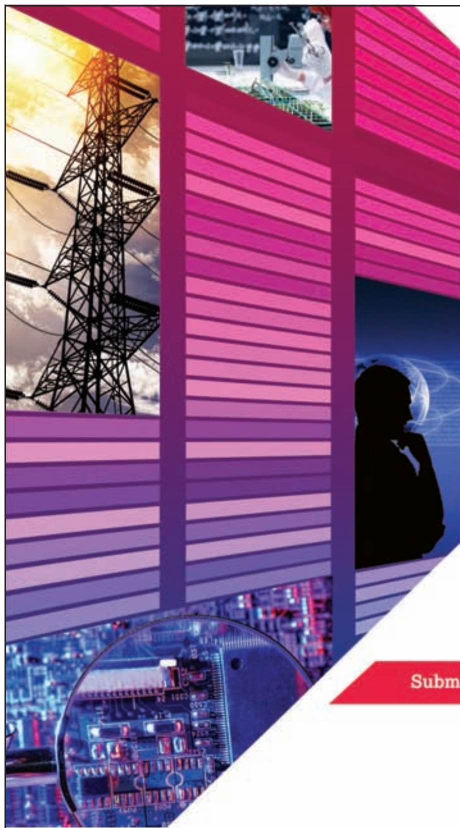
Geet Sethi is a PhD student in the Department of Computer Science at Stanford

University. His research interests include serverless computing, machine learning, and computer architecture. Sethi received a BS in computer science and mathematics from Rutgers University, where he performed the work for this article. He is a student member of IEEE and ACM. Contact him at geet@cs.stanford.edu.

Abhishek Bhattacharjee is an associate professor in the Department of Computer Science at Rutgers University. His research interests span the hardware–software interface. Bhattacharjee received a PhD in electrical engineering from Princeton University. He is a member of IEEE and ACM. Contact him at abhib@cs.rutgers.edu.

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

myCS Read your subscriptions through the myCS publications portal at http://mycs.computer.org.



CALL FOR STANDARDS AWARD NOMINATIONS

IEEE COMPUTER SOCIETY HANS KARLSSON STANDARDS AWARD

A **plaque** and **$2,000 honorarium** is presented **in recognition of outstanding skills and dedication to diplomacy, team facilitation, and joint achievement in the development or promotion of standards** in the computer industry where individual aspirations, corporate competition, and organizational rivalry could otherwise be counter to the benefit of society.

NOMINATE A COLLEAGUE FOR THIS AWARD!

DUE: 15 OCTOBER 2017

- Requires 3 endorsements.
- Self-nominations are not accepted.
- Do not need IEEE or IEEE Computer Society membership to apply.

Submit your nomination electronically: awards.computer.org | Questions: awards@computer.org