CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface

Yatin A. Manerkar Daniel Lustig Michael Pellauer* Margaret Martonosi
Princeton University *NVIDIA
{manerkar,dlustig,mrm}@princeton.edu mpellauer@nvidia.com

ABSTRACT

In parallel systems, memory consistency models and cache coherence protocols establish the rules specifying which values will be visible to each instruction of parallel programs. Despite their central importance, verifying their correctness has remained a major challenge, due both to informal or incomplete specifications and to difficulties in scaling verification to cover their operations comprehensively. While coherence and consistency are often specified and verified independently at an architectural level, many systems implement performance enhancements that tightly interweave coherence and consistency at a microarchitectural level in ways that make verification of consistency difficult.

This paper introduces CCICheck, a tool and technique supporting static verification of the *coherence-consistency interface* (CCI). CCICheck enumerates and checks families of microarchitectural happens-before (μhb) graphs that describe how a particular coherence protocol combines with a particular processor's pipelines and memory hierarchy to enforce the requirements of a given consistency model. To support tractable CCI verification, CCICheck introduces the ViCL (Value in Cache Lifetime), an abstraction which allows the μhb graphs to cleanly represent CCI events relevant to consistency verification, including demand fetching, cache line invalidation, coherence protocol windows of vulnerability, and partially incoherent cache hierarchies. We implement CCICheck as an automated tool and demonstrate its use on a number of case studies. We also show its tractability across a wide range of litmus tests.

1. INTRODUCTION

Memory consistency models (MCMs) establish the rules by which programmers or compilers can reason about which values will be visible to each instruction of a given parallel program. Likewise, cache coherence protocols establish system support for correct sharing of cached values across the memory hierarchy. Both components need to be verified as part of the design of a parallel processor. Unfortunately, the cost of processor verification continues to grow with each generation, and now often represents over half of a project's total hours [19]. Substantial industry and academic effort has gone into verification of coherence and consistency, but bugs arising from unexpected interactions of coherence, consistency, and address translation continue to appear in hardware even today [4, 8, 26].

Generally, coherence protocols and consistency models are verified independently [6, 7, 13, 34, 44, 45], with coherence verifiers ignoring consistency implications and consistency verifiers making assumptions about coherence. However, coherence and consistency are often tightly interwoven at the implementation level, commonly for the sake of aggressive performance optimizations such as speculative load reordering, but even in simpler microarchitectures as well [9, 20, 25, 40]. The coherence protocol assumes that any remaining orderings for consistency beyond its coherence guarantees will be provided by the rest of the microarchitecture, and the rest of the microarchitecture likewise expects certain guarantees to be provided by the coherence protocol. The guarantees that the coherence protocol provides to the rest of the microarchitecture—and that the rest of the microarchitecture in turn expects from the coherence protocol—together constitute the *coherence-consistency interface* (CCI).

If the coherence protocol does not provide the guarantees that the rest of the microarchitecture expects it to provide, the combination of their orderings may not be strong enough to enforce the architecture's consistency model, leading to consistency violations. For example, coherence protocol verification often checks for properties such as the *single writer/multiple readers* (SWMR) invariant (only one writer or multiple readers per address at any time) or the *data value invariant* (DVI) (the value of an address at the start of a new read epoch is the same as its value at the end of the last read-write epoch) [39]. However, some coherence protocols may also contain features like a livelock prevention mechanism. In some cases (e.g., Section 2.1), such a mechanism may cause stale data to occasionally be returned to a core, but since it does not break coherence invariants like SWMR or DVI, it will not be flagged as a bug by a coherence protocol verifier. Meanwhile, if the pipeline is verified (and operates) under the assumption that the coherence protocol will never allow stale data

MICRO-48, December 05-09, 2015, Waikiki, HI, USA
Copyright 2015 ACM. ISBN 978-1-4503-4034-2/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830782

to be returned to it, it may use stale data in cases where it should not, resulting in consistency violations. This is an example of the verification gap that arises when coherence and consistency are verified independently: it may lead to situations in which the coherence protocol and the rest of the microarchitecture each expect the other to enforce required orderings, resulting in bugs.

As the above example shows, it is generally impossible to verify the correctness of a memory model implementation unless ordering enforcement at the CCI is explicitly taken into account. We therefore present CCICheck, a general-purpose, open-source framework and tool for verifying that the orderings required by an architecture's consistency model are maintained by the combination of the microarchitecture-level orderings enforced by the pipeline, coherence protocol, and memory hierarchy. CCICheck follows PipeCheck in enumerating all possible microarchitectural happens-before (μhb) graphs for a suite of litmus tests[1] on a given implementation [30, 31]. CCICheck borrows its pipeline model from PipeCheck, but it provides constructs for more detailed memory system modeling. CCICheck replaces the *reads-from* (`rf`), *write serialization* (`ws`) and *from-reads* (`fr`) edges used by PipeCheck with CCI-aware edges that represent the *microarchitectural* enforcement of relationships like SWMR and DVI by a particular coherence protocol.

Another key contribution of CCICheck is its notion of a ViCL, or "value in cache lifetime", which summarizes the residency of data values in locations in the memory hierarchy. ViCLs allow CCICheck to tractably and yet comprehensively model scenarios involving demand fetching, partial incoherence, and a variety of coherence protocol transitions. This allows it to detect bugs that would have gone unnoticed by prior verifiers.

CCICheck's main high-level contribution is that it achieves scalable CCI-aware verification by clearly enumerating how implementation-level guarantees provided by the pipeline, coherence protocol, and memory hierarchy combine to enforce all of the requirements of a consistency model. CCICheck allows users to explicitly specify CCI-aware orderings for their implementation, and it automatically and intelligently enumerates all possible executions of provided litmus tests on the given implementation. Through intelligent pruning that reduces the number of graphs enumerated, CCICheck can automatically verify large systems with non-trivial memory hierarchies in tractable runtimes.

2. MOTIVATION

2.1 Window of Vulnerability/Peekaboo

As one example of non-intuitive CCI behavior, consider the *window of vulnerability* problem, a situation in which certain coherence protocols are prone to livelock due to repeated invalidation-before-use of data propagating through a cache hierarchy [28]. In one proposed livelock-avoidance technique, a core is allowed to perform one operation on the data when it arrives, even if it has already been invalidated [39]. This does not violate coherence as the operation is effectively ordered at the time of the invalidation. However, this use of stale data may lead to consistency violations, and is known as the "Peekaboo" problem.

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under TSO: Forbid r1=1, r2=0	

(a) Code for litmus test `mp`

	Core 0	Core 1
1		x: prefetchS miss, issue GetS/IS^D
2	x: receive Fwd-GetS, send Data[0]/S	
3	x: store miss; issue GetM/SM^{AD}	
4	x: receive Data[0](ack=1)/SM^A	x: receive Inv, send Inv-Ack/IS^{D}I
5	x: receive Inv-Ack, perform store/M[1]	
6	y: store hit/M[1]	
7		y: load miss, issue GetS/IS^D
8	y: receive Fwd-GetS, send Data[1]/S[1]	
9		y: receive Data[1], perform load r1=1/S[1]
10		x: load miss, stall/IS^{D}I
11a		x: receive and drop Data[0], replay GetS/IS^D
12a	x: receive Fwd-GetS, send Data[1]/S	
13a		x: receive Data[1], perform load r2=1/S[1]

(b) The Baseline WoV solution drops stale data upon receipt: livelock-prone, but no consistency violation

	Core 0	Core 1
11b		x: receive Data[0], perform load **r2=0**/I

(c) If livelock avoidance is naively added for WoV cases, steps 11a, 12a, and 13a are replaced by step 11b. Stale data can be read, resulting in a consistency violation.

