47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014

PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models

Daniel Lustig
Princeton University
Princeton, NJ
dlustig@princeton.edu

Michael Pellauer
Intel Corporation
VSSAD Group
Hudson, MA
michael.i.pellauer@intel.com

Margaret Martonosi
Princeton University
Princeton, NJ
mrm@princeton.edu

***Abstract*—We present PipeCheck, a methodology and automated tool for verifying that a particular *microarchitecture* correctly implements the consistency model required by its architectural specification. PipeCheck adapts the notion of a "happens before" graph from architecture-level analysis techniques to the microarchitecture space. Each node in the "microarchitecturally happens before" (μhb) graph represents not only a memory instruction, but also a particular location (e.g., pipeline stage) within the microarchitecture. Architectural specifications such as "preserved program order" are then treated as propositions to be verified, rather than simply as assumptions.**

PipeCheck allows an architect to easily and rigorously test whether a microarchitecture is stronger than, equal in strength to, or weaker than its architecturally-specified consistency model. We also specify and analyze the behavior of common microarchitectural optimizations such as speculative load reordering which technically violate formal architecture-level definitions. We evaluate PipeCheck using a library of established litmus tests on a set of open-source pipelines. Using PipeCheck, we were able to validate the largest pipeline, the OpenSPARC T2, in just minutes. We also identified a bug in the O3 pipeline of the gem5 simulator.

I. Introduction

Multicore processors rely on *memory consistency models* to ensure the correctness of all inter-core communication performed through shared memory spaces. Unfortunately, definitions of consistency models often suffer from a lack of formalism, as they are often specified using natural language instead of mathematical models. While ongoing work has seen some success in formalizing memory models, such work intentionally abstracts away the details of the microarchitecture itself, rendering such models unable to describe how the architectural requirements are actually enforced. In particular, architecture-level models do not address the reordering implications of instructions performing at multiple points in the pipeline, and they do not always account for microarchitectural realities such as speculative load reordering that technically violate the formal reordering rules [14].

We present PipeCheck, a formal methodology for verifying that a given *microarchitecture* meets the specifications of a given architectural consistency model. PipeCheck first defines within a pipeline a set of locations (e.g., stages) at which one can make *local* declarations about the ordering of memory instructions passing through that location. Given these, PipeCheck constructs a global graph of "microarchitecturally happens before" (μhb) edges. Our work is the first to apply state-of-the-art consistency model analysis techniques to the microarchitecture level. PipeCheck verifies that each ordering edge that must be preserved according to the architectural consistency model (e.g., each Store→Store ordering for Total Store Ordering (TSO)) is in fact provably maintained by the microarchitecture. As a result, PipeCheck reduces the problem of verifying consistency model implementation correctness to the more tractable problem of verifying local reordering properties at various points in the microarchitecture.

We implement PipeCheck using Coq [46], a proof verification assistant, to make our code amenable to formal analysis and for easier integration with existing architecture-level models [3]. We then use built-in Coq functionality to automatically extract the computational portion into a standalone automated OCaml tool.

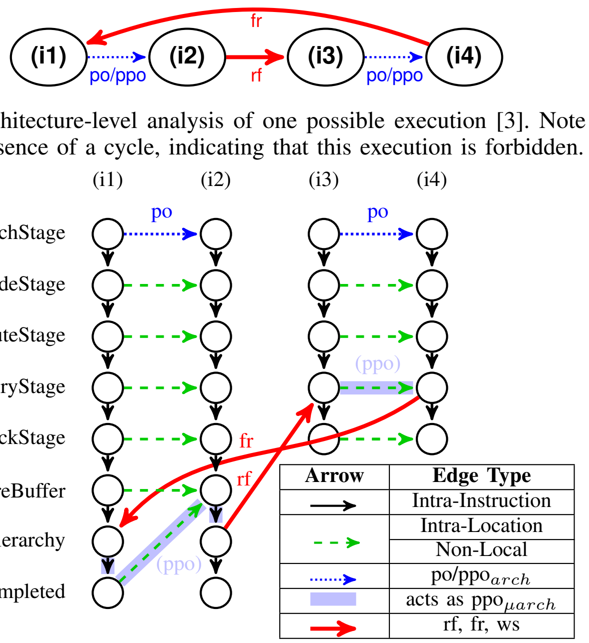
Our contributions are as follows:

- We are the first to develop an automated methodology and tool, PipeCheck, in which a given microarchitecture can be validated against its architecture-level consistency model specification. In particular, PipeCheck defines "preserved program order" as a *proposition* to be proven rather than merely as an architectural assumption.
- PipeCheck calculates if the consistency model implemented by a pipeline is weaker than, equal to, or stronger than its architectural consistency model.
- PipeCheck verifies microarchitectural optimizations such as speculative load reordering which, although in widespread use, technically violate existing formal architecture-level consistency model specifications.
- We reconcile our results with existing architecture-level analyses, including a large body of litmus tests.
- We use PipeCheck both to verify the correctness of the OpenSPARC T2 processor with respect to its consistency model and to find a bug in the implementation of the gem5 O3 simulated pipeline. Both analyses are able to run to completion in just minutes.

The rest of the paper is organized as follows. Section II describes a motivating example. Section III gives background on architecture-level models, while Section IV describes the PipeCheck microarchitecture-level approach. Our analysis methodology and tool flow is described in Section V.

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

(a) Litmus Test Code



(b) Architecture-level analysis of one possible execution [3]. Note the presence of a cycle, indicating that this execution is forbidden.

(c) PipeCheck eliminates `ppo` as an assumption, and instead checks that it is replaced by calculated edge(s). In this example, the gray highlighted edges replace the `ppo` edges and complete a cycle.

Fig. 1: Load→Load and Store→Store ordering litmus test `iwp2.1/amd1/mp`.

Sections VI and VII highlight important case studies and summarize overall results, respectively. Finally, Section VIII describes related work, and Section IX concludes.

II. Motivating Example

We begin with a brief summary of existing architecture-level modeling techniques, the gaps in these analyses, and an example of how PipeCheck fills in those gaps.

A. Architecture-Level Analysis

Formal architecture-level hardware models [3, 37] (and software models, e.g., in C/C++11 [28, 29]) generally represent each program as a graph in which vertices represent memory instructions in the program. Edges between these vertices represent ordering relationships between the source and destination of the edge: an edge from an instruction s to another instruction d indicates that s *happens before* d, in some formal sense defined by the model.

Figure 1a gives an example architectural analysis of a litmus test named in different sources as `iwp2.1`, `amd1`, or `mp`. A *litmus test* is a program written for the purpose of testing a consistency model. This particular test asks whether there is some execution of the two threads that produces the result r1=1 and r2=0 on a processor following the TSO consistency model. All memory locations are assumed to hold the value 0 originally. If the pipeline were sequentially consistent, this would be equivalent to asking whether there were such an interleaving of the threads. However, for performance reasons, most modern processors use more permissive consistency models that allow for instruction reordering, cores reading writes early, and so on. As a result, interleaving analysis is not sufficient.

Broadly, we can classify "happens before" edges into two groups that subsequent sections will address separately [24]. The first consists of *static edges*: those derived from the program itself. This includes preserved program order and fence edges (see Sec. III). The second class, *observed edges*, are those observed during a particular execution. As a given program may have multiple legal executions, there will generally be many possible graphs for a given program, and each graph will have its own set of observed edges. The model developed by Alglave [3] highlights three types of observed edge (s, d), which we summarize here (informally):

- "reads from" (`rf`): if d reads from s, then s must happen before d, from at least some points of view
- "write serialization" or "coherence order" (`ws`): s comes before d in the (assumed) total ordering of all memory instructions accessing a given location, from the point of view of the memory hierarchy
- "from reads" or "reads before" (`fr`): s is a read that gets its value from a write that comes before d in the set of `ws` edges

To calculate the edges in Figure 1b, one can work backwards from the proposed result. Since r1 is hypothesized to hold the value 1, instruction (i3) must have read the value written by instruction (i2). Hence, (i2) must have *happened before* (i3). This establishes the "reads from" (rf) edge in Figure 1b. Similarly, (i4) must have happened before (i1), because otherwise (i4) would also have returned the value 1. This translates into the "from reads" (fr) edge in Figure 1b. At the architecture level, TSO itself guarantees the two "preserved program order" (ppo) constraints shown in the graph to indicate that Load→Load and Store→Store orderings within a thread must both be preserved. Now, given these four edges, there is a cycle in the ordering graph of memory instructions in Figure 1b. Such a cycle indicates that an instruction can somehow happen before itself. Since this is clearly impossible, the execution is ruled out. In other words, the proposed outcome r1=1 and r2=0 is in fact forbidden under TSO.

