2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications

Yatin A. Manerkar Daniel Lustig* Margaret Martonosi Aarti Gupta
Princeton University NVIDIA*
{manerkar,mrm,aartig}@princeton.edu dlustig@nvidia.com

***Abstract*—Memory consistency models (MCMs) specify rules which constrain the values that can be returned by load instructions in parallel programs. To ensure that parallel programs run correctly, verification of hardware MCM implementations would ideally be *complete*; i.e. verified as being correct across *all possible executions* of *all possible programs*. However, no existing automated approach is capable of such complete verification.**

To help fill this verification gap, we present PipeProof, a methodology and tool for complete MCM verification of an axiomatic microarchitectural (hardware-level) ordering specification against an axiomatic ISA-level MCM specification. PipeProof can *automatically* prove a microarchitecture correct in all cases, or return an indication (often a counterexample) that the microarchitecture could not be verified. To accomplish unbounded verification, PipeProof introduces the novel *Transitive Chain Abstraction* to represent microarchitectural executions of an arbitrary number of instructions using only a small, finite number of instructions. With the help of this abstraction, PipeProof proves microarchitectural correctness using an automatic abstraction refinement approach. PipeProof's implementation also includes algorithmic optimizations which improve runtime by greatly reducing the number of cases considered. As a proof-of-concept study, we present results for modelling and proving correct simple microarchitectures implementing the SC and TSO MCMs. PipeProof verifies both case studies in under an hour, showing that it is indeed possible to automate microarchitectural MCM correctness proofs.

***Index Terms*—Memory consistency models, automated verification, formal verification, happens-before graphs, abstraction refinement.**

I. Introduction

Memory consistency models (MCMs) specify rules which constrain the values that can be returned by load instructions in parallel programs [1]. MCMs are defined at various levels of the hardware-software stack, including programming languages [15, 16, 25] and instruction set architectures (ISAs) [3, 14]. An ISA-level MCM like x86-TSO [14] serves as both a target for x86 compilers and a specification of what x86 hardware must implement. A key challenge in parallel hardware design is ensuring that a given microarchitecture (hardware design) obeys its ISA-level MCM. If the hardware does not respect its MCM in all possible cases, then the correct operation of parallel programs on that implementation cannot be guaranteed.

MCM bugs in hardware are becoming more common as parallel processing becomes ubiquitous and implementations increase in complexity. For example, two transactional memory bugs were discovered in recent Intel processors [11, 32]. One of these bugs was fixed by disabling the transactional memory functionality entirely, reducing processor capabilities. In addition, MCM bugs have been discovered in research simulators [18], coherence protocols [9], and open-source processors [23]. Furthermore, incorrect MCM implementations can be leveraged to create security exploits [10].

The ramifications of MCM bugs in parallel processors necessitate verifying that the hardware correctly implements its MCM. Such verification would ideally be *complete*; i.e. it would cover all possible executions of all possible programs. However, truly complete verification is extremely difficult. The only existing complete consistency proofs of hardware are implemented in interactive theorem provers [7, 31], which require significant manual effort.

Automated approaches could make it much easier for microarchitects to verify their designs. However, no current automated approach is capable of complete MCM verification. Dynamic verification approaches [12, 26] only examine a subset of the possible executions of any program they test, so they are incomplete even for those programs. Formal MCM verification approaches look at all possible executions of the programs they check, but all such approaches that are automated have only been able to conduct verification of implementations for a *bounded* set of programs. The verified programs in such an approach may be a suite of litmus tests[1] which focus on the scenarios most likely to exhibit bugs [2, 18, 19, 23, 24, 30], or all programs smaller than a certain bound (∼10 instructions) [33].

Critically, litmus test-based or bounded verification only ensures that the implementation will correctly enforce orderings for the verified programs themselves. It does *not* ensure that the implementation will correctly enforce required orderings for all possible programs. These incomplete approaches have missed important bugs because the set of programs they verified did not exercise those bugs [33]. This necessitates an automated approach capable of conducting *complete* MCM verification of microarchitectural implementations.