Figure 1: Two executions of `mp` in scenarios prone to the window of vulnerability problem (adapted from [39])

Figure 1 demonstrates two operational execution sequences of the `mp` litmus test (Figure 1a) which demonstrate potential livelock and a consistency violation respectively. Under TSO, the `mp` test outcome `r1=1, r2=0` is forbidden, as neither the stores nor the loads may be reordered. The executions of Figure 1 begin identically, as denoted by Steps 1-10 which are common to both: a prefetch or speculative request for `x` is issued (step 1) before the load of `y` executes. The prefetched line is invalidated (step 4) by core 0's store to `x` before the data is received. The demand request for `x` from the core is also issued (step 10) before the (now stale) data for `x` arrives.

At step 11, the two sequences diverge in behavior. In the first execution, the stale data is dropped and

[1] A litmus test is a small program used to test consistency model implementations.

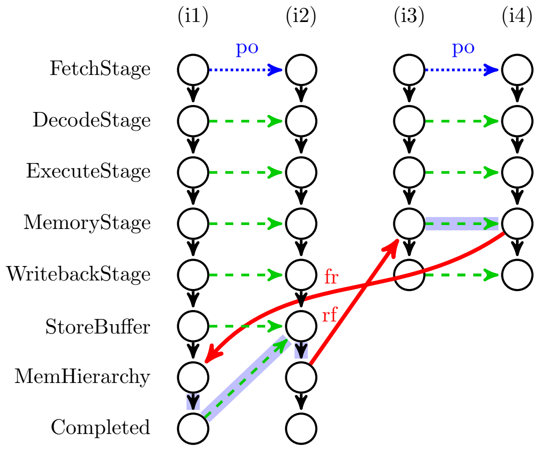


Figure 2: PipeCheck μhb graph for the mp litmus test.

the load is retried; this satisfies TSO but may lead to livelock. In the second execution, when the stale value of x arrives at core 1, the coherence protocol returns that value of x to the core (Step 11b) to prevent livelock. When it does so, it creates a consistency violation by allowing the forbidden outcome r1=1, r2=0 to occur.

One solution which avoids livelock while also satisfying TSO ordering requirements is to allow access to invalidated data if and only if the accessing instruction was the oldest unperformed load or store in program order at the time the coherence request for the now invalidated data was issued [39] (henceforth referred to as the "Peekaboo solution"). This is clearly a case where a feature of the coherence protocol (the livelock-avoidance mechanism) affects the MCM implementation in a way that goes beyond traditional coherence protocol properties such as SWMR. The interplay between the pipeline, coherence protocol, and memory hierarchy gives rise to complexity beyond what is verifiable through non-rigorous, intuitive methods alone. CCICheck's modeling and verification of Peekaboo are covered in detail in Section 6.3.

2.2 Inadequacies of Current Verifiers

Current MCM verifiers cannot address CCI verification problems such as the Peekaboo scenario. For example, the recently-proposed PipeCheck tool (which CCICheck builds on and generalizes) performs automatic MCM verification at the pipeline level [30] using enumerations of microarchitectural happens-before (μhb) graphs. Figure 2 shows an example of the μhb graph for two five-stage in-order TSO cores for the litmus test mp. A column of nodes in a μhb graph (e.g., column (i1)) represents an instruction executing on a core. Each node in a column represents an instruction at a specific pipeline stage. Each edge in the graph represents a happens-before ordering enforced at the microarchitectural level via passing of messages, enforcement of in-order execution, or some other mechanism. Acyclic and cyclic μhb graphs correspond to observable and unobservable executions respectively. If a cycle occurs, it would mean that an event happens before itself, which is contradictory; therefore such a scenario must be impossible to observe.

In this graph, the black vertical edges represent the progression of an instruction through the various pipeline stages. Blue po edges represent instructions passing through the fetch stage in program order (po). The horizontal green edges enforce that the pipeline stages are in order.

The red rf ("reads-from") edge represents the transfer of the value y = 1 between core 0 (i2) and core 1 (i3) for this particular execution of the test. Notably, it abstracts away the various coherence protocol transitions and individual caches that comprise the transfer. Likewise, the red fr ("from-reads") edge enforces that for the read of x to return a value of 0 in this execution, it must occur before the store of x = 1 reaches the memory hierarchy.

As can be seen in Figure 2, PipeCheck abstracts the entire memory hierarchy into a single row of nodes. This makes it impossible to use it for the analysis of CCI scenarios such as the Peekaboo problem, partial incoherence, lazy coherence, etc., as there is no way to refer to individual caches, let alone cache sub-events. For example, consider using Figure 2 to model the Peekaboo solution described in Section 2.1. The red from-reads (fr) edge cannot be kept in the graph, as core 1 is made aware of the store of 1 to x through an invalidation *before* the load occurs. The edge cannot simply be removed either, as doing so would make the graph acyclic and imply that the execution was observable, even though the Peekaboo solution guarantees that such an execution is forbidden.

CCI verification requires the ability to reason about cache occupancy and coherence protocol events for a specific memory hierarchy and coherence protocol. For example, in the case of the Peekaboo problem, the analysis must keep track of when the data x = 0 is present in Core 1's cache relative to other events in the execution. It also needs to keep track of whether the invalidation of x's cache line happens before or after the data is used by Core 1. CCICheck reasons about cache occupancy and coherence events using the ViCL abstraction, which is explained in the next section.

3. THE ViCL ABSTRACTION

This section introduces the Value in Cache Lifetime (ViCL) abstraction. ViCLs model cache occupancy and coherence protocol events relevant to consistency verification in a way that maintains the ability to perform tractable and flexible analysis.

3.1 ViCLs: Definition and Usage

Conceptually, a Value in Cache Lifetime (ViCL) represents the period of time (relative to a single cache) over which a given value[2] is present in a specific cache or memory. To formally define a ViCL, we first con-

[2] We assume without loss of generality that each store produces a unique value even if its contents are identical with the value of another store. This does not affect correctness, as our enumeration algorithms would consider either store to be a possible source for a load returning the value in question.

ceptually assign a unique *cache_id* to every cache in the system, and a unique *generation_id* to each line brought into a given cache over the duration of an execution. This allows us to uniquely refer to each cache line in an execution using its *cache_id* and *generation_id*. Formally, a ViCL is a 4-tuple

$$(cache\_id, address, data\_value, generation\_id)$$

which maps onto the period of time within which the cache line corresponding to *generation_id* in the cache identified by *cache_id* holds the value *data_value* for address *address*. A given address and data value pair may have many matching ViCLs over the course of an execution. These ViCLs could be in different caches (different *cache_id*s), be brought into the same cache at different points in the execution (different *generation_id*s), or both. Likewise, different data values for an address correspond to different ViCLs, which is key to their use in enumerating possible read-write orderings for consistency verification. There may also be gaps between ViCLs for a given address and cache pair where there is no value in the cache for that address. In addition, our definition allows for the (admittedly uncommon, but feasible) possibility that a cache may hold two lines for the same address simultaneously.

