COATCheck: Verifying Memory Ordering at the Hardware-OS Interface

Daniel Lustig *
Princeton University
dlustig@nvidia.com †

Geet Sethi *
Rutgers University
geet.sethi@rutgers.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

Abhishek Bhattacharjee
Rutgers University
abhib@cs.rutgers.edu

Abstract

Modern computer systems include numerous compute elements, from CPUs to GPUs to accelerators. Harnessing their full potential requires well-defined, properly-implemented memory consistency models (MCMs), and low-level system functionality such as virtual memory and address translation (AT). Unfortunately, it is difficult to specify and implement hardware-OS interactions correctly; in the past, many hardware and OS specification mismatches have resulted in implementation bugs in commercial processors.

In an effort to resolve this verification gap, this paper makes the following contributions. First, we present COATCheck[1], an address translation-aware framework for specifying and statically verifying memory ordering enforcement at the microarchitecture and operating system levels. We develop a domain-specific language for specifying ordering enforcement, for including ordering-related OS events and hardware micro-operations, and for programmatically enumerating happens-before graphs. Using a fast and automated static constraint solver, COATCheck can efficiently analyze interesting and important memory ordering scenarios for modern, high-performance, out-of-order processors. Second, we show that previous work on Virtual Address Memory Consistency (VAMC) does not capture every translation-related ordering scenario of interest, and that some such cases even fall outside the traditional scope of consistency. We therefore introduce the term *transistency model* to describe the superset of consistency which captures all translation-aware sets of ordering rules.

* The first two authors contributed equally to this paper.

† The work was performed while this author was at Princeton University.

[1] Consistency Ordering and Address Translation Checker

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2872362.2872399

1. Introduction

Computer systems are becoming increasingly complex, with multiple processing elements (e.g., multicore CPUs, GPUs, and other accelerators) running multiple layers of system software (user code, libraries, operating systems, hypervisors, etc.). These heterogeneous systems frequently enable inter-element communication by presenting the user with an abstraction of shared virtual memory, even when the underlying hardware may contain discrete physical memory blocks [11, 42]. Harnessing these systems' full potential requires careful coordination between the hardware and the OS to ensure that the memory consistency model(s) (MCMs) and address translation (AT) mechanisms are properly-implemented. Unfortunately, the ability to rigorously verify these subsystems and their interactions with each other remains a vexing problem.

Recent years have seen increased attention being paid to the need for well-defined memory models and analysis techniques at each layer of the hardware-software stack. Many architectures and programming languages have recently developed formal consistency model specifications [10, 13, 30] and tools to help analyze them [3, 40]. However, most (but not all [38, 39]) of these models ignore the implications of virtual-to-physical address translation, such as synonyms and page permission updates, on memory ordering. Furthermore, these tools cannot verify the underlying implementations of these models, leaving a verification gap within which bugs often arise.

A key challenge is that microarchitectural events (i.e., those which are not architecturally visible) and OS behavior can affect memory ordering in ways for which standard (i.e., non-translation-aware) memory consistency analysis can be fundamentally insufficient [38, 39]. Proper implementation of a memory model requires correctness to be maintained through library calls, through system calls, and through the varying and/or unpredictable behavior of the microarchitecture. Events within each of these layers interact with and affect the state of memory, and, crucially, *events within these low-level layers may behave differently from the "normal" accesses described by the formal memory model*. For example, on the x86-64 architecture, which implements the rela-

tively strong total store ordering (TSO) memory model [32], events such as page table accesses may be inherently racy: page table walks are automatically issued by hardware, can happen at any time, and are often not ordered with respect to most fences [16, 20].

No existing notion of memory consistency captures the strictest possible translation-aware set of orderings. As we show in this paper, even data-race-free programs [1], sequentially consistent machines [23], and systems obeying sequential consistency for virtual address memory consistency (SC-for-VAMC) [38, 39] can nevertheless be prone to (perhaps surprising) ordering bugs. These bugs relate to the checking of metadata which is not directly associated with the virtual *or* the physical address being accessed; this places it outside the scope of memory consistency, including VAMC. We therefore use the term *memory transistency model* to refer to any set of memory ordering rules which explicitly takes virtual-to-physical address translation issues into account, even through the extra layers of indirection needed above.

To aid in the analysis of transistency models and their implementations, this work develops techniques and tools—collectively called COATCheck—for verifying memory ordering enforcement in the context of virtual-to-physical address translation. COATCheck extends existing tools and techniques [26, 29] to allow users to reason about system calls, interrupts, microcode, and so on. The goal of COATCheck is to improve the ability to specify and verify system behaviors at an already bug-prone interface [38, 39] whose complexity is worsening with heterogeneous parallelism. Our contributions are as follows.

First, we demonstrate a comprehensive yet tractable methodology for specifying and statically verifying memory ordering enforcement at the hardware-OS interface. We develop a Domain-Specific Language (DSL) called μspec within which each component in a system (e.g., each pipeline stage, each cache, each TLB) can independently specify its own contribution to memory ordering using the languages of first-order logic and μhb graphs [26, 27, 29]. μspec extends the constraint-based approach of previous work [29] to support modeling of TLB occupancy, page table walk microcode, activities that emit memory references despite not being part of the user-level instruction stream, system calls for memory allocation (e.g., `malloc/mmap`), and interrupts (e.g., inter-processor interrupts to maintain TLB coherence). The μspec approach allows components to be swapped in and out without affecting others, thereby providing a more modular approach to memory ordering verification.

Second, we develop a fast and general-purpose constraint solver which automates the analysis of μspec specifications, thereby allowing interactive exploration of memory ordering scenarios more complex than previous tools have handled. We demonstrate the use of COATCheck (the methodology and the tool) on several case studies that highlight interesting challenges at the hardware-OS boundary: a sophisticated model of an Intel Sandy Bridge-like processor running Linux, as well as classes of translation-related bugs recently identified by processor vendors. The full toolset (the DSL, models, litmus tests, and analysis tool) is open-source and publicly available.[2]

Finally, we use COATCheck to identify cases in which transistency goes beyond the traditional scope of consistency. We demonstrate cases where even sequentially consistent (or, following recent work, SC for VAMC [38, 39]) code may be buggy due to improper handling of page table entry status bits for virtual address synonyms. Overall, our work offers formal, yet practical tools for memory ordering checking, and it broadens the very scope of memory consistency.

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) Litmus test code

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

(b) A possible execution showing how the proposed outcome is observable, *if* x and y point to different physical addresses.

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

(c) The execution is forbidden if x and y point to to the same physical address. (Only one possible interleaving is shown.)

Figure 1: The litmus test outcome is permitted if TSO is considered to apply only to virtual addresses or if x and y are not synonyms, and forbidden otherwise.

2. Overview

2.1 Background and Motivation

As motivation, consider the litmus test[3] in Figure 1a. As written, x and y appear to be distinct addresses. Under that assumption, Figure 1b shows that even a strong MCM such as sequential consistency (SC) [23] considers the proposed final values to be observable, because an event interleaving exists to achieve that value outcome. If instead, as in Figure 1c, x and y are actually synonyms (i.e., both map to the same physical address), the test is forbidden by SC, because

[2] `http://github.com/daniellustig/coatcheck`.

[3] Litmus tests are small programs testing some aspect of a MCM. Each proposes a particular outcome (i.e., the value returned by each load) and then specifies/tests whether that outcome is permitted or forbidden by the MCM's rules.

if the addresses are the same, 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 when determining expected behaviors and verifying correctness.