B. Microarchitecture-Level Analysis: PipeCheck

The analysis in Figure 1b says nothing about the behavior of any individual *microarchitectural implementation* of that architecture. On one hand, certain architecturally-permitted behaviors may not be observable on a given microarchitecture. For example, a sequentially consistent (SC) pipeline is a valid implementation of a TSO architecture, although many executions that are legal under TSO will not be observable in such a pipeline—the microarchitectural memory model is stricter than the architecture requires. On the other hand, architecturally-forbidden behaviors may be observable on a

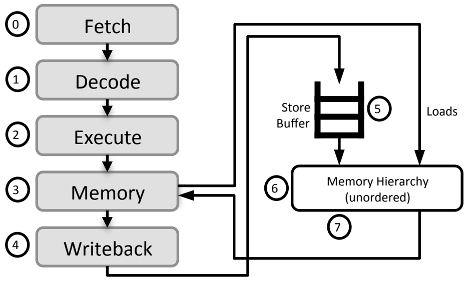


Fig. 2: Classic Five-Stage RISC Pipeline plus a store buffer and an unordered memory system. This pipeline example recurs throughout this paper.

given microarchitecture—this would correspond to a bug in the implementation. In this case, the microarchitecture is erroneously weaker than the architecture requires.

As a running example of a microarchitecture, we will use the classic RISC five-stage pipeline, augmented with a store buffer, as shown in Figure 2. For clarity, we assume here that this microarchitecture has no cache; in later sections we will describe how to model cache behavior in pipelines for which consistency and coherence are interdependent. Furthermore, in this example we assume an unordered memory system. This highlights an important point: even for an in-order pipeline, the presence of an unordered network and/or memory hierarchy can reorder memory accesses from the point of view of other processors. As a result, PipeCheck must also account for such behavior. Although we use a simple example here to build intuition, we verify more complex processors, including OpenSPARC T2, later in the paper.

Figure 1c shows the edges from Figure 1b as translated into the PipeCheck model of the microarchitecture of Figure 2. The four memory operations, (i1), (i2), (i3), and (i4), are depicted from left to right, and various stages in the microarchitecture are shown from top to bottom. Specifically, each vertex corresponds not just to a memory instruction, but also to a particular *location* within the pipeline or memory system. Each column of vertices therefore corresponds to that instruction progressing through the various locations in the microarchitecture. The various edge types will be described in detail in later sections.

While ws edges are defined with respect to main memory, PipeCheck maps the endpoints of rf and fr edges to the points in the pipeline at which the instructions can be considered to have *performed* with respect to the core at the other endpoint. Traditionally, a store from core i is said to have performed with respect to core j when a load of the same address issued by core j returns the value written by the store (or by some subsequent store) [19, 40]. A load from core i has performed with respect to core j when a subsequent store from core j to the same address cannot affect the value returned by the load. A store or a load has *performed globally* when it has performed with respect to all cores. More recent studies, however, have criticized this definition as being too hypothetical for formal analysis [4, 35].

From a microarchitectural point of view, Section IV-A will therefore unambiguously define "perform" in terms of *locations* in the pipeline. Continuing the example, the rf edge from (i2.MemHierarchy) to (i3.MemoryStage) indicates that instruction (i2) must have performed with respect to core 1 (i.e., been written back from core 0 to the memory hierarchy) before instruction (i3) performs with respect to core 0 (i.e., reads memory from the Memory stage of the core 1 pipeline).

In contrast, we treat ppo edges as *propositions* rather than assumptions: we want to check that a pipeline properly maintains these edges, rather than simply assuming their presence. In fact, we consider ppo edges to be merely some orderings that must be guaranteed among many orderings that are guaranteed at various locations within a pipeline. In Figure 1c, we draw (in dashed green) orderings that are maintained at each location; this calculation will be the focus of Section IV. In this particular example, with an in-order pipeline, each pipeline stage maintains at its output the instruction ordering it observes at its input. This results in the po orderings being maintained throughout the pipeline, with the exception of writes sent to memory. The ordering of stores at the memory hierarchy is instead maintained by the store buffer via the diagonal edge from (i1).Completed to (i2).StoreBuffer—a detail we will come back to later.

Consider the ppo edges that were part of the cycle in Figure 1b. While their presence was assumed in the architecture-level analysis, PipeCheck does not assume them; it *checks* for them. The ppo edge from (i1) to (i2) is enforced by a sequence of three microarchitecture-level "happens before" edges, as shown highlighted in gray in Figure 1c. The ppo edge from (i3) to (i4) is enforced by a single microarchitecture-level edge. Together with the observed microarchitecture-level rf and fr, the union of the highlighted edges forms the microarchitecture-level equivalent of the cycle in Figure 1b. Thus, at least for this particular example, the ppo edges are correctly maintained, and hence the pipeline correctly implements the TSO restrictions on this litmus test. While this example covers only one test, later sections will describe the process of fully verifying all possible orderings.

III. Background: Architectural Analysis

To verify a microarchitectural implementation, we must first formally define the requirements of the architecture-level consistency model against which the microarchitecture will be verified. In this section, we survey existing work and describe how PipeCheck models the architecture-level specification.

Preserved Program Order. *Preserved program order* defines the set of reorderings guaranteed by the architecture not to occur, based on the types of the accesses. Industry specifications often present this information in the form of a large table like Figure 3a. (See also Table 7.3 of [9], Table 9.2 of [36], or Sec. 8.2.2 of [26].) For simplicity, as in other memory consistency studies, we omit "non-

		Second op.		
		Load	**Store**	**...**
First op.	**Load**	Y	Y	...
	Store	N (mfence)	Y	...
	...	...	...	...

(a) Preserved Program Order (PPO). The fence in parentheses indicates the fence type used to restore the otherwise-missing ordering.

Subset of Program Order	Locally	Remotely
Program Order & same address	Y	N
Preserved Program Order	N	Y

(b) Architecturally-Required Orderings

Property	Value
Read own writes early	Permitted
Read other cores' writes early	Forbidden

(c) Other Properties

Fig. 3: Definition of TSO. Atomics/fence types may vary.

standard"[1] reads and writes in this paper, leaving four cases to consider: Load→Load reorderings, Load→Store reorderings, Store→Load reorderings, and Store→Store reorderings. The PipeCheck approach of considering ppo edges as propositions rather than assumptions is described in Section IV-D.

Uniprocessor Orderings. Cores may reorder local operations freely as long as the architecturally-observable behavior maintains correct single-thread semantics. In particular, from the point of view of the issuing core, accesses to different addresses are considered entirely independent. Accesses to the same address must perform locally in order, but they *need not* always be enforced from the point of view of other processors. This property is tested explicitly by permitting the proposed outcome in the n7 litmus test [37]. These same-location program ordering properties, which we refer to as po-loc [3], are summarized in Figure 3b. The PipeCheck approach to verifying po-loc is described in Section IV-D.

Dependencies. Certain subsets of program order are considered to contain *dependencies* between instructions. Some architectures distinguish between address, control, and data dependencies for architectures on which it matters [7, 8, 33, 39]. Other studies abstract away the details of dependencies, instead either considering them to be architecture-independent or instantiating more precise definitions later on a case-by-case basis [3]. Dependency orderings are currently enforced in PipeCheck through explicit microarchitectural rules, allowing for easy adaptation to any particular instantiation.