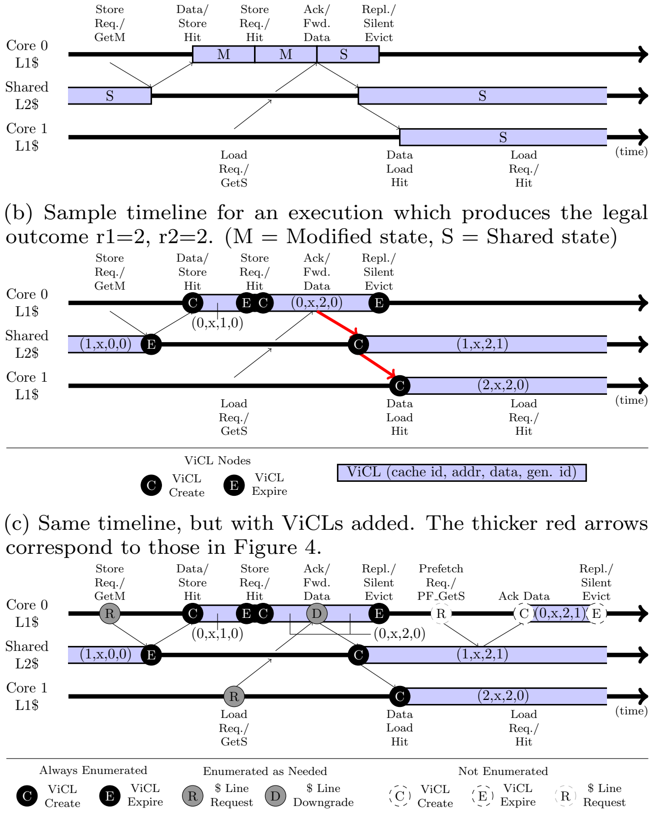
A ViCL mapping's time period starts at a *ViCL Create* event and ends at a *ViCL Expire* event. ViCL Create and ViCL Expire events represent the points in time at which the corresponding ViCL 4-tuple starts and stops serving the data in question respectively. A ViCL Create event occurs for address `x` when either (i) a cache line containing `x` enters a usable state from a previously unusable state, or (ii) when a value is written to `x` in a cache line. A ViCL Expire event for address `x` occurs when (i) its cache line enters an unusable state from a previously usable state, or (ii) a value is written to `x` in a cache line.

ViCLs are per-address, while cache lines generally hold more than one address. We assume without loss of generality that the creation and expiration of a ViCL for a given address has no inherent effect on ViCLs of other addresses, even if they do share a cache line. Any such sharing possibilities (e.g., evictions due to false sharing) are already handled by CCICheck's comprehensive enumeration. In such cases, our analysis is conservative but correct. For clarity, as in previous consistency model studies [7, 30, 32, 36, 38], we assume that values are identically-sized (at the granularity at which cores address memory) and that all addresses are aligned to this same granularity (i.e., there is no partial address aliasing).

ViCLs can also represent main memory, because we can also assign a unique *cache_id* to capture all of main memory. Because memory lines are not evicted in the same way that cache lines are, the *generation_id* for all main memory ViCLs will never change[3]. ViCLs representing main memory can be useful to represent uncacheable accesses.

Thread 0	Thread 1
(i1) [x] ← 1	(i3) r1 ← [x]
(i2) [x] ← 2	(i4) r2 ← [x]
In TSO: r1=2, r2=2 Allowed	

(a) Litmus test `co-mp`



(b) Sample timeline for an execution which produces the legal outcome r1=2, r2=2. (M = Modified state, S = Shared state)

(c) Same timeline, but with ViCLs added. The thicker red arrows correspond to those in Figure 4.

(d) Same timeline, but with additional cache line events that may be needed to model certain scenarios

Figure 3: An example of how ViCL nodes relate to events in the cache hierarchy.

3.2 ViCL Timeline Example

Figure 3 shows an example of how litmus test `co-mp` (Figure 3a) might be executed on a fully coherent system with private L1 caches and a shared L2 cache. Figure 3b presents a timeline using traditional notions of cache line state, while Figure 3c presents the same timeline using ViCLs. In addition to ViCL create and expire points, it also denotes the value of ViCL 4-tuples at noteworthy points in the timeline. At the beginning, the shared L2 cache holds the value 0 for address `x`. The first thread 0 store then misses in its L1 cache, causing it to fetch the line for exclusive ownership. When the L2 cache receives the request, the L2 cache's ViCL expires[4]. Once the data arrives at core 0's L1 cache and the store completes, a new ViCL is created at the L1 cache, representing both the move to a valid state and the writing of new data. When the second store then completes, the first L1 ViCL

[3] ...unless data is swapped from memory out to disk, which we do not consider (though the formalism supports it).

[4] The line may not necessarily be invalidated; it may move to a state tracking the L1 cache as the new owner. Nevertheless, the ViCL expires because the old data is no longer being served.

(which had a data value of 1) expires, and a new ViCL is created for `x` with the value 2.

When thread 1 starts executing sometime later, it will fetch the data from core 0's L1 cache. This does not cause a ViCL expiration, as no components of the core 0 L1 ViCL's 4-tuple change. Instead, the L1 cache forwards the data to the other L1 through the L2, creating new ViCLs in those caches in the process.

In some scenarios, ViCL create and expire events may be sufficient to verify all necessary orderings. In more complex scenarios, however, we can add additional coherence-related events to the timeline, and infer orderings for those events with respect to ViCL create and/or expire events. For example, a cache line downgrade event (for a cache line in an exclusive state) may take place between ViCL Create and ViCL Expire. Similarly, a cache line request event will happen before the ViCL it fetches gets created. Figure 3d shows how these additional events can be included in the timeline for the same execution of `co-mp`. Our Peekaboo and TSO-CC case studies in Section 6 require such additional nodes.

3.3 Using ViCLs in μhb Graphs

A key benefit of ViCL events is that they map naturally into nodes within CCICheck's μhb graphs. Likewise, orderings between such events (e.g., due to message passing) map naturally onto μhb graph edges.

When modeling a particular execution by a μhb graph, we add μhb nodes to the graph for every cache-accessing instruction representing the ViCL create and ViCL expire events for the ViCL(s) that instruction accesses in the execution. We rely on the microarchitecture definition (Section 4.1) to list all possible caches that a memory-accessing instruction could interact with (e.g., read directly from L1, read from L2 through the L1, etc.). The address and data of a ViCL's 4-tuple must match those of its instruction, and the possibilities for a ViCL's generation ID are used to check whether two ViCLs of the same address, value, and location are the same ViCL or not. The maps from instructions to ViCLs for an execution are not injective (since there may be multiple instructions which map to a single ViCL); each ViCL's create and expire nodes appear in the μhb graph no more than once. The maps are also not surjective because there may be ViCLs which are not accessed by any instruction, such as the first ViCL in Figure 3c. We do not need to draw such non-accessed ViCLs as they are not relevant to consistency enforcement.

Figure 4 illustrates a CCICheck μhb graph with ViCL nodes for the execution timeline depicted in Figure 3c. The four ViCLs in the graph represent the four rightmost ViCLs in Figure 3c (the very first ViCL does not need to be enumerated as its data is not read by any instruction). The edges between the ViCLs reflect the ordering constraints in the microarchitecture specification (Section 4.1). For example, the dot-dashed brown "SW" edge enforces that ViCLs for the first write to `x` (i1) must expire before the ViCLs for the subsequent write (i2) are created, as per the coherence protocol's SWMR invariant. The "NoDups" edge (which in this case overlaps with the "SW" edge) between the two ViCLs for `x` in core 0's L1 cache enforces that the first ViCL for `x` in the cache must expire before a second one for the same address in the same cache can be created (i.e., there can be no duplicates within a single cache at any time). The two red "SourcedFrom" edges correspond to the arrows in Figure 3c showing `x=2` being propagated from the core 0 L1 to the core 1 L1 through the L2. Finally, the four black "UsesViCL" edges represent the fact that i3 and i4 access the same ViCL in this execution.