Relationship to Past Work: The bulk of prior MCM work has focused on the high-level programming language and hardware layers [2, 10, 13, 30, 32, 40]. However, abstractions effective at these levels (e.g., SC-for-DRF [1], TSO [43]) are often ineffective at the hardware level, as high-performance data structures and low-level hardware operations are often inherently weakly ordered and racy. For example, although x86 processors implement the Total Store Ordering (TSO) MCM, *page table walks are TSO-ordered with respect to neither normal memory accesses nor* `mfence` *instructions (which enforce orderings between all normal loads and stores)* [16, 20]. Likewise, many parallel data structures bypass software memory models in favor of higher-performance (but less portable) assembly implementations that interface more directly with the non-SC hardware memory model [10, 24, 25, 31].

Romanescu et al. were the first to distinguish between MCMs meant for virtual addresses (VAMC) and those for physical addresses (PAMC) [38, 39]. They considered hardware to be responsible for enforcing the latter, and a combination of hardware and OS for the former. Accordingly, traditional hardware models such as TSO would fall under PAMC, while synonyms, page mapping changes, and side effects (i.e., page status bits) would be added to form VAMC. COATCheck provides a rigorous means of specifying and verifying the interaction between the two models. This paper also goes beyond the VAMC-PAMC distinction to identify cases in which ordering bugs can be found even when both VAMC and PAMC are made sequentially consistent.

PipeCheck and CCICheck verify memory ordering enforcement through the use of microarchitecture-level happens-before (μhb) graphs [26, 27, 29]. PipeCheck performs MCM verification by enumerating a complete family of μhb graphs for any given litmus test. CCICheck extends PipeCheck to handle coherence-consistency interface issues. COATCheck extends both of these by providing a fully-general DSL for specifying ordering enforcement, and by demonstrating how PAMC, VAMC, and transistency in general can be analyzed using μhb graphs.

2.2 The COATCheck Approach

Figure 2 shows layers at which memory ordering issues might be considered. Our work enables the building and verification of detailed models for hardware-OS memory ordering implications, particularly focusing on layers 3, 4, and 5 in Figure 2. This allows us to analyze memory ordering in an execution stream that includes library and kernel code, as well as microcode-level events such as the hardware page table walks executed on behalf of the program

1. At the highest level: Programmer-written source code is transformed into compiler-generated low-level code.
2. After dynamic linking with libraries: Compiler-generated user-level code with user-level OS support (e.g., syscalls).
3. After addition of OS kernel work (e.g., system call handlers, interrupts): User and kernel level code.
4. After the addition of microcode: Microcode and post-ISA user and kernel code.
5. After mapping onto a given microarchitecture: Micro-ops (or low-level instructions) traversing pipeline structures.

Figure 2: In a typical system layering, our approach supports memory order verification for Levels 3, 4, and 5. OS code Insertion (Section 3.3) creates Kernel-Level Litmus Tests that lie at Layer 3. The further insertion of ghost instructions (Section 3.4) creates ELTs that lie at Layer 4. Finally, with designer input via μspec, the constraint solver and graph enumeration explore microarchitecturally-aware event orderings at Layer 5.

at TLB misses. Previously, incomplete specifications, incorrect implementations, or poor coordination between the layers could (and did [5, 6, 18, 21]) cause bugs. These include forbidden multithreaded outcomes becoming observable, legal data disappearing due to incorrect updates of page table entry dirty bits, processors experiencing deadlock/livelock, and any number of other undesirable outcomes.

COATCheck overcomes these problems by building *modular* models of memory ordering enforcement at the hardware-OS interface. In particular, each component (a pipeline stage, a page table walker, an OS mechanism, etc.) can specify its own independent contribution to memory ordering enforcement. Prior to verification, the independent contributions of the components which form the system under test are merged into a single overall specification. The COATCheck tool then uses this combined specification to generate families of μhb graphs to statically verify the overall correctness of the system.

Figure 3 shows the overall COATCheck toolflow. First, traditional (i.e., address translation-unaware) litmus tests are converted into *enhanced litmus tests* which include all (micro) code relevant to memory ordering. Second, the enhanced litmus test is analyzed according to the rules of the μspec specification of the orderings enforced at various parts of the system. This produces a set of constraints describing the conditions under which the outcome proposed by the litmus test would be observable. Third, the constraints are analyzed by the COATCheck constraint solver tool to determine whether any observable execution (in the form of a μhb graph) can be found. We analyze each step in detail in the sections that follow.

After describing the overall flow, Section 6 gives the concrete example of how a system consisting of Intel Sandy Bridge-like hardware, a Linux-like OS, and interesting litmus tests as software can be modeled and analyzed using COATCheck. This is followed with an analysis of the perfor-

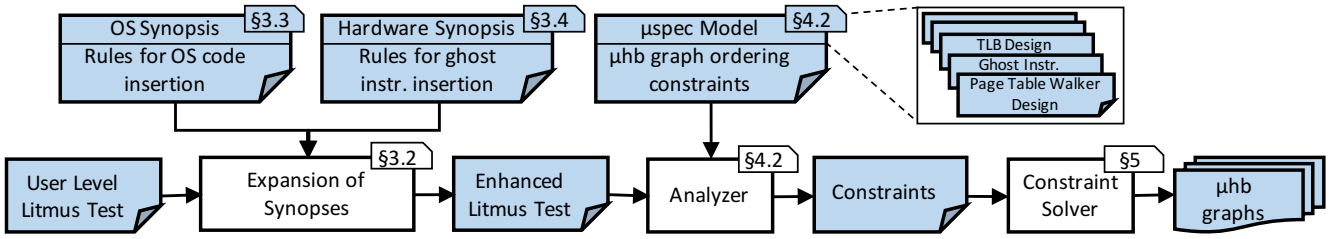


Figure 3: COATCheck flow diagram. Section numbers are listed in the top-right of the appropriate boxes.

mance of the COATCheck constraint solver, using the above system as a case study.

3. Beyond Traditional Litmus Tests

Litmus tests have become a standard tool in memory model analysis. This section describes how we derive *Enhanced Litmus Tests* (ELTs) which account for address translation operations, page table walks, remappings, memory-accessing microcode, and other operations relevant to memory ordering and other system-level operations that execute concurrently with, and on behalf of, user-level code.

3.1 Enhanced Litmus Tests: An Overview

ELTs use three key constructs in moving beyond traditional, user-level litmus tests. First, ELTs describe virtual and physical addresses as distinct and simultaneous entities. In the same way that traditional litmus tests propose an outcome consisting of a set of values returned by the loads in the program, ELTs likewise propose as part of the outcome the physical addresses used by each access. In this way, the analysis of later sections can directly test whether a proposed set of address translation outcomes is legal.

Second, ELTs incorporate relevant chunks of OS activity, such as map/remap functions (MRFs) or inter-processor interrupts (IPIs) as used in software TLB coherence [37, 46]. They similarly include events such as the sending of IPIs, register reads/writes which cause the enabling/disabling of interrupts, and so on.

Third, ELTs include "ghost instructions" which model lower-than-ISA operations (e.g., page table walks) executed by the hardware on behalf of user-level or system-level code, even if these instructions are not fetched, decoded or issued as part of the normal ISA-level instruction stream. The combined power of address tracking, ghost instructions and OS module insertions gives ELTs sufficient expressive power to test all aspects of memory ordering enforcement as it relates to address translation.

Figure 4 presents an example flow from user-level litmus test to ELT. Our example shows the *store buffering* (`sb`) litmus test, which will be further discussed in Section 7. The stated test outcome is forbidden under sequential consistency (because it does not represent any simple interleaving of the four user-level instructions), but as its name implies, it is observable under models such as TSO which allow store buffering/store→load reordering to be visible. We form a more system-oriented example by inserting an extra system call to `mprotect` at the beginning of thread 0 in Figure 4a.

3.2 Litmus Test Expansion Synopses

As Figure 4 shows, each ELT comes about as the result of a sequence of modification passes. We systematize this transformation in the form of *litmus test expansion synopses*, or simply *synopses* for short. Each synopsis is a "recipe" for expanding each instruction in an input litmus test into one or more instructions in an output litmus test. As described below, there is some flexibility in what the recipes look like: the original instruction may or may not be maintained, and instruction expansions may insert new code into other threads as well. Furthermore, each expanded instruction may be inserted either before or after the original instruction. In this way, synopses serve as succinct representations of behaviors that we hope might one day become more rigorously and precisely formalized.