Reading Writes Early. Many non-SC models allow a processor to read its own and/or other processors' writes "early", i.e., "before the write has been committed to memory". For example, TSO cores may read only their own writes early, but may not read other cores' writes early. Sarkar et al. model this behavior through *partial coherence* operations and explicit intra-core forwarding rules [39]. Alglave formalizes it by describing the subset of "reads from" edges which are excluded from the global "happens before" consensus [3]. PipeCheck handles this situation by defining multiple *performing locations* (Section IV-A), thereby causing writes to perform with respect to different cores at different locations within a pipeline.

Atomic Operations and Memory Fences. Essentially all modern processors provide native atomic and fence operations for the purpose of making inter-core synchronization practical. Details vary by architecture, even among those with otherwise similar consistency models. In this paper, for clarity and due to space constraints, we do not aim to describe all possible atomic operations or fences. We instead describe the implementation of a Store→Load fence in Section VI-C, and we leave the analysis of other possibilities for future surveys.

IV. PipeCheck Microarchitecture-Level Analysis

A central observation of PipeCheck is that orderings between instructions are often too complicated to be captured by a single architecture-level "happens before" edge. A single pair of instructions may *fetch* in order, *issue* out of order, *execute* in order, *commit* in order, and *perform globally* out of order. Restricting "happens before" edges to specify only orderings with respect to the memory hierarchy ignores all of the other important memory orderings that take place within the pipeline itself. PipeCheck therefore defines "microarchitecturally happens before" (μhb) edges to specify both an instruction and a particular *location* within the pipeline:

Definition 1 (Microarchitecturally Happens Before)**.** *The* μhb *graph is a directed graph* (V, E) *in which each vertex* $(inst.loc) \in V$ *represents a memory instruction* $inst$ *passing through a particular location* loc*, and each edge* $(inst_i.loc_a, inst_j, loc_b)$ *represents a guarantee that instruction* $inst_i$ *passes through location* loc_a *before instruction* $inst_j$ *passes through location* loc_b*.*

Throughout this paper, we depict μhb graphs in a grid, as in Figure 1c, with instructions along the x-axis and microarchitectural locations along the y-axis. Not all instructions pass through all locations (e.g., loads do not occupy the store buffer), and so some entries in the grid are left empty. Despite the grid depiction, only relationships depicted by arrows provide any ordering guarantee.

A. Microarchitecture Definition

In PipeCheck, a microarchitecture is defined by:

- A list of the *locations* of interest in the microarchitecture
- Legal *path(s)* per instruction type.
- *Performing locations* within each path
- The *local reordering guarantee* at each location
- *Non-local edges*, i.e., edges which are both inter-instruction and inter-location

These terms are more carefully defined below.

During execution, as instructions flow through the pipeline, they pass through the chosen locations along some well-defined *path*. A memory instruction may have more than one legal path through a pipeline, depending on the type of instruction, the state of the pipeline, and/or the state of the

[1] E.g., for space and clarity reasons, for x86, we do not consider non-cacheable reads, write-combining accesses, etc., as these are not used in most user-level code. Atomics are considered separately in Section III.

memory system during execution. For example, a read may take a different path depending on whether it performs by reading from the store buffer, from the cache via a cache hit, or from the cache after a cache miss.

Each path also defines the set of *locations* at which each instruction can *perform*. PipeCheck defines "perform" in terms of location rather than potential visibility, maintaining the intention of the traditional definition given in Section II-B while removing its hypothetical nature:

Definition 2 (Performing Location)**.** *A location l is a performing location with respect to core c if:*

- *a load at location l can read the value written by a store from core c*
- *the data being written by a store at location l is visible to core c.*

A location l is a global performing location *if it is a performing location with respect to all cores.*

This definition also addresses the open question of how/whether a load from one core can be said to "happen before" a read from another core. For example, A-cumulativity is a Power architecture [25] concept that instructions from any core that have performed with respect to core i before a fence is executed by core i are ordered by that fence. Previous architecture-level studies [4, 35] were unable to formalize this notion, as they did not consider cores to observe reads from other cores. In PipeCheck, this would simply correspond to the presence or absence of the relevant μhb edge.

To more carefully define "in order" and "out of order" sections of a pipeline, we define a *local reordering guarantee* at each location. This specifies the reorderings that location does or does not permit on instructions passing through it. At one extreme, a *FIFO* local reordering specifies that all inter-instruction orderings guaranteed at entry into a location will also be guaranteed leaving that location. At the other extreme, a *NoGuarantees* local reordering specifies that no orderings are guaranteed for instructions leaving the location. Other guarantees may lie in between. The specific guarantees of each pipeline stage will vary from processor to processor.

Location Definitions				Path Defs.		
#	Location	Local Reord. Guarantee	NLE	Load		Store
				M	S	
0	Fetch	FIFO	-	Y	Y	Y
1	Decode	FIFO	-	Y	Y	Y
2	Execute	FIFO	-	Y	Y	Y
3	Memory	FIFO	-	Y^G	Y^G	Y^L
4	Writeback	FIFO	-	Y	Y	Y
5	Store Buffer	FIFO	SB			Y
6	Mem. Hierarchy	NoGuarantees	-			Y^G
7	Completed	-	-			Y

L: local performing location. G: global performing location.
Load-M: path in which load reads from memory.
Load-S: path in which load reads from the store buffer.
SB: only one store can be outstanding from the store buffer at a time: for all stores s, for the immediately subsequent store s', $(s.\text{Completed}) \xrightarrow{\mu hb} (s'.\text{StoreBuffer})$.

TABLE I: PipeCheck definition of the Five-Stage RISC Pipeline of Figure 2. "NLE" means non-local edges.

Algorithm 1 Generating a μhb graph: StaticEdges

```
$Graphs_{\mu hb,SE} = \emptyset$
$Scenarios = CrossProduct(Paths[inst_0], \ldots, Paths[inst_n])$
for all $ScenarioPaths \in Scenarios$ do
    for all $inst_i$ do    // Intra-Instruction Edges
        for all $0 \le j < len(ScenarioPath[inst_i]) - 1$ do
            $G_{\mu hb,SE} = G_{\mu hb,SE} \cup \{(inst_i.ScenarioPath[inst_i][j], inst_i.ScenarioPath[inst_i][j+1])\}$
    for all $inst_i$, $i > 0$ do    // Program Order
        if $core(inst_{i-1}) = core(inst_i)$ then
            $G_{\mu hb,SE} = G_{\mu hb,SE} \cup \{(inst_{i-1}.Fetch, inst_i.Fetch)\}$
    for all $loc_b$ do    // Intra-Location Edges
        for all $loc_a < loc_b$ do
            $G_b = \emptyset$
            if $(inst_i.loc_a, inst_j.loc_a) \in G_{\mu hb,SE} \wedge$
            $(inst_i.loc_a, inst_i.loc_b) \in G_{\mu hb,SE} \wedge$
            $(inst_j.loc_a, inst_j.loc_b) \in G_{\mu hb,SE}$ then
                $G_b = G_b \cup (inst_i.loc_b, inst_j.loc_b)$.
            $G_{\mu hb,SE} = G_{\mu hb,SE} \cup$
                $\{LocalReorderingGuarantees(G_b)\}$
    for all $loc_a$ do    // Per-Location Non-Local Edges
        $G_{\mu hb,SE} = G_{\mu hb,SE} \cup NonLocalEdges(loc_a)$
    $Graphs_{\mu hb,SE} = Graphs_{\mu hb,SE} \cup \{G_{\mu hb,SE}\}$
return $Graphs_{\mu hb,SE}$
```

Table I shows the PipeCheck definition of the classic RISC pipeline of Figure 2. The rows of the table are microarchitectural locations defined by the properties in the first four columns. The last three columns define the possible paths each class of instructions can take through the pipeline: each path passes through the microarchitectural locations indicated by the "Y" entries. The definition explicitly defines the local reordering guarantees at each location, as well as a set of non-local edges specific to the store buffer. The list of paths also clearly defines the set of locations at which each memory instruction performs with respect to other cores (the L and G superscripts), as determined by the sets of physical wires in the implementation. Note that these performing locations are the locations chosen to be the endpoints of the `rf` and `fr` edges of Figure 1c. Given a microarchitecture definition of this form, and given a program binary, PipeCheck can directly calculate the complete set of static edges in the μhb graph as described in the next section.