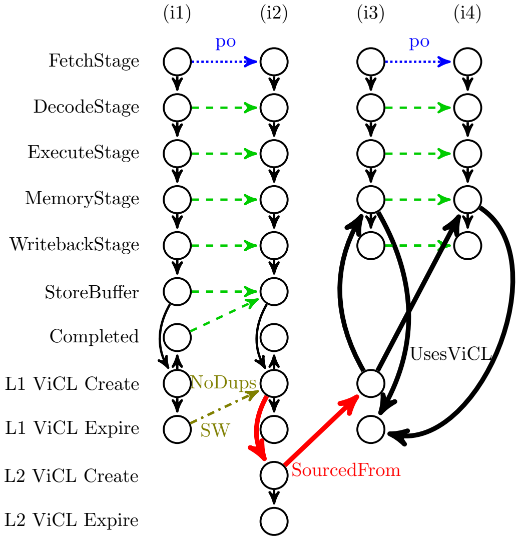


Figure 4: CCICheck graph for the Figure 3c scenario.

For each cache-accessing load, μhb edges represent the fact that the instruction must access its ViCL sometime between its creation and expiration. Likewise, for each cache-accessing store, μhb edges represent the fact that ViCLs are created when store values reach caches (because they change the data component of the ViCL 4-tuple). μhb edges are also used to represent any orderings enforced or implied between ViCLs due to coherence protocol behavior (such as the "SW" edge in Figure 4) or memory hierarchy restrictions (like the "NoDups" edge in Figure 4). The details of these orderings are specific to each protocol (or class of protocols). If other cache line events (like fetch requests and downgrades) are modeled, then the graph will include μhb nodes representing these events and edges representing their orderings with respect to other events in the graph. Examples of such cases can be found in Section 6.

The graph in Figure 4 is acyclic, so the outcome `r1=2,r2=2` is observable on the target machine. This graph is one of many that CCICheck enumerates to comprehensively cover all of the different ordering possibilities for a piece of code under test. Section 4 discusses the details of microarchitecture definitions and the enumeration process.

4. CCICHECK OPERATION

4.1 Microarchitecture Definitions

A CCICheck microarchitecture definition consists of three primary components. First, the definition specifies the set of paths each instruction type can take through the pipeline and memory hierarchy. (For example, a load may read from the L1 or from the L2 through the L1, etc.) Second, it specifies orderings maintained by the pipeline between pipeline stages (e.g., declaring that a decode stage is FIFO). These two steps match the approach taken by PipeCheck [30, 31].

Third, and most importantly, CCICheck does *not* reuse the reads-from (`rf`), write serialization (`ws`), and from-reads (`fr`) edges used by PipeCheck and other previous work [3, 7]. These edges represent the behavior of a highly-abstracted memory hierarchy and are insufficient for CCI-aware verification. Instead, a CCICheck microarchitecture definition requires users to explicitly specify *constraints*: self-contained axioms that describe how instructions interact with ViCLs and how ViCLs interact with each other. In this way, just as PipeCheck replaced abstract architectural preserved program order (`ppo`) edges with an individual pipeline's enforcement of that preservation, CCICheck replaces `rf`, `ws`, and `fr` edges with lower-level edges representing a specific implementation's enforcement of the coherence axioms like SWMR and DVI [39] that underly these higher-level orderings.

The constraint-based approach serves two purposes. Firstly, it allows us to build off (rather than compete with) the significant research effort that has gone into verifying coherence protocols. Coherence axioms such as SWMR and DVI can be verified by coherence verifiers and then used as axioms in CCICheck's constraints. Secondly, the constraint-based approach (alongside our ViCL abstraction) meets our requirement of capturing sufficiently many ordering enforcement details while remaining decoupled from any irrelevant coherence protocol implementation details. In contrast to what might be expected, we find that few coherence protocol axioms are universal at the implementation level. While properties such as SWMR may hold true across many architectures in an abstract sense, the implementations may vary widely (e.g., eager vs lazy invalidation). We therefore necessarily leave the precise specification of each axiom to each individual microarchitecture model. Section 4.3 gives an example.

Some constraints take the form of logical implications. For example, a constraint may look like "for all ViCL Create events v taking place after μhb node n, add edge e from n to v". CCICheck therefore iterates over the entire set of constraints until the graph converges (i.e. until no new edges need to be added to satisfy the constraints). This is nearly always within a few iterations.

Most constraints specify orderings enforced on ViCLs by a coherence protocol. Every ViCL must have a source: some may be sourced directly from a store instruction, some may be sourced from other ViCLs via coherence protocol events, and still others may be sourced from the initial state of memory. Each such case is modeled by μhb edges such as the "SourcedFrom" edges in Figure 4. Likewise, edges representing invalidations of sharers on a write, explicit cache/line flushes, and other orderings may also be added to the constraints for a given model. Collectively, the set of constraints should be strong enough to capture all orderings needed to verify the correctness of an implementation.

4.2 Enumerating Graphs

The CCICheck μhb graph enumeration algorithm involves two high-level steps. First, since the microarchitecture definition specifies that each instruction takes one out of some well-defined number of paths, CCICheck calculates the cross product of the path choices for each instruction in the given litmus test. This enumerates all the ways in which the instructions could flow through the microarchitecture. Second, CCICheck enumerates all of the ways in which each of the constraints in a given scenario can be satisfied.

Each constraint will have zero or more solutions. Each solution adds zero or more edges to the μhb graph to represent orderings that hold true in that microarchitecture. If there are no solutions, the scenario is marked as impossible and discarded. For example, a "StoreBuffer-Forwarding" constraint may be impossible to satisfy if there are no previous stores to the same address from the core in question. Meanwhile, if there are multiple legal solutions to a given constraint, then the solutions are considered as separate possible graphs and further constraints are enumerated independently for each graph.

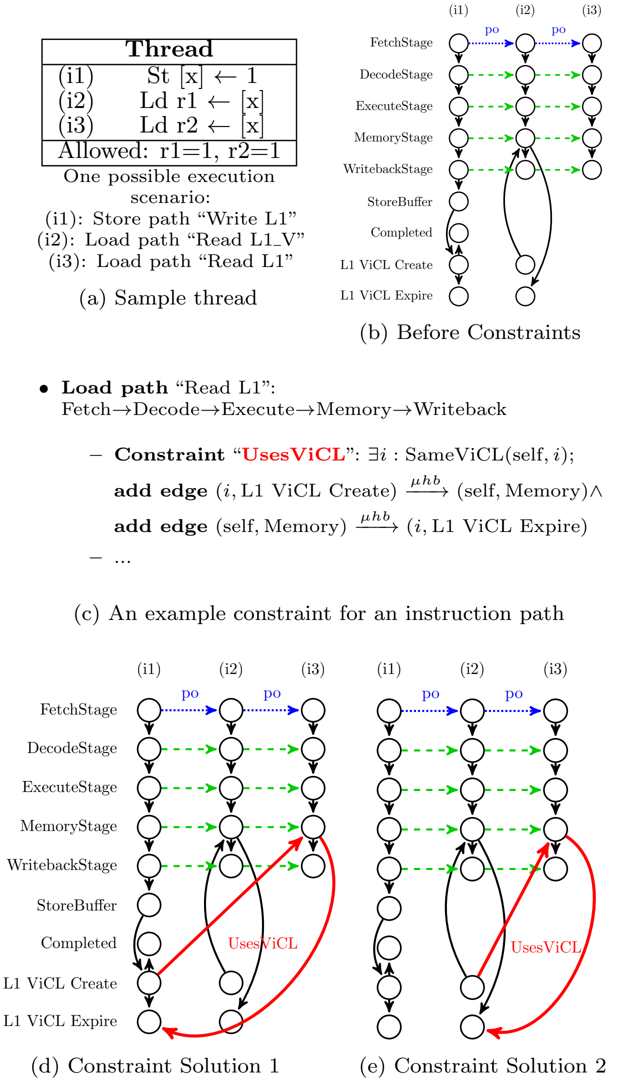
Once a graph has converged, if it is acyclic, then the graph represents an observable outcome on the target machine. On the other hand, if the graph becomes cyclic at *any* point, then the execution is impossible and no further enumeration is needed for that graph (because edges are never removed). If no acyclic graph can be generated for a test, then the test outcome is not observable on the target machine.