3.3 From User-Level to Kernel-Aware Litmus tests

Incorporating system calls and kernel-level code into a user-level litmus test requires multiple changes. First, in the invoking thread (e.g., Thread 0 in Figure 4a), we must be able to substitute the system call with a module of memory reference operations that correctly encapsulate the system call behavior relevant to memory ordering verification. Figure 4b continues the example of Figure 4a by expanding the system call into a set of four instructions on the invoking Thread 0. The expanded regions are shaded in blue.

It is worth noting that this expansion can affect multiple threads at once. Because one of the instructions expanded in Thread 0 triggers an inter-processor interrupt (IPI), we also add a three instruction interrupt handler to all of the other threads (in this case, Thread 1a). Notably, since the relative timing of Thread 0 and Thread 1a has not yet been established, it is also impossible (at this point) to determine the relative ordering of Threads 1a and 1b. For example, while it is clear that the interrupt handler must happen sometime after the interrupt is originally triggered, there is little other synchronization between the threads. Any such orderings will be filled in later by the μspec specification of IPI behavior in the microarchitecture in question.

Initially: [x]=0, [y]=0	
Core 0/Thread 0	**Core 1/Thread 1a**
`syscall mprotect [x], r/w`	
`St [x] ← 1`	`St [y] ← 1`
`Ld [y] → 0`	`Ld [x] ← 0`
Outcome: Permitted	

(a) Original user-level test.

Initially: [x]=0, [y]=0, VA x → PA a (R/O, acc, !dirty), VA y → PA b (R/W, acc, dirty), VA z → PTE for VA x (R/W, acc, dirty)		
Core 0	**Core 1**	
User Thread 0	**User Thread 1a**	**Int. Hndlr. Thread 1b**

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

`St [x/a] ← 1` `Ld [y/b] → 0`		
Outcome: Permitted		

(b) User+Kernel litmus test. On core 1, threads 1a and 1b will be interleaved in some fashion, but the interleaving cannot be statically determined. The "R/W" store value is shorthand to indicate that the entire PTE is written, but with the R/W bit set.

Initially: [x]=0, [y]=0, VA x → PA a (R/O, acc, !dirty), VA y → PA b (R/W, acc, dirty), VA z → PTE for VA x (R/W, acc, dirty) VA w → PTE for VA y (R/W, acc, dirty)		
Core 0	**Core 1**	
User Thread 0	**User Thread 1a**	**Int. Hndlr. Thread 1b**

`St [z/PTE(x)] ← R/W`
`invlpg [x]`
`Send IPI`
`Wait for IPI Acks`
	`St [y/b] ← 1`
	`Ld PML4E(x)`
	`Ld PDPTE(x)`
	`Ld PDE(x)`
	`Ld PTE(x)`
	`Ld [x/a] → 0`
		`IPI Recv`
		`save state`
		`disable ints`
		`invlpg [x]`
		`Send Ack`
		`iret`

`Ld PML4E(x)`
`Ld PDPTE(x)`
`Ld PDE(x)`
`Ld PTE(x)→clean`
`LdAtomic PTE(x)→clean`
`StAtomic PTE(x)←dirty`		

`St [x/a] ← 1` `Ld [y/b] → 0`		
Outcome: Permitted		

(c) Microarchitecture-level litmus test (ELT). However, page table accesses for [y], accessed bit updates, etc., are not even expanded in this example; the real situation would potentially grow even larger.

Figure 4: To represent the memory accesses taking place at the microarchitecture level, we expand user-level litmus tests into *enhanced litmus tests*.

OS Synopses: On the invoking core, an *OS synopsis* specifies a mapping from each system call into a simple pre-defined sequence of microops capturing the effects of that system call on address translation and consistency. When the system call contains an inter-processor interrupt (IPI), the OS synopsis also instantiates pre-defined interrupt handler threads on those cores, again using a sequence of statically-determined instructions. Each interrupt handler may be arbitrarily interleaved with the other thread(s) assigned to that core, subject to μspec constraints (Section 4.2).

We do not curently model the complexities of OS decision making or data structures; our OS synopses currently include only memory accesses which update the paging structures and any synchronization used to enforced orderings with respect to these updates. However, these synopses could be made more sophisticated as needed; we believe that attempting to formalize the relationship between these OS synopses and the full OS code would make for interesting follow-up work.

3.4 Memory-Accessing Microcode: Ghost Instructions

During V-P translation, there are many microcode operations which are *not* fetched as ISA-level instructions (either user or kernel) but which still play a key role in enforcing consistency. These microcode operations are used to support hardware page table walks, TLB refills, accessed/dirty bit updates, and so on. We refer to these operations as *ghost instructions*, as they are present but not visible to the user or to the OS kernel.

The presence and behavior of ghost instructions depends heavily both on the architecture and on the microarchitecture in question. At an architecture level, operations such as page table walks may be specified as being enforced entirely by hardware, entirely by software, or anywhere in between. COATCheck is flexible enough to cover any point on this spectrum. We break this problem into two parts: the specification of the instructions (and ghost instructions) which are emitted to cause ordering to be enforced (this section), and the specification of the orderings enforced between these instructions at different points in their executions (Section 4.2).

Figure 4c depicts an ELT derived from Figure 4b. Darker red-shaded regions are microcode operations that have been expanded at this phase; lighter blue regions remain those expanded in the previous step. For this test scenario, thread 0's access to `[x]` requires a page table walk, because the TLB entry for that virtual address would have been invalidated by the `invlpg` instruction. Also, since the initial condition states that the page containing `[x]` is clean, hardware would also mark the page as dirty prior to the write (specified to occur on x86 using a LOCKed atomic operation [20]). Other accesses may also take TLB misses and trigger page table walks themselves, although (for space reasons) the figure does not show all of them. Finally, the ELT includes hardware operations for receiving the interrupt, saving state, and disabling nested interrupts via the microcode preamble to thread 1b. 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.

Microarchitecture Synopses: As with the OS synopses, our *microarchitecture synopses* currently consist of rela-

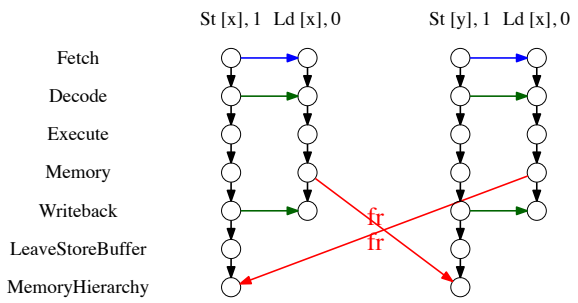


Figure 5: μhb graph for the test of Fig. 4a, without the syscall.

tively straightforward pre-defined expansions which insert page table walk ghost instructions next to each normal instruction. Unlikely as it may be, for completeness we perform this expansion for every instruction, as each instruction may cause a TLB miss and hence trigger a page table walk, and every instruction may update the associated page status bits. Likewise, the expansions of loads in our current synopses include ghost instructions corresponding to accessing both clean and dirty pages, as both possibilities must be considered. The comprehensive enumeration performed by the μspec models may choose to use all ghost instructions, or it may choose to ignore optional possibilities which are not needed (e.g., PT walks not needed during TLB hits) or impossible (e.g., a page being clean and dirty simultaneously).

As with the OS synopses, we envision microarchitecture synopses becoming more sophisticated as needed. Further, we envision that the notion of such synopses will ultimately become part of future microarchitectural specifications too.