B. Generating μhb Graphs

Static Edges. Algorithm 1 gives the full procedure for generating the static edges (as defined in Section II) in a μhb graph. We begin by adding a set of *intra-instruction μhb* edges between consecutive locations along the path for that instruction. For example, an instruction being in the fetch stage will "microarchitecturally happen before" the point when that same instruction is in the decode stage. These are the solid black vertical arrows in Figure 1c.

Second, each location observes instructions passing through in some order. For example, we assume *program order* to be the ordering of instructions at the fetch stage of the pipeline. Some subsequent pipeline stages also guarantee to maintain intra-location ordering guarantees from previous stages. We add *intra-location μhb* edges to represent these per-location

Algorithm 2 Generating a μhb graph: ObservedEdges

```
$WS = \emptyset.\ RF = \emptyset.\ FR = \emptyset.$
for all $addr$ do    // Write Serialization
  for all $i \in AllInterleavings(Stores(addr))$ do
    for all $inst_j, inst_{j+1} \in i$ do
      $WS[addr][i] = WS[addr][i] \cup \{(inst_j.MemHierarchy, inst_{j+1}.MemHierarchy)\}$
for all $inst_j \in Loads$ do    // Reads-from
  $RF[inst_i] = \emptyset$
  for all $inst_i \in Stores$ such that $addr(inst_i) = addr(inst_j)$ and $data(inst_i) = data(inst_j)$ do
    for all $loc_a \in path[inst_i]$ such that $loc_a$ performs with respect to $core(inst_j)$ do
      for all $loc_b \in path[inst_j]$ such that $loc_b$ performs with respect to $core(inst_i)$ do
        $RF[inst_i] = RF[inst_i] \cup \{(inst_i.loc_a, inst_j.loc_b)\}$
$Graphs_{\mu hb} = \{G_{\mu hb,SE} \cup e, \forall e \in CrossProduct(WS[addr_0], \ldots, WS[addr_m], RF[inst_0], \ldots, RF[inst_n])\}$
for all $G \in Graphs$ do    // From-reads
  for all $(inst_i.loc_a, inst_j.loc_b) \in rf(G)$ do
    for all $inst_k$ such that $(inst_i, inst_k) \in ws(G)$ do
      for all $loc_b \in path[inst_j]$ such that $loc_b$ performs with respect to $core(inst_k)$ do
        for all $loc_c \in path[inst_k]$ such that $loc_c$ performs with respect to $core(inst_j)$ do
          $G = G \cup \{(inst_j.loc_b, inst_k.loc_c)\}$
return $Graphs_{\mu hb}$
```

guarantees. These are the dashed green horizontal arrows in Figure 1. When the ordering at a location cannot be guaranteed, no such μhb edges can be drawn.

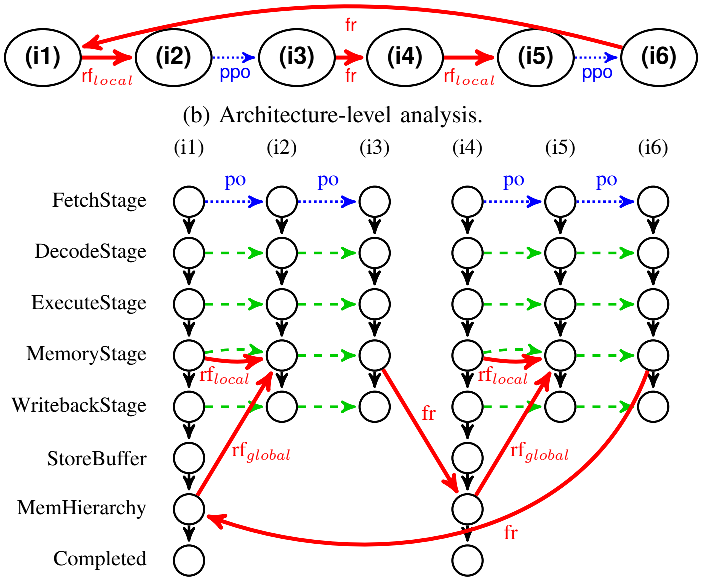
Third, we use additional *non-local* (i.e., inter-location and inter-instruction) μhb edges to model any ordering guarantees implemented by the pipeline across multiple instructions and locations. Such non-local μhb edges are relatively rarer; they correspond to non-local wires and/or communication across a chip, making them expensive in practice. However, they often do serve to enforce critical ordering guarantees. An example of such a non-local edge would exist in store buffers that enforce that "after issuing a request to memory, the store buffer must await an acknowledgment from memory before issuing a subsequent request". This is the diagonal dashed green edge from (i1).Completed to (i2).StoreBuffer in Figure 1c.

Observed Edges. In architecture-level models, each observed edge is defined to exist between certain pairs of memory instructions, but these edges are not specified to correspond to any particular locations within the pipeline. PipeCheck defines the endpoints of observed edges to be at the *performing location(s)* (Sec. IV-A) of each instruction's path. When there is more than one possibility, (e.g., a load can read either from the store buffer or from memory), PipeCheck exhaustively enumerates all cases and considers them separately.

The algorithm for enumerating all possible sets of executions (i.e., all sets of observed edges for a given program) is shown in Algorithm 2. PipeCheck first calculates all possible serializations of write instructions at the memory hierarchy (WS[addr] for each address) and calculates all writes (or the initial value) of the same address and value that each read may have read from (RF[inst] for each instruction). It then takes the cross product of these to fully enumerate the set of μhb graphs required to analyze the given program. Finally, for each case, it uses the `rf` and `ws` edges to calculate the microarchitectural `fr` edges for that scenario.

Core 0	Core 1
(i1) [x] ← 1	(i4) [y] ← 1
(i2) r1 ← [x]	(i5) r3 ← [y]
(i3) r2 ← [y]	(i6) r4 ← [x]
Allow: r1=1, r2=0, r3=1, r4=0	

(a) Litmus test `iwp2.4/amd9`



(b) Architecture-level analysis.

(c) μhb graph for the five-stage RISC microarchitecture.

Fig. 4: Analyzing litmus test `iwp2.4/amd9`.

Enumeration of all `ws` possibilities is necessary for correctness. As demonstrated by litmus test `n5` [37], if `ws` edges are not enumerated, then the `fr` edges that form the cycles that forbid the result are not generated, as their presence is calculated depending on `ws`. As a result, we hypothesize that there is a total order on instructions passing through *every* location in each core, and that this order needs to be fully enumerated at points at which paths from different cores are merged. Although our approach is general, the results of Section VII focus on TSO processors, which (due to multi-copy atomicity) have only one join point—the memory hierarchy. Our total order hypothesis therefore corresponds exactly to existing architecture-level definitions of `ws` [3, 7, 8]. More generally, weakly-ordered cores may have multiple join points (e.g., store buffers shared by some but not all cores), and so our hypothesis would require enumeration of ordering possibilities besides just `ws`.

C. Transitivity of μhb Edges

A major benefit of PipeCheck's μhb graphs over architecture-level approaches is that it avoids the problem faced by these existing models that certain edges need to be treated specially. For example, Figure 4a shows a litmus test which demonstrates that the presence of store buffering can be observed by software. Despite this reality, Figure 4b shows that architecture-level analysis would produce a cyclic graph.