To fill this need for complete microarchitectural MCM verification, we present PipeProof[2], a methodology and automated tool for unbounded verification of axiomatic microarchitectural ordering specifications [19] against axiomatic ISA-level MCM

[1]Litmus tests are small programs used to test MCM implementations.
[2]open-source and publicly available at github.com/ymanerka/pipeproof.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00069

specifications [2]. PipeProof uses an approach based on Satisfiability Modulo Theories (SMT) [4] and Counterexample-Guided Abstraction Refinement (CEGAR) [8]. PipeProof's unbounded verification covers all possible programs, core counts, addresses, and values. The key to PipeProof's unbounded verification is its novel *Transitive Chain Abstraction*, which allows PipeProof to inductively model and verify the infinite set of program executions that must be verified for a given microarchitecture. As its output, PipeProof either provides a guarantee that the microarchitecture is correct, or returns an indication that the microarchitecture could not be verified.

The contributions of this paper are as follows:

- **Beyond Litmus Tests:** PipeProof is the first automated microarchitectural MCM verification tool that is capable of conducting complete verification of microarchitectural ordering specifications against axiomatic ISA-level MCM specifications.
- **Abstraction for Modelling an Infinite Set of Executions:** Verification of a microarchitectural design across all possible programs requires reasoning about an infinite number of possible executions. PipeProof's Transitive Chain Abstraction allows this infinite set of executions to be modelled inductively, thus making complete microarchitectural verification feasible.
- **Comprehensive Automated Verification of General Microarchitectures:** PipeProof's algorithms and its Transitive Chain Abstraction have broad applicability across microarchitectures and ISA-level patterns. They are not restricted to a particular microarchitecture or a particular ISA-level MCM.

The rest of this paper is organised as follows. Section II provides background on PipeProof's two main inputs (ISA-level MCM specification and microarchitectural ordering specification). Section III covers PipeProof's procedure for proving microarchitectural correctness. Section IV presents the supporting proofs and modelling techniques leveraged by the microarchitectural correctness proof. Section V covers optimizations that reduce PipeProof's runtime, while Section VI discusses our methodology, results, and ideas for future work. Section VII discusses related work, and Section VIII, our conclusions.

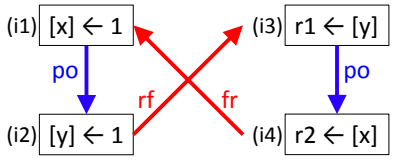
II. Background Information

A. ISA-Level MCMs and Their Axiomatic Specification

Figure 1a shows the message-passing (`mp`) litmus test, where core 0 communicates data (`x`) to core 1 by setting a flag (`y`) to indicate when the data is ready. In the outcome shown (`r1=1, r2=0`), core 1 observes the write to the flag but not the write to the data. The ISA-level MCM of the system dictates whether or not hardware should allow this behaviour. For example, sequential consistency (SC) [17] requires results that are consistent with a total order on all memory operations where (i) each load reads from the latest store to its address in the total order and (ii) each core performs its operations in program order. Thus, SC forbids the execution of `mp` where

Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
SC forbids r1=1, r2=0	

(a) Code for litmus test `mp`



(b) ISA-level execution of `mp` forbidden under SC, due to the cycle in the po, rf, and fr relations.

Fig. 1: Example litmus test `mp` and ISA-level execution of `mp`.

`r1=1, r2=0` (as Figure 1a shows), because there is no total order satisfying SC's requirements that results in this outcome.

ISA-level MCMs are often defined axiomatically using a model of relations between instructions in an execution, as described by Alglave et al. [2]. Each individual relation between a pair of instructions represents some ordering relevant to the MCM.

Definition 1 (ISA-Level Execution): An ISA-level execution $(Instrs, Rels)$ is a graph. Nodes $Instrs$ are instructions, and edges $Rels$ between nodes represent ISA-level MCM attributes.

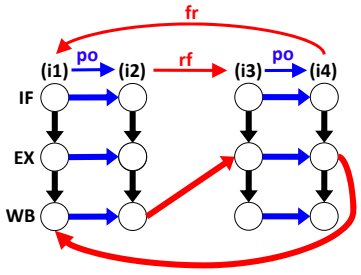
Figure 1b shows the ISA-level execution for the outcome of `mp` from Figure 1a. The po relation represents program order, so $i1 \xrightarrow{po} i2$ and $i3 \xrightarrow{po} i4$ represent that $i1$ is before $i2$ and $i3$ is before $i4$ in program order. The rf (reads-from) relation links each store to all loads which read from that store. For example, $i2 \xrightarrow{rf} i3$ represents that $i3$ reads its value from $i2$ in this execution. The fr (from-reads) edge between $i4$ and $i1$ enforces that the store that $i4$ reads from comes before the store $i1$ in coherence order (a total order on same-address writes in an execution). The co relation (not present in Figure 1b) is used to represent coherence order. Other ISA-level MCMs require extra relations to model reorderings and fences [2].