4. μspec: A μhb Graph-Centric Ordering Specification Language

4.1 Background: μhb Graphs

Microarchitecture-level happens-before (μhb) graphs capture consistency model enforcement at the implementation level [26, 27]. Figure 5 gives an example. Nodes in a μhb graph represent events corresponding to a particular instruction (column) at a particular physical location (row). For example, a node may represent a particular instruction passing through a particular pipeline stage. Edges represent happens-before orderings guaranteed by some property of the microarchitecture: an instruction flowing through a pipeline, a FIFO-ordered structure, the passage of a message, and so on.

All edges in a μhb graph are equivalent, regardless of their label. This allows the transitive closure of two μhb graphs to itself always be a legal μhb graph, and it implies that *any* cycle in a μhb graph indicates that the execution is infeasible. This places μhb graphs in contrast to some other existing approaches in which only specific subsets of a happens-before graph are analyzed for cycles [3].

```
Axiom "ReadsFrom":
forall microops "r", IsRead r =>
  exists microop "w", IsWrite w /\
  SamePhysicalAddress w r /\ SameData w r /\
  EdgeExists ((w, AccessMemory), (r, AccessMemory), "rf").
```

Figure 6: μspec syntax example.

4.2 μspec: A DSL for Specifying μhb Graph Orderings

μspec is a domain specific language in which hardware or software component designers express the ordering relationships enforced by that component. Through the μspec specification, designers provide a set of μhb graph-focused *axioms* which must all be satisfied for a given test to be considered observable [29]. This axiomatic approach is widely used in memory model analysis [2, 3, 28, 32], but μspec extends this down to a lower level of abstraction.

μspec axioms are first order logic (i.e., AND, OR, NOT, EXISTS, FORALL) statements built on top of instruction- and μhb graph-related predicates. The instruction-related predicates are used to constrain an axiom to a relevant subset of instructions. For example, an axiom that states a property relating two reads to the same physical address would use the two predicates `IsRead` and `SamePhysicalAddress`.

The core μspec axiom is the predicate `EdgeExists` (or, in plural form, `EdgesExist`). This predicate takes three arguments: a source μhb node, a destination μhb node, and a label. The nodes are in turn formed of two components: an instruction, and a microarchitectural location or event. These correspond to the column and row of the node in the μhb graph, respectively. The edge label is purely cosmetic, as all edges in a μhb graph are equivalent (Section 4.1).

Figure 6 gives an example of the μspec syntax. The `ReadsFrom` axiom asserts that for every read `r`, there must be some corresponding write `w` to the same *physical* address. Further, this write must match the value returned by the read. When such a read is found, the axiom specifies that an edge labeled "rf" should be added to the μhb graph between the two μhb nodes (w, AccessMemory) and (r, AccessMemory). If there is more than one candidate for w which satisfies `SamePhysicalAddress` and `SameData`, then each possibility (and sub-cases derived thereof) will be considered independently by the constraint solver (Section 5).

Each axiom in a μspec model specification represents an independent and ideally localized property of one particular mechanism. Component models are formed by conjoining (i.e., logically ANDing) the axioms within a model. Likewise, system models are built by conjoining the models of the components forming the system. The overall system specification therefore consists of a first order logic formula whose satisfiability determines whether a proposed execution is feasible on the system being analyzed.

The approach of separating out individual axioms makes μspec models highly modular: axioms can be added or removed as necessary without affecting any of the other ax-

ioms or components. In other words, this approach allows microarchitecture-specific axioms to be easily swapped out for another set while OS- and architecture-level are kept unchanged, to give just one example. While μspec cannot (yet) automate the extraction of a specification from Verilog or HDLs, it does greatly reduce the effort required to express ordering requirements and expectations, and we hope to elaborate on this aspect in future work.

5. Constraint Solver and Software Implementation

As described in Section 4.2, the μspec specification of a system produces a first-order logic formula whose satisfiability corresponds to the feasibility of a particular execution. The COATCheck constraint solver accepts this formula and some litmus test as inputs, and it searches to find any execution of that test which satisfies all of the constraints of the model. If one can be found, then the proposed outcome is observable. If not, then the proposed outcome is forbidden. In PipeCheck, which introduced μhb graphs, the graphs were generated using naive exhaustive enumeration [26, 27]. However, the PipeCheck approach does not scale to the sizes and numbers of graphs needed to handle ELTs.

5.1 Constraint Solver Algorithm

The preliminary step in the solving process is to eliminate the quantifiers in the formula. Doing so produces a quantifier-free propositional logic formula that is more directly amenable to being solved. Since the domain of each quantifier is concrete in the context of some particular litmus test, the quantifiers are removed by simply converting each `forall` into a conjuction (AND) over its domain (i.e., cores, threads, or instructions). Likewise, each `exists` is converted into a disjunction (OR).

The solver algorithm itself resembles and is inspired by the Davis-Putnam-Logemann-Loveland (DPLL) algorithm widely used in SAT solvers [14]. We apply it to μhb graphs, making our solver resemble a primitive but effective SMT solver for the theory of acyclic directed graphs. At a high level, the solver uses a backtracking approach: given a starting point, it generates a list of subcases of "either-or" edge additions, and it then recursively descends into each subcase, abandoning those which cannot satisfy the given conditions and stopping when it reaches a leaf node (i.e., an acyclic graph). Although we could have used an off-the-shelf SAT or SMT solver, our custom solver provided significantly better debugging ability and status visualization (e.g., of partially-completed μhb graphs and decision trees) than would have been possible with a black-box solver. We found this empirically to be very useful during our work.

5.2 Software Toolchain

We implemented the COATCheck methodology into the complete working toolflow shown in Figure 3. In normal operation, the tool takes as input a user-level litmus test, HW/OS synopses (Section 3.1), and μspec models (Section 4.2). For low-level debugging, the tool also allows the direct input of a manually-written ELT (bypassing the synopses). The ELT and component μspec models feed into the model analyzer, which applies the axioms of the models to the ELT to generate a tree of μhb graph constraints. The constraints are then passed to the constraint solver to determine if the outcome is feasible.

Our core analysis infrastructure is written in Coq to allow for formal verification [44]. However, we have not yet completed any formal proofs of correctness; this remains an open problem, and in any case, the specifications of expected behavior often simply do not yet exist. We therefore leave this to future work, and we instead extract the Coq code to OCaml to build COATCheck as a standalone tool.

Because we aim for backwards compatibility with existing MCM analysis frameworks, we interface our tool with `herd` and the `litmus` format [3]. This allows us to draw from a large existing body of litmus tests. Although these tests do not distinguish virtual and physical addresses, they serve as valuable sanity checks for the basic correctness of a pipeline model. Furthermore, hardware models from previous work [26, 27, 38, 39] can be easily adapted into μspec and our OS models, and hence into COATCheck as well.

6. Detailed Processor+OS Model Case Study

This section presents an in-depth case study of how hardware and software designers might use COATCheck and μspec to model a high-performance out-of-order processor and OS, respectively. The resulting μspec model is the one used for Section 7's litmus test case studies, and Section 8 performance results.

6.1 Basic Overview

Our case study model has three main parts, two of which are provided by the hardware designer and one of which is provided by someone familiar with the OS. The first component is a μspec model which describes a given processor microarchitecture. This model provides a set of μspec axioms representing the ordering constraints enforced by the hardware. In this case study, this hardware component is inspired by the Intel Sandy Bridge microarchitecture, and was developed in detail using public documentation [19, 20], information gleaned from patents [16], and some educated guesses used to fill in gaps. Many of the low-level details remain proprietary, so it cannot be an exact match, but the paper nevertheless refers to this as our *SandyBridge* model. Table 1 gives an overall enumeration of the μspec axioms in this model.