We do *not* enumerate all possible concrete generation IDs of ViCLs during the enumeration. For any two instructions which access the same cache id, address, and data in a given execution, we simply enumerate whether the generation IDs for the two ViCLs are equal (in which case the ViCL is shared by both instructions) or distinct (each instruction uses a distinct ViCL).

4.3 Enumeration Process Example

A partial microarchitecture definition is given in Figure 5. Figure 5a depicts a code snippet and one particular choice of path for each instruction. Although this depicts only a single thread for space reasons, the approach is the same for the multithreaded case. Figure 5b shows a baseline μhb graph for this case before constraint-derived edges have been added.

Figure 5c depicts a subset of the constraints for the "Read L1" path taken by the load (i3). In particular, the constraint shown demonstrates the scenario in which (i3) expects that it accesses a ViCL that is also accessed by another instruction in the program. As specified in the logic of the constraint in Figure 5c, the constraint can be satisfied by nodes from any instruction that shares the same ViCL as the load in question. If such an instruction is found, edges are drawn according to the



Thread	
(i1)	St [x] ← 1
(i2)	Ld r1 ← [x]
(i3)	Ld r2 ← [x]
Allowed: r1=1, r2=1	

One possible execution scenario:
(i1): Store path "Write L1"
(i2): Load path "Read L1_V"
(i3): Load path "Read L1"

(a) Sample thread

(b) Before Constraints

- **Load path** "Read L1":
  Fetch→Decode→Execute→Memory→Writeback
  - **Constraint** "**UsesViCL**": $\exists i$: SameViCL(self, i);
    add edge $(i, \text{L1 ViCL Create}) \xrightarrow{\mu hb} (\text{self}, \text{Memory}) \wedge$
    add edge $(\text{self}, \text{Memory}) \xrightarrow{\mu hb} (i, \text{L1 ViCL Expire})$
  - ...

(c) An example constraint for an instruction path

(d) Constraint Solution 1

(e) Constraint Solution 2

Figure 5: Using CCICheck path constraint definitions to build sets of μhb graphs.

"add edges" portion of the constraint: from the ViCL Create node of the satisfying instruction to the Memory stage of the load, and from the Memory stage of the load to the ViCL Expire node of the satisfying instruction. In this case, there are two possible solutions to the "UsesViCL" constraint of (i3). Figures 5d and 5e show the cases where the constraint is satisfied by (i1) and (i2) respectively.

4.4 Pruning and Scalability

While the CCICheck enumeration process may justifiably raise concerns about scalability, in practice most litmus tests do not need to go through all the enumeration steps. Large portions of the solution space are generally pruned (due to early detection of cycles) or ruled invalid (due to unsatisfiability of constraints). We also perform the analysis in a depth-first manner to enable faster pruning (e.g., by stopping as soon as any acyclic graph is found). Through such pruning, we are able to make the analysis tractable and relatively efficient. We could easily further improve the performance of our tool through optimizations such as parallel analysis of independent graphs. However, for this work, we focus our effort on generality and verifiability rather than performance. Our results section (Section 7) demonstrates that for realistic CCI verification scenarios, the runtimes remain quite tractable.

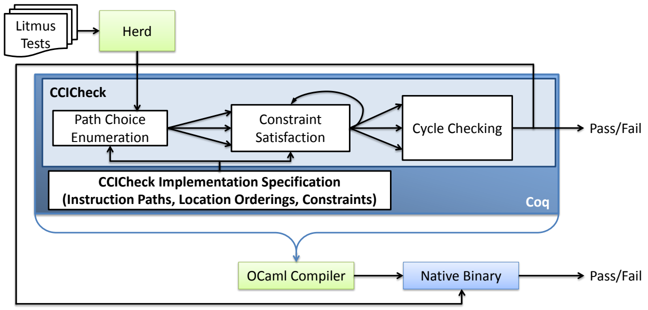


Figure 6: CCICheck structure and toolflow.

5. TOOLFLOW AND METHODOLOGY

CCICheck is intended to be used at design time for static consistency verification. The CCICheck toolflow is shown in Figure 6. Given an implementation specification and a litmus test in `litmus` format [6, 7, 42], CCICheck uses `herd` [7, 42] to parse the test and also to check whether the execution is permitted or forbidden by the architecture-level memory model. CCICheck automatically enumerates the set of μhb graphs representing all possible executions of the litmus test on the implementation according to the steps described in Section 4. After generating the graphs, it compares them to the behavior expected by the architecture-level model (e.g., forbidden outcomes must not generate acyclic μhb graphs). As its output, CCICheck informs the user whether the memory model maintained by the implementation being tested is stronger than, equivalent to, or weaker than the architectural model it is supposed to implement, across the given suite of litmus tests.

CCICheck is written in the Coq proof assistant [41], which allows CCICheck's models and analysis framework to be analyzed formally. We use Coq's built-in extraction functionality to extract our Coq code to OCaml so it can be compiled and run as a standalone binary.

Table 1 describes the various microarchitectures that we analyze in this paper. To emphasize how unexpected CCI behaviors can appear even in simple microarchitectures, we model most architectures as having a five-stage in-order pipeline model. Nevertheless, CCICheck easily adapts to more complex and/or less restrictive pipelines. As one example, Table 1's **TSOCC (OoO)** model uses an out-of-order processor, reflecting the implementation of the original authors [18].

Our models cover a variety of memory hierarchies and coherence protocols, including single and dual layers of private and/or shared caches. "Cache-Cache Trans-

Name	Cache Hierarchy	Cache-Cache Transfers	Protocol Classification
PrivL1	Private L1s	At L1	Eager
SharedL1	Shared L1	—	Eager
Peekaboo	Private L1s	At L1	Eager, with Livelock Prevention
PrivL1L2	Private L1s and L2s	At L2	Eager
PrivL1L2CC	Private L1s and L2s	At L1 and L2	Eager
PL1/SL2	Private L1s, shared L2	At L1	Eager
TSOCC (in-order)	Private L1s, shared L2	At L1	Lazy
TSOCC (OoO)	Private L1s, shared L2	At L1	Lazy

Table 1: Memory hierarchy specifications and coherence protocol features of the analyzed microarchitectures.

fers” describes whether cache lines (and thus ViCLs) in private caches may be sourced from other caches at the same level or whether such propagation must take place via lower levels. The “Eager” coherence protocol classification refers to an abstract vanilla coherence protocol that eagerly invalidates sharers before every write and which is adapted to each different cache hierarchy arrangement. Other classifications listed in Table 1 are described in the corresponding parts of Section 6.

We test our models using a suite of 85 x86-TSO litmus tests. These tests combine hand-written tests from existing x86-TSO suites [36], tests that were automatically generated using the `diy` tool [42], and custom tests addressing ViCL-inspired scenarios not already captured by the suite. Our results are detailed in Section 7.

6. CASE STUDIES

6.1 Partial Incoherence

Our first example studies partially incoherent cache hierarchies such as those found in many GPU systems today. Alglave et al. recently found that the caches in both NVIDIA and AMD GPUs were prone to numerous undocumented or under-specified coherence/consistency model behaviors [4]. Without accounting for cache occupancy and relevant coherence protocol attributes, previous techniques were incapable of modeling such scenarios. Using ViCLs, CCICheck provides a natural framework in which the consistency implications of such scenarios can be analyzed.