Algorithm 3 `ppo/po-loc` Satisfaction Tests

```
for all $(inst_i, inst_j) \in ppo_{arch}$ do
    $Graphs_{\mu hb,SE} = StaticEdges(inst_i, inst_j)$    // Alg. 1
    $Graphs_{\mu hb} = \{ObservedEdges(G_{\mu hb,SE}),$
                $\forall G_{\mu hb,SE} \in Graphs_{\mu hb,SE}\}$    // Alg. 2
    for all $G_{\mu hb} \in Graphs_{\mu hb}$ do
        if $cyclic(G_{\mu hb}) \vee (s,d) \in G^{+}_{\mu hb}$ then    // $^{+}$: Trans. Clsr.
            continue
        if $LegalOptimization(G_{\mu hb}, (s,d))$ then    // Sec. IV-E
            continue
        return FAIL
return PASS
```

Algorithm 4 Litmus Tests

```
$Graphs_{\mu hb,SE} = StaticEdges(inst_i, inst_j)$    // Alg. 1
$Graphs_{\mu hb} = \{ObservedEdges(G_{\mu hb,SE}),$
            $\forall G_{\mu hb,SE} \in Graphs_{\mu hb,SE}\}$    // Alg. 2
$obs$ = NOT_OBSERVABLE
for all $G_{\mu hb} \in Graphs_{\mu hb}$ do
    if $acyclic(G_{\mu hb})$ then
        $obs$ = OBSERVABLE
        break
if $exp$ = PERMITTED $\wedge$ $obs$ = NOT_OBSERVABLE then
    return PASS    // Pipeline stricter than necessary
else if $exp$ = FORBIDDEN $\wedge$ $obs$ = OBSERVABLE then
    return FAIL    // Pipeline bug
else
    return PASS    // Matches expected outcome
```

Although a cycle may seem to forbid the observable outcome, as an instruction cannot happen before itself, Alglave [3] considers only cycles formed entirely of *global* edges to forbid an outcome; $\texttt{rf}_{local}$ edges are ignored for this purpose. Later improvements by Alglave refine this local vs. global distinction to check for acyclicity or irreflexivity of various subsets of the graph [8], but in no case can all edges be treated equally.

Figure 4c depicts the PipeCheck approach. With the extra location information in the graph, $\texttt{rf}_{local}$ and $\texttt{rf}_{global}$ map to distinct but equal-strength edges. In this graph, it is clear that while the μhb equivalents of the $\texttt{rf}_{global}$ edges still create a cycle, the equivalents of the $\texttt{rf}_{local}$ edges do *not*. Neither μhb edge requires special treatment.

As a consequence of this strength equality, it is legal to take the transitive closure of μhb edges. At the microarchitecture level, transitivity is respected because each μhb edge represents either a local ordering *at a particular microarchitectural location* or a communicated message (i.e., a Lamport clock [31] "happens before"). Similarly, just as architecture-level cycles do not imply microarchitecture-level cycles in Figure 4, causality (if A causes B, and core i observes B, then it must have observed A) is *not* a consequence of μhb transitivity. This is verified by the `rwc` (read-write causality) litmus test in Section VII. In particular, this ensures that PipeCheck supports non-causal architectures (e.g., Power [25]).

D. Verification Flow

This section presents two ways in which μhb graphs can be used to verify the correctness of a pipeline implementation.

ppo/po-loc Satisfaction Tests. As described in Section III, PipeCheck treats `ppo` and `po-loc` edges as propositions rather than assumptions. To verify whether these edges are satisfied by μhb edges (or a legal microarchitectural optimization described in Section IV-E), PipeCheck enumerates the full set of possible μhb graphs for each pair of instructions that the architectural specification requires be preserved. As shown in Algorithm 3, PipeCheck ensures that each `ppo` or `po-loc` edge is present in the transitive closure of each μhb graph.

Litmus tests. Litmus tests are a standard tool used by a large body of related work on consistency models. As we have already described, they are (usually very short) programs designed to test particular rules or subcases for a consistency model. Given a program, they state whether a proposed outcome is permitted or forbidden under the rules of the model. The program can then be executed on a given microarchitecture, and the proposed outcome may or may not then be observed. A permitted but unobserved outcome may mean that the pipeline is stronger than strictly necessary, but it may also simply mean that the execution produced a different legal outcome. A forbidden but observed outcome, however, indicates either a pipeline bug or an incorrect specification. Algorithm 4 presents the complete procedure for checking litmus tests.

Due to the inherent complexity of defining even simple consistency models and/or due to incomplete or even incorrect documentation, even programs as short as five or six instructions (e.g., `amd6/iriw` [14], `n4/n5/n6` [37], A-cumulativity tests [4], `coRSDWI/mp+dmb+fri-rfi-ctrlisb` [8]) can be very difficult to analyze properly. An important sanity check is therefore to verify that our analyses match both behaviors described formally in previous work as well as those observed on real processors.

Runtime. The number of graphs each algorithm enumerates varies with the number of instructions in the test and the size of each pipeline. While potentially exponential in the worst case, the absolute numbers for each quantity are very small: long pipelines contain just dozens of stages, and the longest litmus test of the suite we survey contains eight instructions. Furthermore, the algorithm can terminate early if it finds an observable outcome. Finally, the graph checking can be trivially parallelized. This ensures that PipeCheck remains feasible and fast in practice. Section VII will confirm this by measuring both the number of enumerated graphs per test, as well as the total runtime for each test.

E. Discussion: Advanced Techniques

Many microarchitectures use various forms of speculation to improve performance. PipeCheck models this simply by not including incorrectly-speculated events in μhb graphs. If a squashed instruction is replayed, then that replay will be included (unless it is itself squashed). If a squashed instruction is not replayed, then it simply does not appear. Squashes also imply μhb behavior. For example, if execution of $inst_a$ squashes a speculatively-executed $inst_b$, then

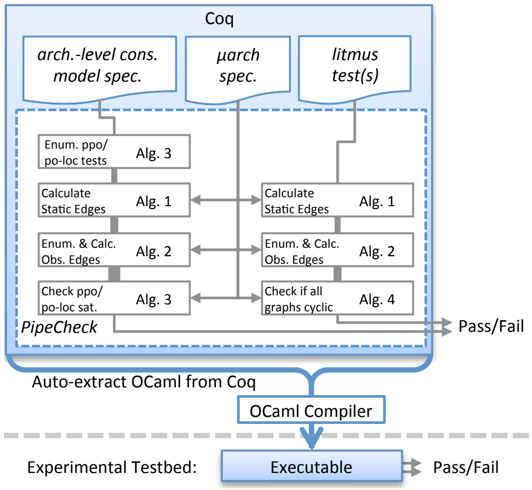


Fig. 5: PipeCheck block diagram and toolflow. Increasing line widths represent enumerations of new cases.

Pipeline	Lines of code	# Locations (c: # cores)	# Paths	
			Ld	St
RISC 5-Stage (w/o SB)	37	$5c$	1	1
RISC 5-Stage (Tab. I)	62	$6c + 2$	2	1
gem5 O3 (Tab. III)	106	$9c + 2$	2	1
OpenSPARC T2	115	$14c + 1$	3	1

TABLE II: Microarchitectures analyzed in this paper. "# Locations" means the number of locations/stages in the modeled pipeline. "# Paths" indicates the number of possible paths each instruction type can take through the pipeline.

Location Definitions				Path Defs.	
#	Location	Local Reord.	NLE	Load	Store
0	Fetch	FIFO	-	Y	Y
1	Decode	FIFO	-	Y	Y
2	Rename	FIFO	-	Y	Y^L
3	Issue	NoGuarantees	Dep	Y	Y
4	Execute	NoGuarantees	S0-S2	Y^G	Y
5	Cache Line Inv.	NoGuarantees	-		
6	Writeback	NoGuarantees	-	Y	Y
7	Commit	RestoreOrderAt 2	-	Y	Y
8	Store Buffer	FIFO	SB		Y
9	Mem. Hierarchy	NoGuarantees	-		Y^G
10	Completed	-	-		Y

L: local performing location. G: global performing location.

Dep: dependencies are enforced at the issue queue: for all loads l, for subsequent dependent instructions i, $(l.\text{Execute}) \xrightarrow{\mu hb} (i.\text{Issue})$.

S0: loads execute before the cache line they read is invalidated: for all loads l, $(s.\text{Execute}) \xrightarrow{\mu hb} (s.\text{CacheLineInvalidate})$.

S1: loads execute before the cache line read by any subsequent load to the same address is invalidated: for all loads l, for subsequent loads l' to the same address, $(s.\text{Execute}) \xrightarrow{\mu hb} (s'.\text{CacheLineInvalidate})$.