The second component is a SandyBridge hardware synopsis (Section 3.4) which specifies how litmus tests might be expanded by hardware when executed on SandyBridge. These expansions pertain to hardware page table walks or other hardware-initiated events (i.e., ghost instructions) in-

Axiom	Description
Reads	Path (Pipeline stage sequence) and μhb orderings for read instructions
Writes	Path/orderings for write instructions
mfence	Path/orderings for `mfence`
invlpg	Path/orderings for `invlpg`
iret	Path for `iret` instruction
RMW	Atomicity of LOCKed RMW operations
FetchPO	Program Order enforced at Fetch
DispatchPPO	Fetch order maintained at Dispatch
CommitPPO	Dispatch order maintained at Commit
StBufPPO	Commit order maintained at Store Buffer
Write Serialization	Per-physical address total order on all writes reaching `AccessCache`
SLR	Speculative Load Reordering
IPIInsertions	IPI handlers embedded within user thread
IPIOrdering	Enumeration of all nestings of IPI handlers
IPIReceive	Paths for OS code modeling receiving of IPI
IPIRecvAtomicity	OS code modeling receving of IPI is atomic
IPIAcks	IPI handlers must complete before issuing thread allowed to proceed
TLBEntries	Paths for PT walks and TLB entry μhb nodes
TLBEntriesNoDups	No concurrent duplicate TLB entries

Table 1: Axioms for SandyBridge model. Some axioms include macros which expand to address other orderings not listed.

serted within the litmus test. The third component is the Linux OS synopsis (Section 3.3) that specifies code sequences for system calls and for interrupt handlers. Our Linux synopsis is derived from careful inspection of the Linux source code (and, for the architecture-specific portions, the 64-bit x86 version). Note that there is no μspec model for the Linux component; this is because the software specifies the instructions to be executed (i.e., the column headings in a μhb graph), while the hardware actually executes the instructions (i.e., the nodes and edges in the graph).

In general, the μspec model and hardware synopsis rely on hardware knowledge, and the OS synopsis relies on OS knowledge, but the three components compose together; no single designer must be fluent across all components. In the text that follows, we discuss several important-to-model aspects of hardware-OS behavior. For each, we describe how the μspec model, the hardware synopsis and the OS synopsis work together to reflect the appropriate hardware-OS behavior.

6.2 Memory Dependency Prediction and Disambiguation

The first type of functionality we consider is a sophisticated, high-performance store buffer (SB) forwarding mechanism. We focus on capturing the key role address translation plays in the store buffer. If the implementation operated using only virtual addresses, for example, then it would be unable to detect virtual address synonyms, leading to problems such as the one in Figure 7. Placing the TLB on the critical path would avoid this problem; however, this would

`x` and `y` are synonyms
Thread 0
`St [x/PA1] ← 1`
`St [y/PA1] ← 2`
`Ld [x/PA1] → r1`
Proposed outcome: `r1=1`

Figure 7: Store buffer forwarding implementations must be able to detect synonyms to ensure that each load receives its value from the latest store to the same physical address

come with a performance cost. This realistic example reveals the perhaps-surprising complexity involved in the seemingly straightforward process of store buffer forwarding. We start by describing the mechanism, and we then describe its implementation in μspec. Hardware or OS synopses are not used in modeling this feature, so they are not discussed.

Mechanism: High-performance forwarding in our SandyBridge model consists of memory dependency prediction and memory disambiguation [19]. The prediction stage anticipates dynamic same-physical address dependencies between stores and loads to try to preemptively prevent consistency violations that might arise. The disambiguation stage later ensures that all predictions were correct. This pairing ensures that synonyms can be detected while keeping the TLB off of the forwarding critical path.

The mechanism we model works as follows. All stores write their virtual address and data into the SB in parallel with accessing the TLB. The physical address is later written into the SB as well, once the TLB provides it. Loads, in parallel with accessing the TLB, write their lower 12 bits ("index bits") into a CAM-based load buffer that holds all loads that have not yet committed. Due to the minimum 4KB page size, these lower 12 bits will always be identical between virtual and physical addresses.

Initially, each load compares its index bits against the index bits of all older stores present in the store buffer. If no index match is found among the filled-in entries, then clearly no match exists. If an older entry is allocated but not filled in (because its address was not yet generated), it is predicted to cause no dependencies. If an index match is found, the load will then compare high-order bits. If the load's virtual tag matches the virtual tag of the store, the store will forward its value to the load[4]. If not, the load compares its physical tag against the store's physical tag, stalling if either instruction's TLB access has not yet returned a translation. If the physical tags match, the store will forward its data to the load. Otherwise, the load will have determined that no dependency exists between the load and that particular store.

In addition to the above, our model must also reflect disambiguation, in which stores determine whether their speculations were legal. Before each store commits, it checks the load buffer to see if any younger loads matching the same

[4] Homonyms are handled on Linux/x86 by flushing non-global TLB entries on each context switch.

```
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).
```

Figure 8: μspec Macro describing instance of SB-Forwarding

physical address have speculatively executed before it. If so, it squashes and replays them.

μspec Axioms: Based on that store buffer mechanism, Figure 8 shows a μspec model snippet of one piece of the prediction and disambiguation mechanism; other pieces are analogous but omitted for space reasons. The figure shows a μspec snippet for the sub-case in which a load has an index match, virtual tag miss, and physical tag match with a previous store.

The first part of the code, '`exists microop "w"`', will only evaluate to true if there is at least one microop that satisfies the enclosed statements. The middle part enforces orderings via two types of μhb edges: some ensuring that the write `w` is still in the store buffer when the load `i` searches for it, and some describing the path `i` itself takes through the forwarding mechanism. Finally, it expands a macro (not pictured due to space reasons) checking that the store is in fact the youngest matching store in the store buffer. The model accomplishes this by ensuring that for all stores in between, either the virtual and physical tags do not match or the indexes of the middle stores are not present (because they have not yet executed).

Other possibilities exist for memory disambiguation. An advantage of μspec is that the axioms representing the approach we describe above can be easily replaced by those representing other solutions (e.g., load reexecution [15]).

6.3 Memory Mapping/Remapping Functions, System Calls, and Interrupts

A second component of our *Linux+SandyBridge* model reflects the functionality of system calls and interrupts as they relate to memory mapping/remapping functions. This section describes mechanisms for how these work, particularly noting which parts are implemented in hardware and OS. The model also serves as an example of how key hardware-OS functionality is reflected in our SandyBridge μspec model and the hardware and OS synopses.

```
DefineMacro "EmbeddedIn" t1 t2:
forall microops "i1", OnCore c i1 => OnThread t1 i1 => (
  forall microops "i2", OnCore c i2 => OnThread t2 i2 => (
    AddEdge ((i1, Commit), (i2, Fetch), "EmbeddedIn") \/
    AddEdge ((i2, Commit), (i1, Fetch), "EmbeddedIn"))).

Axiom "IPIOrdering":
forall threads "t1", forall threads "t2",
~(IsMainThread t1) => ~(IsMainThread t2) =>
~(SameThread t1 t2) =>
  ExpandMacro EmbeddedIn t1 t2 \/
  ExpandMacro EmbeddedIn t2 t1.
```

Figure 9: Enumerating thread interleavings using μspec. The user specifies only the constraint that there exists some embedding; the solver later enumerates the possibilities.

Mechanism: 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, as well as the support for cores to communicate via inter-processor interrupts (IPIs). `invlpg` is a privileged instruction which serializes the instruction stream and invalidates the TLB entry containing the mapping corresponding to the specified virtual address. As a serializing instruction, `invlpg` causes all previous instructions to commit and drains all pending writes in the store buffer to memory 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.

Linux Synopsis: Our Linux synopsis expands the system call `mprotect` into code snippets which 1) update the PTE appropriately, 2) invalidate the now-stale TLB entry on the current core, and 3) send TLB shootdowns to other cores via IPIs and interrupt handlers. The interrupt handler performs its own `invlpg` operation before sending an acknowledgment to the sender and returning from the interrupt.