For example, Alglave et al. used a modified version of the `mp` litmus test to experiment with GPU cache hierarchies. This version of `mp` has `membar` fences inserted between the two stores and between the two loads in an attempt to impose enough orderings to avoid the forbidden outcome `Ld [y] = 1, Ld [x] = 0` on partially incoherent GPU architectures. Alglave et al. discovered empirically that `membar` fences are *not* sufficient to enforce the test’s necessary orderings on many GPU architectures, but there is no verification framework to

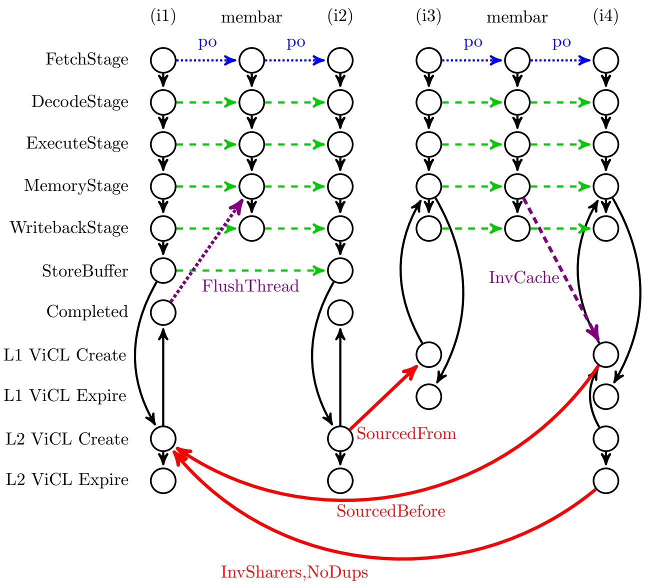


Figure 7: μhb graph for modified `mp` on a hypothetical GPU with no-allocate-on-write and no SWMR guarantee [4]. In particular, if the core 1 fence does not enforce the load→load ordering, the graph will be acyclic.

elucidate this further. As the studied microarchitectures are proprietary, we apply CCICheck to a hypothetical yet realistic GPU model with small, in-order cores, no-allocate-on-write, and in which private L1 caches may be incoherent with the shared L2 cache.

Figure 7 depicts a CCICheck graph for the modified `mp` litmus test on this architecture. (Instruction labels at the top of the graph correspond to the `mp` code in Figure 1a.) Both stores write directly to the L2 cache, while both loads read from the L1. Since the L1s are not coherent with the L2, the ViCL of the load of `x` does not need to be *invalidated* before the ViCL for the store of `x` is created. Instead, it merely needs to be *created* (i.e. the L1 needs to read the value from the L2) before the store of `x` reaches the L2 and creates the store’s ViCL.

CCICheck’s μhb graphs have the right level of abstraction to reason about counterintuitive litmus test outcomes such as the behavior of the modified `mp` litmus test. Consider the dotted “InvCache” edge in Figure 7 between the `membar` fence and the load of `x`. If the fence does not enforce this ordering, then there is no cycle in this graph. Recalling that acyclic μhb graphs indicate a scenario is observable, this graph allows CCICheck to demonstrate how the outcome may be observable on partially incoherent/weakly-consistent GPU architectures. The specific details of a μhb graph can guide designers towards understanding which features are helpful in alleviating non-intuitive or erroneous CCI behavior.

6.2 Lazy Coherence

Our second case study uses CCICheck to analyze CCI behavior in architectures using the recently proposed TSO-CC protocol [17, 18]. TSO-CC is a scalable lazy coherence protocol for TSO architectures that uses private L1s and a shared NUCA L2 that doubles as a directory cache. TSO-CC does not have a complete sharer list in the general case and allows for lazy invalidation of sharers. This results in low on-chip storage requirements for coherence, but requires each core to "catch up" to a future point in logical time whenever it has a L1 miss by invalidating all shared lines in its private L1 cache (and hence any possibly out-of-date values) at that time. This ensures that when a core sees one value from a core other than itself, it is made aware of all previous values from all other cores as well.

Architectures like TSO-CC have an unconventional CCI which can result in the violation of some required consistency orderings. Scenarios like this—in which a coherence protocol's design is coupled with features of an MCM's implementation—are great examples of the need for CCICheck's CCI-aware verification. We model the baseline TSO-CC protocol in CCICheck by adding appropriate constraints to load, store, and fence instruction paths. The load instruction paths have the following edges to reflect cache invalidations on L1 misses: 1) to the cache-missing instruction's L1 ViCL Create node from the L1 ViCL Expire node of each previous shared L1 ViCL on the same core, and 2) from the cache-missing instruction's L1 ViCL Create node to the L1 ViCL Create node of every subsequent L1 ViCL on the same core. We also modify store paths to account for the possibility of a cache line downgrade (Modified→Shared) with paths containing ViCL Downgrade nodes, as Modified lines would not be invalidated on a cache miss while downgraded lines would be. If a store's ViCL in its L1 cache is downgraded before an L1 miss in that cache, it is considered to be a shared ViCL for the invalidation edges of the miss. If it is downgraded after the miss (or not downgraded at all), it is immune to the invalidation for that miss and is not subject to the corresponding invalidation edges. Since these edges deal with single caches as opposed to distributed sets of caches, it is safe to assume a total ordering on events in that single cache for any execution. In cases where the temporal ordering of ViCLs in an L1 with respect to each other is not known, we enumerate all possible permutations to cover all possible cases. Finally, we add a constraint to the fence instruction path to enforce TSO-CC's requirement that all shared lines in a core's L1 are invalidated upon a fence.

Figure 8 shows one possible μhb graph for the mp litmus test (Figure 1a) executing on a microarchitecture implementing TSO-CC. Since shared lines in the L1 caches are invalidated lazily, the L1 ViCL for the load of x need only be sourced before the store of x occurs (as in Section 6.1) as opposed to being invalidated before it. In TSO-CC, if the load of y returns the value 1, it must have taken a cache miss at some point, allowing the new value of 1 to be fetched from core 0. Core 1 would have invalidated shared lines in its L1 cache upon this miss, and thus the ViCL for the load of x must be created after the ViCL for the load of y is created.

The mp litmus test specifies a forbidden outcome, so the verification goal is to identify a cycle in the μhb graph. If the TSO-CC implementation properly invalidates the caches as specified, then one of the "InvCache" edges completes the required cycle. If these edges did not exist, the forbidden outcome would be observable, indicating that they are critical to maintaining TSO in TSO-CC. As in the partial incoherence example, the ViCL abstraction allows for comprehensive CCI verification, and the automatically-enumerated μhb graphs give the designer intuition about how and why correct behavior is preserved. Although the authors of TSO-CC performed testing via simulation, CCICheck is more comprehensive because it conducts exhaustive enumeration of all possible ordering scenarios, where simulation may be limited to just a few different orderings.

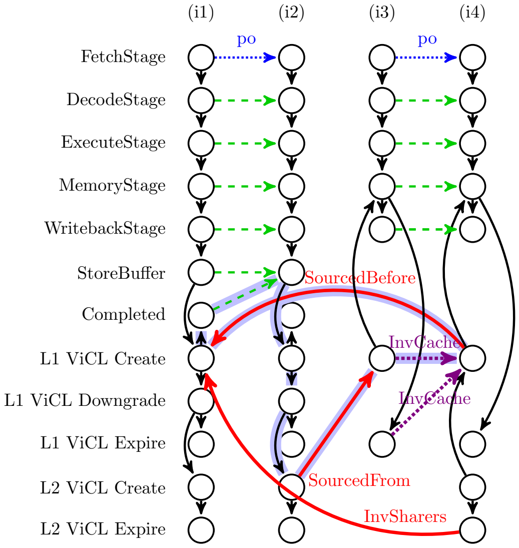


Figure 8: One possible μhb graph for mp on TSO-CC [18]. Although the coherence protocol does not eagerly invalidate any sharers of x before allowing (i1) to perform, the necessary orderings are enforced by (i3) invalidating shared data in its L1 upon a cache miss.