S2: check for store-load violations (in MemDepUnit/StoreSet): for all stores s, for subsequent loads l to the same address, $(s.\text{Execute}) \xrightarrow{\mu hb} (l.\text{Execute})$.

SB: only one store can be outstanding from the store buffer at a time: for all stores s, for the immediately subsequent store s', $(s.\text{Completed}) \xrightarrow{\mu hb} (s'.\text{StoreBuffer})$.

TABLE III: PipeCheck definition of the gem5 O3 Pipeline [13]. "NLE" means non-local edges.

$(inst_a.Execute) \xrightarrow{\mu hb} (inst_b.Fetch)$. The WeeFence [18] example of Section VI-C will discuss such a case in detail.

Many modern pipelines are also superscalar. This can be modeled in PipeCheck by simply treating each location with multiple lanes as multiple locations. Alternatively, one could assign an arbitrary priority between lanes sharing a location to resolve ties, or one could even imagine μhb edges which indicate "happens before or at the same time as". For space reasons, we leave a more detailed description for future work.

Finally, some pipelines use microarchitectural optimizations which are considered legal, but which nevertheless violate the "must happen before" requirements of ppo. To legalize such optimizations, in these cases PipeCheck necessarily permits a ppo proposition to be satisfied by a condition other than simply "microarchitecturally happens before". Section VI will use PipeCheck to analyze two examples: one which relaxes Load→Load ordering (speculative load reordering [20]), and one which relaxes Store→mfence→Load orderings (WeeFence [18]).

V. Tool Flow and Methodology

The PipeCheck tool flow is depicted in Figure 5. PipeCheck is written using Coq [46], an interactive theorem prover, to make the code easily amenable to formal analysis and integration with existing open-source analysis frameworks also using Coq [3]. To speed up the analysis, we use built-in functionality within Coq to extract the computation into OCaml and then compile this extracted code into a standalone binary.

To demonstrate the effectiveness of PipeCheck, we focus on verification of processors implementing the TSO consistency model. Weak memory models, such as those used by ARM and Power processors, impose ordering requirements only due to dependencies or at specialized synchronization points such as fences, and due to space constraints we do not aim to survey the wide variety of fence types used by different architectures. TSO imposes non-trivial ordering requirements on all memory operations, making verification of TSO a particularly interesting target. Furthermore, its widespread use on x86 and other platforms make its verification very important.

Table II summarizes the four microarchitectures we survey in our results. The first two are the five-stage RISC pipeline of Figure 2 both without and with a store buffer. The former is effectively a sequentially-consistent core, meaning that some of the litmus test outcomes permitted under TSO should not be observable. These two microarchitectures reflect the size of pipelines that might be used in classrooms or as small embedded cores. The third is the gem5 O3 simulated pipeline (v10013) [13]. This represents an average-sized core and demonstrates how simulated cores are also amenable to analysis. Finally, we describe the OpenSPARC T2 pipeline, representing a well-documented industry microarchitecture [44].

We analyze a comprehensive set of TSO litmus tests from previous work [38]. Each litmus test was analyzed on a four core version of each pipeline, as none of our tests required more than four cores. We also analyze the set of ppo and po-loc satisfaction tests (Section IV-D) for each pipeline.

We execute the extracted OCaml code and collect timing results using an Intel Xeon E3-1230 v2 server processor.

VI. Case Studies

Having defined the PipeCheck methodology, we now demonstrate its use by highlighting cases of particular interest.

A. Speculative load reordering

Many microarchitectures speculatively reorder load instructions for performance reasons [9, 20, 26, 43]. The key principle is that two loads l_1 and l_2 in program order can be speculatively reordered (i.e., l_2 can execute before l_1) as long as the value read speculatively by l_2 is the same as it would have been had l_2 in fact performed after l_1 (i.e., non-speculatively). One implementation, as used by the gem5 O3 pipeline [13] that we analyze in this paper, is to hook into the cache coherence protocol. Namely, if a private cache line has not been overwritten or invalidated (due to cache replacement or an external invalidate request) since an earlier read of that line, then the core can safely assert that a subsequent read of that line would return the same value. On the other hand, if the cache line is invalidated, the core is conservative and assumes that the invalidate indicates a failed speculation.

This implementation of speculative load reordering can be modeled in PipeCheck by including cache line invalidation as a "location" within the model. Figure 6a shows an example of PipeCheck's use of this as applied to the gem5 O3 pipeline model and to the depicted litmus test. Extra vertices have been added to represent the invalidations of the cache lines that (i3) and (i4) read from, and the observed edges in the graph have been adjusted to account for these new vertices. In particular, the cache line that (i4) reads from must have been invalidated before (i1) wrote to memory to observe the proposed result.

PipeCheck uses this μhb graph to analyze the correctness of speculative load reordering. This is an out-of-order pipeline, so there is no sequence of edges guaranteeing $\texttt{ppo}_{arch}$ as there was in Figure 1c. However, although a μhb version of this edge would be sufficient, speculative load reordering proposes that it is not strictly necessary. Should the processor guarantee that the $\texttt{ppo}_{slr}$ edge is enforced instead, this would be sufficient to prevent the forbidden outcome from being observed regardless of the presence of the $\texttt{ppo}_{arch}$ edge. As a result, the core can safely realize the performance benefits of reordering (i3) and (i4) as long as it enforces either $\texttt{ppo}_{arch}$ or $\texttt{ppo}_{slr}$.

B. Consistency Bug in gem5 O3 Pipeline

For the gem5 O3 pipeline as defined in Table III, our PipeCheck results indicated that Load→Load `ppo` ordering was not guaranteed, and that four of the litmus tests (including `iwp2.1/amd1/mp`, shown in Figure 6b) failed validation. These results could mean either that we omitted a critical set of non-local edges in the PipeCheck definition of the pipeline, or that PipeCheck had in fact found a bug in the implementation. To analyze further, we wrote a microbenchmark to execute `iwp2.1/amd1/mp` in a tight loop. With this, the software was in fact able to observe the forbidden result, clearly indicating that PipeCheck had found a bug.

While difficult to find without PipeCheck, the Load→Load ordering bug is easily correctable in this case[2]. The pipeline already does correctly implement Load→Load ordering in some cases: it squashes and restarts the second of the two reordered loads if the core sees an invalidate (as described in Section VI-A) to the line read by the second load, but *only if the accesses are to the same address*. Simply removing the second condition is sufficient to restore correctness. Fortunately, as actual ordering violations are relatively rare, we believe this fix to result in only minimal performance changes in practice. This case study demonstrates the ability of PipeCheck to automatically find and identify very specific pipeline bugs and/or missing guarantees in the specification of the pipeline.