SandyBridge Synopsis: The SandyBridge synopsis reflects ghost instructions added to reflect hardware-initiated memory references. In this case, the hardware synopsis first adds a ghost instruction representing the reception of the interrupt. This allows the μspec model to draw μhb edges reflecting that operation. Second, it adds a ghost instruction representing a write to the `FLAGS` register bit that determines whether interrupts are enabled, as on x86, after receiving an interrupt, subsequent interrupts are disabled until re-enabled by the handler or the OS. Interrupt re-enabling is handled by the OS code expanding to sequences containing either `iret` or `cli` instructions; both implicitly model writes to the `FLAGS` register as well.

μspec Axioms: Figure 9 shows how the orderings between these events are specified within our `SandyBridge` μspec model. This snippet shows a macro `EmbeddedIn` and an axiom `IPIOrdering` that makes use of it. The user sim-

ply specifies the set of possibilities, and the solver (Section 5) automatically and efficiently enumerates all ways the axiom can be satisfied. A separate macro (not shown) adds pipeline- and store buffer-draining μhb edges representing the fact that interrupts are precise [19].

6.4 Page Table Walks

Mechanism: On the x86 architecture, the page table walker is built into the hardware [20]. This means that page table walk loads are not issued by the user or by the OS; instead, they are contained entirely within hardware and are invisible to the user.

SandyBridge Synopsis: Every potential page table walk is instantiated by the microarchitecture synopsis as a set of ghost instruction loads of the page table entry. (The TLB is modeled as described in Section 6.6.)

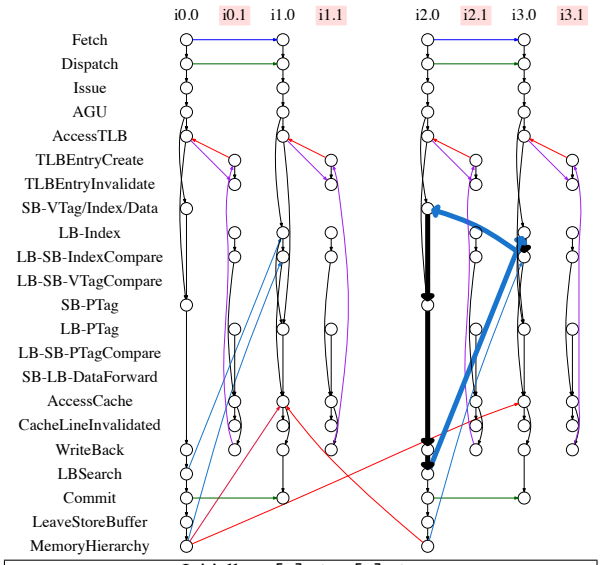
μspec Axioms: Our model handles these ghost loads as follows. First, it does not draw Fetch, Dispatch, etc. nodes in the paths (i.e., columns) for these special loads, as they do not pass through the pipeline. Second, the axiom never uses predicates such as `SameVirtualTag` with them, since page table walks are done using only physical addresses. Third, since the loads are not TSO-ordered, they do not search the load buffer as in Section 6.2. Lastly, it adds exceptions to, e.g., the `mfence` axioms to omit μhb edges touching these instructions. The model does ensure, however, that page table walks are ordered with respect to `invlpg` [16].

6.5 Status Bit Updates

Mechanism: When a page table accessed or dirty bit needs to be updated due to a memory access, our SandyBridge pipeline waits until the triggering instruction reaches the head of the reorder buffer. At that point, the processor injects microcode implementing the update into the buffer. It also must ensure that the update is ordered against younger instructions to prevent later loads from reading the now-stale state of the PTE from before the update.

SandyBridge Synopsis: For each store, the hardware synopsis instantiates dirty bit updates as ghost instructions performing a LOCKed RMW operation on the bits in memory. These instructions are inserted just before the triggering instruction. At a low level, we represent the read and the write of the RMW as separate microops, and atomicity is guaranteed by the μspec axioms described below.

μspec Axioms: The ghost instructions in a status bit update do traverse the Dispatch, Issue, and Commit stages, unlike the ghost page table walks, as the status bit updates do propagate through most of the pipeline and affect architectural state. They are not fetched, however. The orderings enforced for these ghost instructions are modeled in μspec by adding μhb edges to enforce that 1) all previous accesses must have committed, reflecting the condition that the triggering instruction be at the head of the ROB, and 2) each update has LOCK semantics (i.e., `mfence` semantics plus atomicity).



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

Core 0		Core 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: Forbidden; r1 = 2, r2 = 1

Figure 10: Litmus test `n5`. Cycle shown with thicker edges.

6.6 Modeling TLB Occupancy

μspec Axioms: To model TLB occupancy in μhb graphs, we add two special new nodes to the paths of page table walk and status bit update ghost instructions. For page table walks and status bit updates, after the last memory access of the walk completes, the insertion of the entry into the TLB corresponds to a *TLB entry creation* event. Some time later, there is a *TLB entry invalidation* event that takes place for the same entry. Note that this invalidation, in general, occurs long after the page table walk has completed. This approach, which resembles the value-in-cache-line (ViCL) mechanism developed previously [29], tracks the ordering of TLB state changes and applies equally well to microarchitectural variants such as allowing in-place updating of TLB status bits.

We then add a constraint stating that each instruction must read from a TLB entry which exists and which contains matching data (translation and status). More precisely, for a given instruction i accessing virtual address v and physical address p, this means that there must exist a ghost instruction g such that the data returned (for walks) or written (for updates) by g is a PTE mapping the tag of v to the tag of p. The relevant model axiom also checks for status and permission bits in the same way. Once the entry is found,

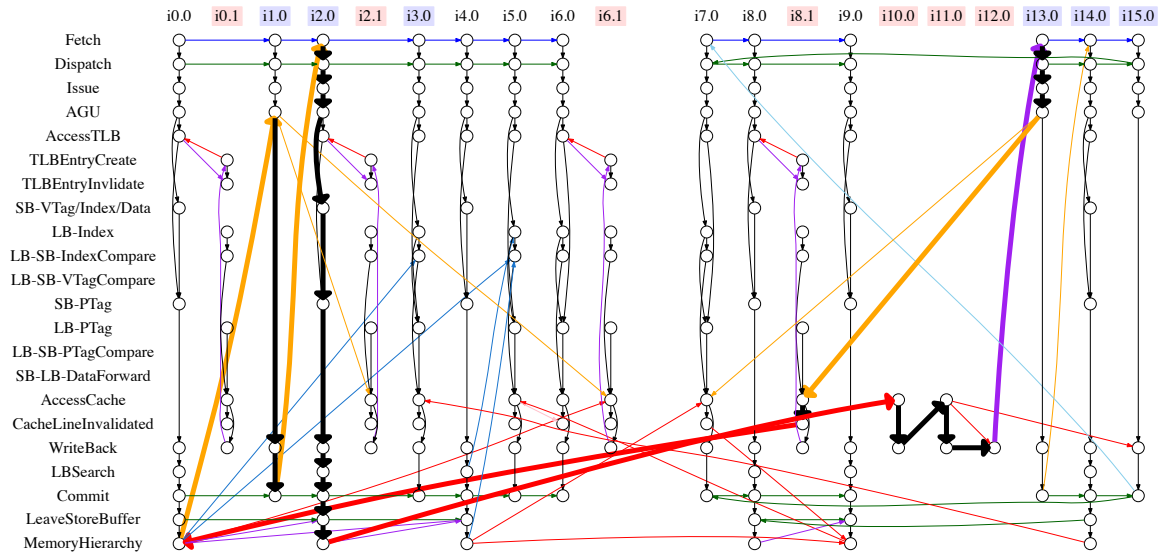


Figure 11: Cyclic graph illustrating one possible ordering for litmus test `ipi8` [38, 39]. Because the graph is cyclic (highlighted with thicker edges), this outcome is not observable in this case. In this case, the cycle was found before the PTEs for `y` were even enumerated. For this test, all executions lead to cyclic graphs, and hence the outcome is not observable, as expected.

the axiom adds two μhb edges representing that 1) the TLB entry must be created before the instruction can access it, and 2) the TLB entry must be accessed before it is invalidated.