We also hope that CCICheck helps clarify statements such as the TSO-CC paper's claim that most coherence protocols are designed for sequential consistency [18]. As numerous widely-used architectures such as x86, ARM, and Power are considered coherent yet not sequentially consistent, it is easy to misinterpret such a claim. The coherence protocol cannot enforce the consistency model independently of pipeline components such as store buffers or of pipeline behaviors such as speculative load reordering. SC may require stronger guarantees from a coherence protocol than other memory models, but it is nevertheless up to the system as a whole to ensure that consistency is properly enforced.

6.3 Window of Vulnerability/Peekaboo

This section demonstrates the use of CCICheck to verify a solution to the Peekaboo problem previously introduced in Section 2.1. The Peekaboo solution allowed loads and stores to access already-invalidated data if and only if they were the oldest in program order at the time of the coherence request for the accessed line. The CCICheck model for Peekaboo therefore includes a

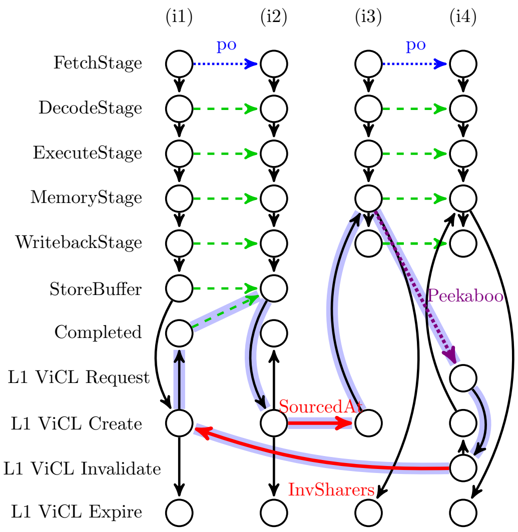


Figure 9: μhb graph for mp for the Peekaboo solution.

ViCL Request node and a ViCL Invalidate node. The ViCL Request node represents the time at which a core makes a coherence request for the cache line of a particular ViCL, and the ViCL Invalidate node represents the point at which an invalidation request for that ViCL's cache line is processed. Here, unlike the cases previously discussed, the invalidation of the ViCL is *not* the same as its expiration. This is because a load may use invalidated data for one operation if the invalidation arrives before the data response, in an effort to prevent livelock.

Figure 9 demonstrates a μhb graph for the Peekaboo solution for the mp litmus test. Consider the load of x that accessed invalidated data (henceforth referred to as the "Peekaboo" instruction). The InvSharers edge emanating from the ViCL Invalidate node of this instruction indicates that the ViCL for the load of x is invalidated before the store of x on core 0, as required by the SWMR invariant. The graph includes a μhb edge (in this case labeled "Peekaboo") to the ViCL Request node of the Peekaboo instruction from the Memory stage (for loads) or Completed stage (for stores) of all preceding accesses on that core. These edges ensure that the Peekaboo instruction is the oldest in program order when its coherence request is issued, as required by the Peekaboo solution. The presence of the Peekaboo μhb edge completes the cycle in this graph, thereby preventing the illegal outcome. Thus, CCICheck helps designers verify that the Peekaboo solution abides by TSO consistency while avoiding livelock.

Finally, we note that (unsurprisingly) the act of formally specifying CCI behavior can sharpen a designer's understanding of system behavior. For example, the original English-language description of the problem [39]—while very thorough and perhaps clear to many—required us to interpret *how far* instructions preceding the Peekaboo load (in program order) must have progressed before the coherence request was issued in order for the load to be allowed to use stale data. In creating the CCICheck model, we were able to use the tool to determine that a correct design would only allow the load to access invalidated data if all previous loads were *performed* and all previous stores had *reached the memory hierarchy and become visible to all cores* before the coherence request for the Peekaboo instruction was issued. By requiring formal and explicit documentation regarding such decisions, the CCICheck model becomes a more precise solution specification than any natural language description could be.

7. RESULTS ACROSS LITMUS TESTS

Figure 10 shows the runtimes of CCICheck across all 85 litmus tests for six of our evaluated microarchitectures. As the graph shows, there can be significant variation in the time it takes CCICheck to run a litmus test across tests and architectures. This is to be expected as litmus tests have varying numbers of instructions and varying potential for ViCL pairings, thus leading to different ranges of possibilities. For example, the sb litmus test has only four instructions and tests that store buffering is observable on TSO architectures. As such, CCICheck exits after finding one observable execution for the test, which is reflected by it finishing in under a tenth of a second across all architectures. Meanwhile, the iwp27 litmus test has 8 instructions and thus significantly more possibilities. It also checks for a forbidden outcome, which means that CCICheck has to explore all possibilities for it. These characteristics are reflected in its high runtime for certain architectures. However, on a simpler architecture like the five-stage architecture with a shared L1, CCICheck finishes the test in only 18 seconds. The geometric mean of test case execution time is under 10 seconds for *all* architectures, indicating that verification with CCICheck can be conducted relatively quickly, especially since each litmus test can be run in parallel with a different instance of CCICheck.

Much of the tractability of CCICheck comes about due to its early pruning and invalid case detection (Section 4.4). Many path combinations that CCICheck needs to verify at the outset turn out to be invalid, due to unsatisfiable constraints in their scenarios. Of the remaining valid scenarios, nearly all graphs are pruned before reaching the end of the enumeration process. Our subsequent research has identified constraint satisfaction methods for further improving runtimes.

8. RELATED WORK

Consistency Models: Consistency models have their roots in Lamport's early work regarding sequential consistency (SC) [29]. Since then, a wide range of memory models have been developed [2, 14, 15, 16, 21, 22, 40]. Recent consistency verification work embraces some formalisms for modeling or verification, but generally remains at the architecture (not implementation) level, abstracts coherence protocol behavior, and does not check the CCI [1, 3, 5, 6, 7, 10, 15, 24, 32, 36, 38, 40, 43]. Dynamic verification tools [23, 35] test the actual implementation of a system and thus can encompass the CCI, but are not exhaustive and thus cannot *conclusively* verify a consistency model implementation as CCICheck

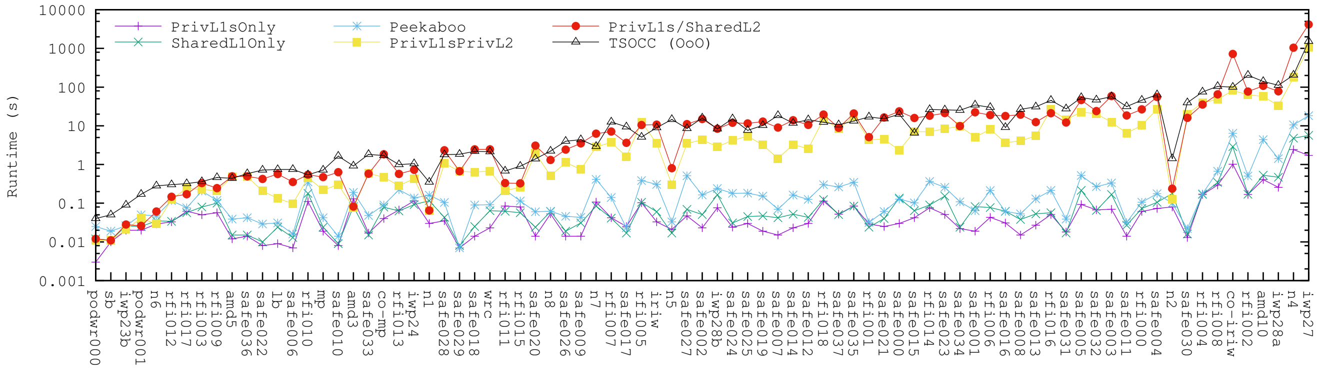


Figure 10: Runtimes of the x86-TSO litmus tests across six of the processors we modeled.

can. Unlike CCICheck, these tools also impose runtime overheads on a system.