C. WeeFence

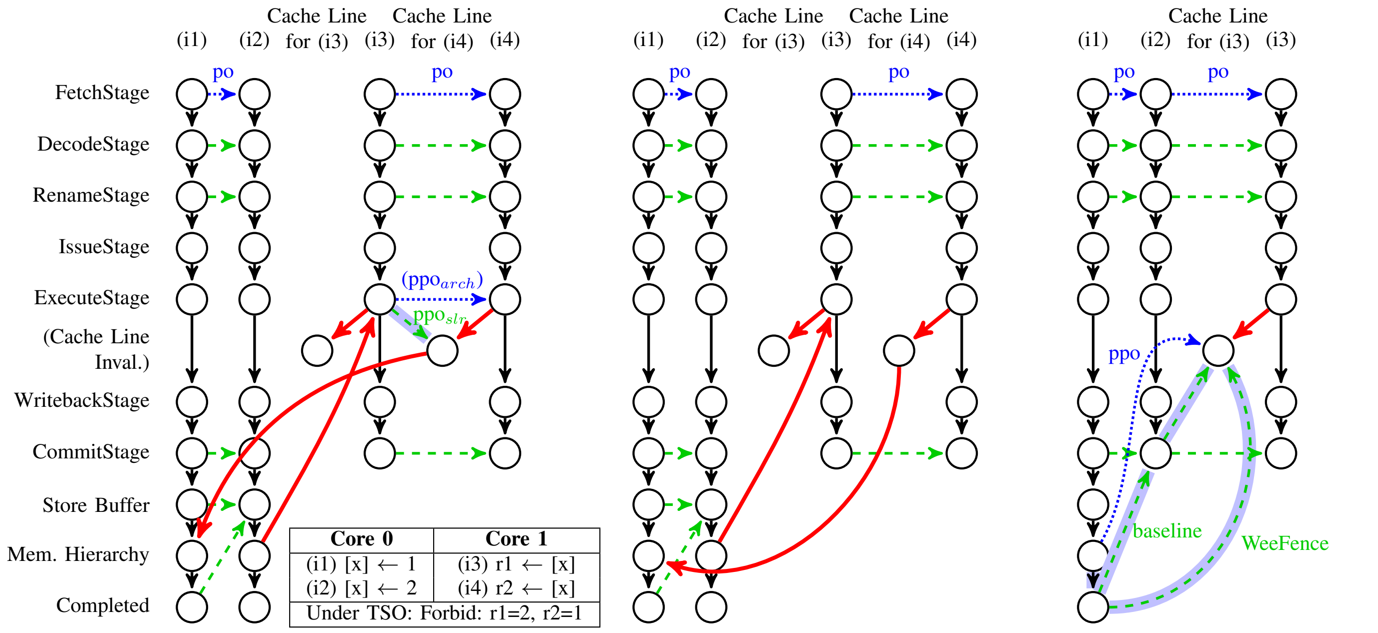
Our third case study uses PipeCheck to verify the correctness of WeeFence [18]. WeeFence proposes a microarchitectural optimization to make enforcement of Store→`mfence`→Load orderings cheap under TSO. Specifically, they propose allowing post-fence loads to perform and retire prior to the fence itself, thereby reducing latency. We check that these continue to correctly enforce TSO in a particular implementation.

Figure 6c demonstrates the use of PipeCheck to validate the correctness of the WeeFence approach. Since their technique is not specific to a particular implementation, we apply it to the gem5 O3 pipeline model, as it allows out-of-order execution. In their baseline microarchitecture, the load may speculatively perform before the fence has retired, similarly to Section VI-A, but it may not retire until after the fence has retired. This in turn must happen after the store has written back to memory. In other words, ordering is enforced from (i1.MemHierarchy) to (i1.Completed) to (i2.CommitStage) to (i3.CacheLineInvalidate), where the last segment is enforced by squashing (i3) if necessary. They then propose the optimization of buffering or bouncing invalidates rather than monitoring for them, which in turn allows the read to safely retire non-speculatively, even before the store has written back to memory. This approach also enforces (i1.MemHierarchy)$\xrightarrow{\mu hb}$(i3.CacheLineInvalidate), but without the slow intermediate step of (i2.CommitStage), thereby saving latency. This analysis demonstrates how PipeCheck can be used to verify and then to demonstrate the correctness of a microarchitectural optimization proposal.

VII. Results Across Litmus Tests

Table IV shows the results of verifying the suite of litmus tests on each modeled pipeline. Individual litmus tests are depicted as rows. For each row, the table shows whether TSO forbids or permits the outcome proposed by the test, and then shows its observability on the four microarchitectures considered. The microarchitecturally-observable behaviors correspond with the architecturally-specified behaviors in almost all

[2] The bug was independently fixed in revision 10149



(a) Speculative load reordering: although ppo$_{arch}$ is not enforced, a legal replacement ppo$_{slr}$ is enforced, and it completes the cycle.

(b) Pipeline bug shown via the iwp2.1/amd1/mp litmus test. The lack of a cycle indicates that the behavior is (erroneously) observable.

(c) WeeFence [18] eliminates the slow "baseline" dependency while maintaining the necessary ordering.

Fig. 6: Case Studies of Section VI, all demonstrated using the gem5 O3 pipeline.

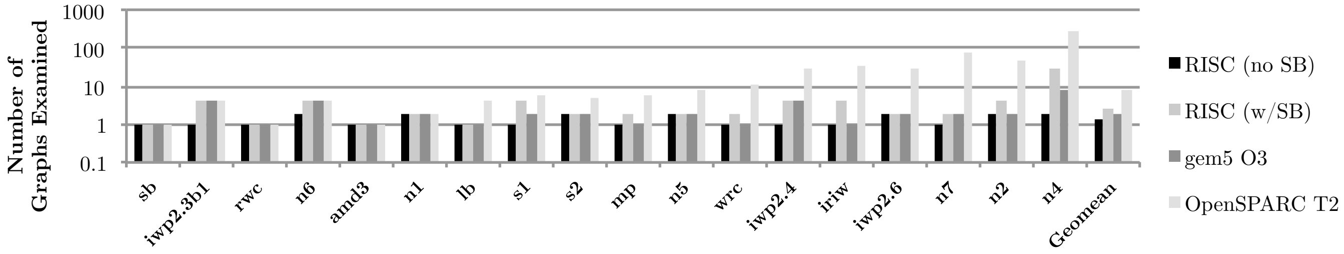


Fig. 7: Number of graphs needed to verify each litmus test.

cases. For the RISC pipeline without a store buffer, six of the proposed results require non-SC behavior, and these results are confirmed as not being observable on the SC pipeline. On the other hand, four of the test results reinforce the observability of the gem5 pipeline bug discussed in Section VI-B.

Figure 7 shows the number of cases that needed to be enumerated to verify each litmus test. Broadly, the "permitted" cases require fewer graphs, as the enumeration can stop once a single match is found. There are exceptions, however. The n7 litmus test (described in Section III) checked 73 cases on the OpenSPARC T2 before finding an ordering causing the proposed result. Nonetheless, performance remains acceptable. Should performance become an issue, heuristics and/or parallelism could be employed to mitigate the cost.

At maximum, for test n4 on the OpenSPARC pipeline, PipeCheck needed to check 288 cases ($288 = 2^5 \cdot 3^2$). First, n4 has 4 reads each taking one of either two or three possible paths (hit, miss, store buffer forwarding if applicable) ($2^2 \cdot 3^2$). Second, two of the reads follow writes to the same address from the same core, and hence they can return values either from the store buffer or from memory (2^2). Finally, there are two possible ws orderings (2^1).

Figure 8 shows the time taken to complete the verification process for each pipeline. The total runtime is closely correlated with the number of graphs per litmus test shown in Figure 7. This runtime could be reduced even further by parallelizing Algorithm 4, which takes 90% of the non-I/O runtime of the current implementation. Nevertheless, the entire suite runs in less than ten minutes for each pipeline, demonstrating that even with code optimized for verifiability rather than performance, the PipeCheck analysis is very practical.

VIII. Related Work

Lamport first defined sequential consistency to be the conditions that the result of an execution is the same as some interleaving of the instructions from each thread, and that the instructions from each thread remain in program

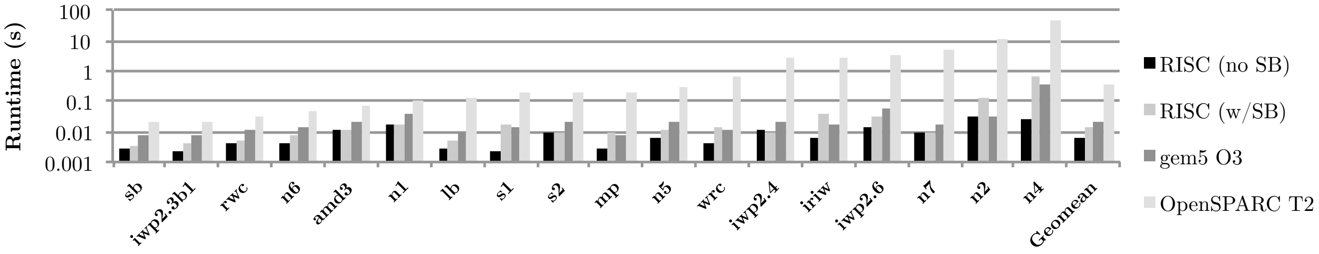


Fig. 8: Verification Time Results (computed using extracted OCaml).