Note that each page table walk may be but need not be associated with a particular instruction. A TLB prefetch could trigger a page table walk at any point in an execution, and so there may be "floating" walks. The TLB occupancy model operates in exactly the same way in both cases.

7. Case Study Litmus Tests

In this section, we study three interesting test cases in the context of the system modeled in Section 6.

7.1 Dependence Checks and Store Buffer Forwarding

As described in Section 2, `n5_synonym` tests whether the store buffer forwarding mechanism takes physical addresses into account. If either core's store buffer does not realize that the two accesses from that core are synonyms, either load may be allowed to bypass the store before it, leading to an illegal outcome. Figure 10 shows one of the μhb graphs by which COATCheck verifies that such an (erroneous) outcome is impossible on our SandyBridge model[5].

The particular execution of Figure 10 shows the scenario in which the processor speculates that the store and load access different addresses and hence (under TSO rules) may be reordered. As in previous litmus tests, the red shading denotes hardware microcode operations (PT walks) executed on behalf of the user-level code. The columns of the μhb graph are annotated with instruction labels (user-level or microcode) corresponding to those in the litmus test.

When the load (i3.0) executes, it finds that the store buffer contains no previous entries with the same index (because in this scenario, the store (i2.0) has not yet issued). This leads to a happens-before edge (shown in thick blue in the figure) between the `LB-SB-IndexCompare` stage of (i3.0) and the `SB-VTag/Index/Data` stage of (i2.0). However, when the store (i2.0) does eventually execute, the condition under which it would not squash the speculatively-executed load is if the load buffer has no entries matching the same index. This could occur only if the load had not yet entered the load buffer. This situation would cause there to be a happens-before edge (also shown in thick blue) from the `LBSearch` stage of the (i2.0) store back to the `LB-Index` stage of the load (i3.0), thereby completing a happens-before cycle indicating that the scenario cannot occur in practice.

7.2 Page Remappings and TLB Shootdowns

Figure 11 shows the μhb graph for the Figure 12 scenario involving address remappings and TLB shootdowns. This scenario is derived from previous work [38, 39]; however, we fill in some underspecified page mappings and initial conditions and adapt it to x86-TSO. The original test also considered two outcomes, one of which was observable and the other of which was forbidden. Here, we split this into two tests; we treat the permitted variant as a separate test and a sanity check that the legal outcome remains observable.

At a high level, 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 writes to `x` (i8.0), whose mapping was just modified. Cor-

[5] In reality, the solver would have stopped searching as soon as it detected a cycle, and it may not have filled in any still-incomplete portions of the graph. However, the complete graph is drawn here for clarity

Initially: [x]=0, VA x → PA a (R/W, acc, dirty), (other mappings omitted for space reasons)			
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(z)	(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
(i6.0)	Ld [x/b] ← 1	(i14.0)	St [v/d] ← ack
(i6.1)	Ld PTE(x)→ TLB	(i15.0)	iret
Outcome: Forbidden			

Figure 12: Code for litmus test ipi8

rectness dictates that this write use the new rather than the old mapping; however, this particular test has the load using the *old, stale* translation in an effort to have COATCheck verify that such a situation is unobservable. Thread 1 (i9.0) sends a message back to thread 0 (via 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) (as it used the old mapping). These orderings (plus the remaining low-level details) cause COATCheck to find μhb cycles in all cases.

This graph combines many COATCheck features: IPIs and their handlers, microcode which does not pass through the entire pipeline, orderings enforced (or not) by fences on different types of orderings, and so on. The scale and complexity of these analyses emphasizes the need for tools like COATCheck to automate the enumeration of such graphs and identify the cycle (thicker highlighted lines) to demonstrate non-observable outcomes.

7.3 OS Responsibility for Synonym Tracking

A third case study discusses maintaining coherence among the status bits in a synonym set. Our SandyBridge hardware does *not* guarantee that dirty bit update ghost instructions will also update all synonym pages [20]; the OS is responsible for identifying and updating dirty bits for any synonym PT entries. This can lead to a scenario in which data may be lost if this coordination is not implemented correctly.

Suppose x and y are synonyms mapped to PA a whose PTEs are both marked as clean. When a store is done to x, hardware will mark its PTE as dirty. Suppose the OS intends to swap out the physical page holding PA a and hence needs to check if the page is dirty. If it does so by only checking the PTE for y (and *not* checking the PTE for its synonym x), then a naive OS may incorrectly think the physical page is clean. We have implemented the model for this scenario (μhb graph not shown due to space reasons) and while the analysis accounts for all known happens-before orderings, an acyclic μhb graph can be found, indicating the event may be observable.

Fixing this case requires that at page eviction time, a less-naive OS such as Linux checks whether PTEs for *all* members of a synonym set are marked clean. The ELT would therefore contain an extra load of PTE(x) and a proposed outcome indicating that this load returned a clean PTE. COATCheck would then detect a violation of the SandyBridge Reads axiom, which requires that each load return the value of the latest store to that physical address (since the dirty bit update would be ordered before the load), and conclude that the bad outcome is no longer observable.

This example highlights the fact that *memory transistency models are broader in scope than memory consistency models, including SC-for-VAMC* [38, 39]. Note that in the example, the bug may be observable even when there is no reordering of any kind taking place. In other words, the bug is observable even on a sequentially consistent system. Furthermore, since the two non-ghost instructions access different virtual and physical addresses, and the status bit updates target a different PTE than the non-ghost load uses, the necessary ordering requirement (i.e., to check the state of the synonym pages) is also outside the scope of VAMC. Hence, there may be a translation-related memory ordering bug even on a SC-for-VAMC system. COATCheck is the first MCM analysis framework which can capture and reason about this additional level of indirection, and it can do so regardless of which layer of Figure 2 is used to solve it.

8. Automated Verification Software

8.1 Test Characteristics

We have performance tested COATCheck on a wide-ranging set of 118 litmus tests. We include a number of tests from Intel and AMD manuals and other prior work to sanity check that our SandyBridge model (Section 6) behaves as expected [7, 17, 32, 45].

Another category of litmus tests are modifications of the "standard" tests above to directly test address translation and OS interface issues. We either derived these from previous work [38, 39] or custom wrote them to test new functionality or scenarios. In some cases these originate as user-level codes and follow the full transformation path shown in Figure 3. In a few cases, we wrote the tests directly as ELTs to expediently achieve the desired test scenario. In general, these litmus tests lead to larger graphs, because they employ more of the ghost instructions and system call modules that lead to test size growth en route to an ELT.

Lastly, we also include the case studies of Section 7, plus numerous variants thereof. In all, this gives us a wide-ranging suite of tests on which we perform our analysis.