The PipeCheck tool [30, 31] breaks through the architectural interface and verifies behavior at the (lower) microarchitectural level. While PipeCheck captures nuances in pipeline operation better than architectural models, it largely continues to abstract the behavior of the memory hierarchy and coherence protocol. Specifically, it does not model cache occupancy and coherence protocol transitions, instead relying on the notion of an abstract unified memory hierarchy and architectural-level notions like `rf` (reads-from) and `fr` (from-reads) edges. This makes it incapable of modeling scenarios where coherence and consistency interact, such as the Peekaboo problem or lazy coherence. Our work addresses this gap by applying a microarchitectural approach to CCI verification.

Coherence: The field of cache coherence protocol verification has also seen significant attention. Early work in this realm explored the verification of the FutureBus+ protocol [13], while subsequent efforts used formal languages like Murphi to check the FLASH coherence protocols [34]. More recently, Zhang et al. [44, 45] proposed techniques for designing cache coherence protocols that are more amenable to formal verification. In all these cases, however, consistency is not verified, thus retaining the isolation of coherence and consistency and ignoring the CCI. Our work seeks to bridge this gap by using the invariants guaranteed by coherence protocol verifiers as axioms for CCI-aware verification.

CCI Definition and Verification: CCI verification has seen much less attention from both coherence and consistency verifiers, in part because it is a compelling notion that coherence and consistency should be decoupled [33]. Zhang et al. [46] recommend keeping cores, coherence, and reordering mechanisms separate from each other in an implementation in order to keep verification scalable. Unfortunately, since most implementations of coherence and consistency become interwoven for performance or design reasons [4, 11, 12, 18, 27, 37, 39], CCI verification is an under-researched necessity. CCICheck's μhb graphs, ViCL abstraction, and axiomatic treatment of protocol behaviors allow it to fill the CCI verification gap.

9. CONCLUSION

In this paper, we presented CCICheck, a methodology and tool for scalable verification of the CCI (coherence-consistency interface) and consistency model implementation of a given microarchitecture. CCICheck uses μhb graphs and exhaustive enumeration of all possible executions to verify that a microarchitecture maintains a consistency model across a set of litmus tests. The key concept underpinning CCICheck's analysis is the ViCL abstraction, which allows it to model cache occupancy and coherence protocol events to the level required for verification without modeling all coherence protocol transitions. CCICheck microarchitecture definitions consist of a set of instruction paths and constraints for those paths, allowing it to handle a wide range of microarchitectures. It uses intelligent pruning and early detection of unsatisfiable constraints to reduce the number of executions it needs to enumerate. We have shown that CCICheck can be used to model CCI issues such as the Peekaboo problem, as well as partially coherent and lazily coherent architectures, both of which have complicated CCIs. CCICheck is open source and publicly available at github.com/ymanerka/ccicheck.

10. ACKNOWLEDGEMENTS

We thank Geet Sethi 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 the grant CCF-1117147).

11. REFERENCES

[1] A. Adir, H. Attiya, and G. Shurek, "Information-flow models for shared memory with an application to the PowerPC architecture," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 5, pp. 502–515, 2003.

[2] S. Adve and M. Hill, "Weak ordering: a new definition," *ISCA*, 1990.

[3] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design (FMSD)*, vol. 41, no. 2, pp. 178–210, 2012.

[4] J. Alglave, M. Batty, A. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU

concurrency: Weak behaviours and programming assumptions," *ASPLOS*, 2015.

[5] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of Power and ARM machine code," *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2009.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," *CAV*, 2010.

[7] 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)*, vol. 36, July 2014.

[8] AMD, "Revision guide for AMD family 10h processors," August 2011. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/41322.pdf

[9] AMD, "AMD64 architecture programmer's manual," 2013.

[10] Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," *ISCA*, 2006.

[11] T. J. Ashby, P. Diaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers*, vol. 60, no. 4, pp. 472–483, 2011.

[12] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," *PACT*, 2011.

[13] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness, "Verification of the futurebus+ cache coherence protocol," in *International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, 1993, pp. 15–30.

[14] F. Corella, J. M. Stone, and C. M. Barton, "A formal specification of the PowerPC shared memory architecture," *CS Tech. Report RC 18638 (81566), IBM Research Division, TJ Watson Research Center*, 1993.

[15] Digital Equipment Corporation, "Alpha architecture reference manual," 1992.

[16] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," *ISCA*, 1986.

[17] M. Elver, "TSO-CC specification," 2015. [Online]. Available: http://homepages.inf.ed.ac.uk/s0787712/res/research/tsocc/tso-cc_spec.pdf

[18] M. Elver and V. Nagarajan, "TSO-CC: consistency directed cache coherence for TSO," in *HPCA*, 2014.

[19] H. D. Foster, "Trends in functional verification: A 2014 industry study," *DAC*, 2015.

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

[21] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *ISCA*, 1990.

[22] J. R. Goodman, "Cache consistency and sequential consistency," SCI Committee, Tech. Rep., March 1989, tech Report 61. [Online]. Available: ftp://ftp.cs.wisc.edu/pub/techreports/1991/TR1006.pdf

[23] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "Tsotool: A program for verifying memory systems using the memory consistency model," in *ISCA*, 2004.

[24] Intel, "Intel Itanium architecture software developer's manual, revision 2.3," 2010. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[25] ——, "Intel 64 and IA-32 architectures software developer's manual," 2013. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[26] Intel, "Intel Xeon processor E3-1200 v3 product family, specification update," April 2015. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf

[27] L. Kontothanassis, M. Scott, and R. Bianchini, "Lazy release consistency for hardware-coherent multiprocessors," in *SC*, 1995.

[28] J. Kubiatowicz, D. Chaiken, and A. Agarwal, "Closing the window of vulnerability in multiphase memory transactions," in *ASPLOS*, 1992.

[29] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computing*, vol. 28, no. 9, pp. 690–691, 1979.

[30] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," *MICRO*, 2014.

[31] ——, "Verifying correct microarchitectural enforcement of memory consistency models," *IEEE Micro (Top Picks of 2014)*, vol. 35, no. 3, 2015.

[32] 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," *CAV*, 2012.

[33] M. M. K. Martin, "Formal verification and its impact on the snooping versus directory protocol debate," *IEEE International Conference on Computer Design (ICCD)*, 2005.

[34] K. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *Correct Hardware Design and Verification Methods (CHARME)*, 2001.

[35] A. Meixner and D. Sorin, "Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2009.

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

[37] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *PACT*, 2012.

[38] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER microprocessors," *PLDI*, 2011.

[39] D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. Hill, Ed. Morgan & Claypool Publishers, 2011.

[40] SPARC, "SPARC architecture manual, version 9," 1994.

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

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

[43] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind, "Analyzing the Intel Itanium memory ordering rules using logic programming and SAT," in *Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[44] M. Zhang, J. Bingham, J. Erickson, and D. Sorin, "PVCoherence: Designing flat coherence protocols for scalable verification," in *HPCA*, 2014.

[45] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *MICRO*, 2010.

[46] M. Zhang, A. Lebeck, and D. Sorin, "Fractal consistency: Architecting the memory system to facilitate verification," *Computer Architecture Letters (CAL)*, 2010.