Litmus Test	TSO (exp.)	RISC (no SB)	RISC (w/SB)	gem5 O3	Open-SPARC
iwp2.1/amd1/mp	F	=	=	**O**[2]	=
iwp2.2/amd2/lb	F	=	=	=	=
iwp2.3a/amd4/sb	P	**N**[1]	=	=	=
iwp2.3b	P	=	=	=	=
iwp2.4/amd9	P	**N**[1]	=	=	=
iwp2.5/amd8/wrc	F	=	=	**O**[2]	=
iwp2.6	F	=	=	=	=
amd3	P	**N**[1]	=	=	=
amd6/iriw	F	=	=	**O**[2]	=
n1	P	**N**[1]	=	=	=
n2	F	=	=	**O**[2]	=
n4	F	=	=	=	=
n5	F	=	=	=	=
n6	P	=	=	=	=
n7	P	**N**[1]	=	=	=
rwc	P	**N**[1]	=	=	=

[1] Implementation more restrictive than TSO requires.
[2] Indicates the presence of a bug. See Section VI-B.

TABLE IV: Summary of litmus test results. "F": Forbid. "P": Permit. "=": Matches expected TSO outcome. "O": Observable. "N": Not observable.

order [32]. Various hardware memory models have since been developed, including weak memory ordering [19], processor consistency [23] and release consistency [21], as well as industrial models such as those used by Alpha [17], SPARC [43], and PowerPC [16].

Shasha and Snir [41] and Collier [15] provided early frameworks to analyze programs running on machines with memory models weaker than sequential consistency. Many other studies (e.g.,) have since analyzed the process of restoring sequential consistency given a weaker hardware model, but these analyses typically remain at the architecture level and above [6, 30]. Formal architecture-level models [11], including those described in Section III, also apply similar graph-theoretic techniques. Modern research into formal models can broadly be classified into two groups: axiomatic models (from industry [17, 27, 43] or academia [1, 3, 4, 7, 33, 48]) or operational models [39], although researchers have been able to demonstrate the equivalence of models of each type (e.g., for TSO [37], for Power [33], for generic models [8]).

To match the hardware models, academic studies into the implications of non-sequentially consistent programming models led to the development of guarantees such as data-race freedom (DRF) [2], which specified conditions under which code executing under a weak memory model will nevertheless behave in a sequentially consistent manner. Modern specifications for languages such as C11/C++11 [14, 28, 29] and Java [34] define graphs with edges such as "synchronizes with" or "sequenced before" that are in many ways analogous to "happens before" edges in hardware models. With this, researchers have also attempted to formalize the mappings of software constructs such as mutexes or acquire/release atomics into the set of primitives exposed by the architecture [6, 12].

Formal methods approaches have also been used for verifying software correctness. Gibbons and Korach demonstrated the NP-completeness of verifying sequential consistency [22]. Nevertheless, in practice many studies have been successful through the use of model checking [5] and/or SAT solving [47] and/or through formalization using proof assistants such as Coq [46] or HOL [42].

Much of the effort in attempting to verify microarchitectural *implementations* of consistency models has focused on creating and evaluating litmus tests [7, 8, 24, 39, 45], including those described explicitly in vendor specifications [9, 10, 17, 25, 26, 43]. We are not aware of any previous attempts to create microarchitecture-level "happens before" graphs, nor of attempts to automate formal microarchitecture-level checking.

IX. Conclusion

We presented PipeCheck, a methodology and tool for verifying the correctness of a microarchitecture with respect to its consistency model. PipeCheck demonstrates the practicality and tractability of defining microarchitectures in terms of their location-by-location ordering properties and then verifying the correctness of their implementation of the given consistency model. Our techniques complement other ongoing efforts to verify the correctness of computation, from the programming language level down to the microarchitecture. We hope that in the future, PipeCheck will serve both as a framework in which designers can define their microarchitectures and as a tool by which they can verify the correctness of their implementations. PipeCheck is open-source and is publicly available at github.com/daniellustig/pipecheck.

Acknowledgments

The authors would like to thank Jade Alglave, Lennart Beringer, Doug Clark, Nirav Dave, and the anonymous reviewers for their helpful feedback. Daniel Lustig was supported in part by an Intel PhD Fellowship. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was also supported in part by NSF under the grant CCF-1117147.

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. V. Adve and M. D. Hill, "Weak ordering—a new definition," *17th International Symposium on Computer Architecture (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, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of Power and ARM machine code," *Declarative Aspects of Multicore Programming (DAMP) Workshop, in conjunction with POPL 2009*, 2009.

[5] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," *25th International Conference on Computer Aided Verification (CAV)*, 2013.

[6] J. Alglave and L. Maranget, "Stability in weak memory models," *23rd International Conference on Computer Aided Verification (CAV)*, 2011.

[7] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[8] 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, no. 2, pp. 7:1–7:74, Jul. 2014.

[9] AMD, "AMD64 architecture programmer's manual, rev. 3.24," Oct. 2013. Available: http://developer.amd.com/resources/documentation-articles/developer-guides-manuals

[10] ARM, "Barrier litmus tests and cookbook," *Document Number PRD03-GENC-007826*, 2009.

[11] Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," *33rd International Symposium on Computer Architecture (ISCA)*, 2006.

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

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

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

[15] W. W. Collier, *Reasoning about Parallel Architectures*. Prentice Hall, Inc., 1992.

[16] 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.

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

[18] Y. Duan, A. Muzahid, and J. Torrellas, "WeeFence: Toward making fences free in TSO," *40th International Symposium on Computer Architecture (ISCA)*, 2013.

[19] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," *13th International Symposium on Computer Architecture (ISCA)*, 1986.

[20] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," *29th 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," *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[22] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 1208–1244, 1997.

[23] J. R. Goodman, "Cache consistency and sequential consistency," *U. of Wisconsin-Madison, CS Dept. Tech. Report 1006*, 1991.

[24] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," *31st International Symposium on Computer Architecture (ISCA)*, 2004.

[25] IBM, "Power ISA, 2.07," 2013. Available: http://www.power.org/resources/reading

[26] Intel, "Intel 64 and IA-32 architectures software developer's manual," *Order Number 325462-048US*, Sept. 2013. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[27] Intel, "Intel Itanium architecture software developer's manual, rev. 2.3," 2013.

[28] ISO/IEC, "Information technology – Programming languages – C," 2011, ISO/IEC 9899.

[29] ISO/IEC, "Information technology – Programming languages – C++," 2011, ISO/IEC 14882.

[30] M. Kuperstein, M. Vechev, and E. Yahav, "Automatic inference of memory fences," *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2010.

[31] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[32] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computing*, vol. C-28, Sept. 1979.

[33] 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," *24th International Conference on Computer Aided Verification (CAV)*, 2012.

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

[35] F. Z. Nardelli, P. Sewell, J. Sevcik, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave, "Relaxed memory models must be rigorous," *2nd International Workshop on Exploiting Concurrency Efficiently and Correctly (EC)^2, in conjunction with CAV 2009*, 2009.

[36] Oracle, "Oracle SPARC architecture 2011, draft D0.9.5d," Jul. 2012. Available: http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/index.html

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

[38] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO (extended version)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-745, Mar. 2009.

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

[40] C. Scheurich and M. Dubois, "Correct memory operations of cache-based multiprocessors," *14th International Symposium on Computer Architecture (ISCA)*, 1987.

[41] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 2, pp. 282–312, 1988.

[42] K. Slind and M. Norrish, "A brief overview of HOL4," *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

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

[44] Sun, "OpenSPARC T2 core microarchitecture specification, rev. A," Dec. 2007.

[45] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey, "Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor-the DEC Alpha 21264 microprocessor," *35th Design Automation Conference (DAC)*, 1998.

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

[47] E. Torlak, M. Vaziri, and J. Dolby, "MemSAT: Checking axiomatic specifications of memory models," *31st Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[48] 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*, 2003, vol. 2860.