8.2 Performance Results

Figure 13 shows the full runtimes for our COATCheck implementation. Performance measurements were taken on a server with a 3.2GHz Intel Xeon E5-2667 v3 CPU. The runtimes shown in this graph are for the path from ELT

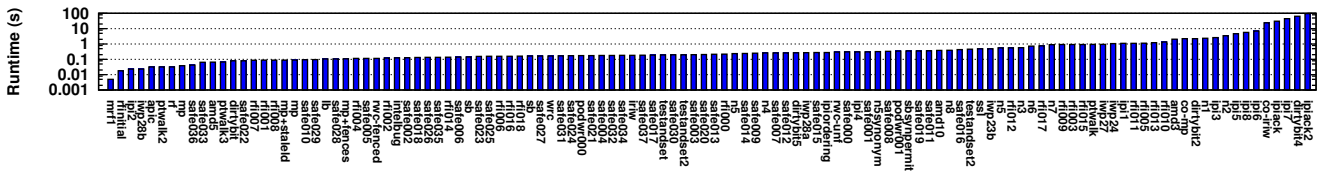


Figure 13: Execution times for full litmus test suite, sorted from smallest to largest run time.

through the constraint solver of Section 5. Recall that a cyclic μhb graph means that the event cannot be observed, and an acyclic μhb graph means that it is possible to observe. Therefore, when performing cycle checks, the cycle-checker can stop its analysis *for a given graph* as soon as a single cycle has been found in it. Likewise, when performing cycle checks for a litmus test intended to be forbidden, one can stop the checking *for the whole litmus test* as soon as a single acyclic graph has been found; one acyclic graph indicates an outcome intended to be forbidden might be observable.

Figure 13 shows the runtimes of the tests, sorted in increasing order. All 118 tests complete in less than 100 seconds, and many are even faster. All tests taking more than 30 seconds were created for this paper to test address translation, IPIs, TLB behavior, and so on. Although these μhb graphs are the largest for which such enumerate-and-check approaches have been used—they often an order of magnitude larger than those from the original PipeCheck work—our runtimes are similar to or less than those of PipeCheck. This shows that even though μspec is more general than previous work on μhb graphs [26, 27], the COATCheck constraint solver algorithm and its implementation are orders of magnitude more efficient. This allows COATCheck to be suitable even for interactive verification, thereby dramatically improving ease of use and debugging in practice.

9. Related Work

Computer architects and verification experts have long studied MCMs and in recent years researchers have made significant progress in formalizing MCM specifications and analysis [3, 13, 23, 28, 30, 32, 40]. These efforts have clearly demonstrated that 1) there is a need for rigor when analyzing memory models, 2) both informal and formal hardware and software models are highly prone to bugs [4, 36], and 3) it can take years to develop models which are sound with respect to hardware, sound with respect to the intention of the designers, and usable by programmers [8–10].

While these contributions have shed light on how to correctly design high-performance shared memory systems, they do not distinguish virtual and physical addresses and do not adequately cover hardware-OS interactions. As such, researchers have only recently focused on the relationship between MCMs and address translation [38, 39]. While our efforts are partly inspired by their observations, their virtual address memory models do not as comprehensively span system layers as ours do. For example, the more abstract VAMC [38, 39] does not reason about the interactions of OS activities like system calls and interrupts, microarchitectural activities like microcode injection, and low-level details like hardware page table walks. Instead, COATCheck formalizes a methodology to understand their interactions, and to verify that their relationship with MCMs actually meets specifications. In fact, these low-level details are essential, not just to design correct systems in the face of increasing system complexity (e.g., with heterogeneous systems and unified address spaces [34, 35]), but also to ensure that various recent TLB proposals on optimizations like prefetching [12, 41], coalescing [33], range TLBs [22], and TLB shootdowns [37, 46] operate correctly.

10. Conclusions

Memory consistency models have long been difficult to define, implement, and analyze. The need to properly handle hardware-OS interface issues such as address translation only adds new complexity. This paper provides methods and a full, efficient toolflow for automatically specifying and verifying memory ordering at the hardware-OS interface. Through many detailed case studies, our work also pushes beyond existing definitions of "consistency" to the more general notion of "transistency", because the ordering requirements in some of our tests cannot be expressed by consistency models alone. The COATCheck toolset facilitates further exploration of hardware-OS memory ordering issues, both in support of system verification itself, and also in the context of forward-looking definitions and explorations of consistency and transistency.

Acknowledgments

We thank Guilherme Cox, Yatin Manerkar, Caroline Trippel, Jan Vesely, and the anonymous reviewers for their helpful feedback. This work was supported in part by C-FAR (under the grant HR0011-13-3-0002), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, and in part by the National Science Foundation (under grants CCF-1117147 and CCF-1253700).

References

[1] S. V. Adve and M. D. Hill. Weak ordering—a new definition. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[2] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design (FMSD)*, 41(2):178–210, 2012.

[3] 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 (TOPLAS)*, 36(2):7:1–7:74, July 2014.

[4] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[5] AMD. Revision guide for AMD Athlon 64 and AMD Opteron processors. Publication number 25759. Revision: 3.79. 2009. URL `http://support.amd.com/TechDocs/25759.pdf`.

[6] AMD. Revision guide for AMD family 10h processors. publication number 41322. revision: 3.92. 2012.

[7] AMD. AMD64 architecture programmer's manual. `http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/`, 2013.

[8] D. Aspinall and J. Sevcik. Java memory model examples: Good, bad and ugly. *1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.

[9] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *38th Symposium on Principles of Programming Languages (POPL)*, 2011.

[10] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *39th Symposium on Principles of Programming Languages (POPL)*, 2012.

[11] Benedict R. Gaster. HSA memory model. *Hot Chips Tutorial*, 2013. URL `http://hsafoundation.com/hot-chips-2013-hsa-foundation-presented-deeper-detail-hsa-hsail/`.

[12] A. Bhattacharjee and M. Martonosi. Inter-core cooperative TLB for chip multiprocessors. In *15th International Symposioum on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[13] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[14] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5 (7), 1962.

[15] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. *International Conference on Parallel Processing (ICPP)*, 1991.

[16] A. Glew, G. Hinton, and H. Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions, Oct. 21 1997. URL `https://www.google.com/patents/US5680565`. US Patent 5,680,565.

[17] Intel. Intel 64 architecture memory ordering white paper. 2007. SKU 318147-001.

[18] Intel. Intel Core Duo processor and Intel Core Solo processor on 65 nm process specification update. Document number 309222. Revision number 20., 2009.

[19] Intel. Intel 64 and IA-32 architectures optimization reference manual, 2013.

[20] Intel. Intel 64 and IA-32 architectures software developer's manual, 2013.

[21] Intel. Intel Xeon processor E5 product family specification update. Reference number 326510-018., 2015.

[22] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. Redundant memory mappings for fast access to large memories. In *42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 28(9), 1979.

[24] N. M. Lê, A. Guatto, A. Cohen, and A. Pop. Correct and efficient bounded FIFO queues. *25th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, 2013.

[25] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *18th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[26] D. Lustig, M. Pellauer, and M. Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *47th International Symposium on Microarchitecture (MICRO)*, 2014.

[27] D. Lustig, M. Pellauer, and M. Martonosi. Verifying correct microarchitectural enforcement of memory consistency models. *IEEE Micro Top Picks of 2014*, 35(3):72–82, May 2015.

[28] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *24th International Conference on Computer Aided Verification (CAV)*, 2012.

[29] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[30] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *32nd Symposium on Principles of Programming Languages (POPL)*, 2005.

[31] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, and O. Giroux. Towards implementation and use of `memory_order_consume`. *ISO SC22 WG21 N4321*, November 2014.

[32] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[33] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *45th International*

Symposium on Microarchitecture (MICRO), 2012.

[34] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on GPUs. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[35] J. Power, M. Hill, and D. Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[36] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[37] B. Romanescu, A. Lebeck, D. Sorin, and A. Bracy. UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all. In *16th International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[38] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[39] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. Address translation aware memory consistency. *IEEE Micro*, 31(1), Jan 2011.

[40] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *32nd Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[41] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *27th International Symposium on Computer Architecture (ISCA)*, 2000.

[42] T. C. Schroeder. Peer-to-peer & unified virtual addressing. *NVIDIA GPU Technology Conference*, 2011.

[43] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7), 2010.

[44] The Coq development team. *The Coq proof assistant reference manual, version 8.0*. LogiCal Project, 2004. URL `http://coq.inria.fr`.

[45] The diy development team. *A don't (diy) tutorial, version 5.01*, 2012. `http://diy.inria.fr/doc/index.html`.

[46] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.