In the relational framework of Alglave et al. [2], the permitted behaviours of the ISA-level MCM are defined in terms of the irreflexivity, acyclicity, or emptiness of certain relational patterns. For example, SC can be defined using relational modelling as $acyclic(po \cup co \cup rf \cup fr)$. This means that any execution with a cycle in these four relations is forbidden. The execution in Figure 1b contains such a cycle, and so is forbidden under SC.

B. μspec Microarchitectural Specifications

PipeProof models microarchitectural executions using the microarchitectural happens-before (μhb) graph model developed by the Check suite [18, 19, 23, 24, 30].

Definition 2 (Microarchitectural Execution): A microarchitectural execution is a μhb graph $(Instrs, N, E)$. Nodes N represent individual sub-events in the execution of instructions $Instrs$. Edges E represent happens-before relationships between nodes.



(a) μhb graph for `mp`'s forbidden outcome on `simpleSC` microarch.

```
Axiom "IF_FIFO":
forall microops "a1", "a2",
(SameCore a1 a2 /\ ~SameMicroop a1 a2) =>
  EdgeExists((a1,Fetch)), (a2,Fetch))) =>
    AddEdge((a1,Execute)), (a2,Execute))).
```

(b) μspec axiom expressing that the Fetch pipeline stage should be FIFO on `simpleSC`.

Fig. 2: Example μhb graph for the `mp` litmus test and example μspec axiom.

Figure 2a shows an example μhb graph for the execution of the `mp` litmus test (Figure 1a) where the load of `y` ($i3$) returns 1 and the load of `x` ($i4$) returns 0 (i.e. `r1=1, r2=0`). Above the μhb graph is the corresponding ISA-level cycle in $po \cup co \cup rf \cup fr$ (also seen in Figure 1b), showing that SC requires this execution to be forbidden. The μhb graph represents an execution of `mp` on a simple microarchitecture (henceforth referred to as `simpleSC`) with 3-stage in-order pipelines. The 3 stages in this pipeline are `Fetch` (`IF`), `Execute` (`EX`), and `Writeback` (`WB`). Each column in the μhb graph represents an instruction flowing through the pipeline. Each node represents a particular instruction at a particular microarchitectural location. For instance, the leftmost node in the second row represents instruction $i1$ at its `Execute` stage, while the second node in the second row represents instruction $i2$ at its `Execute` stage. The edge between these two nodes enforces that they pass through the `Execute` stage in order, as required by the in-order pipeline. All μhb edges are transitive. Thus, a cyclic μhb graph implies that an event must happen before itself (which is impossible), so it is an execution that is *unobservable* on the target microarchitecture. Likewise, an acyclic μhb graph represents an execution that is *observable* on the target microarchitecture. The graph in Figure 2a is cyclic, so this microarchitectural execution is unobservable, as SC requires.

μhb graphs can be constructed according to microarchitectural axioms written in μspec, a domain-specific language similar to first-order logic [19]. A microarchitectural ordering specification is provided to PipeProof as input as a set of μspec axioms. These axioms specify which nodes (events) exist in a given execution and what edges (orderings) exist between those nodes. A set of μspec axioms thus constitutes a design description of microarchitectural orderings. Figure 2b shows an example μspec axiom which enforces that the `Fetch` pipeline stage is FIFO on `simpleSC`. The axiom applies to all pairs of instructions `a1` and `a2` that are on the same core (`SameCore a1 a2`) where `a1` and `a2` are distinct instructions (`~SameMicroop a1 a2`). For such pairs of instructions, if an edge exists between their `Fetch` stages (as denoted by the `EdgeExists` predicate), then an edge must also exist between their `Execute` stages (signified by the `AddEdge` predicate) to fulfill the requirements of the axiom. In the case of the μhb graph in Figure 2a, $i1$ and $i2$ constitute a pair of distinct instructions on the same core with an edge between their `Fetch` stages; thus, the axiom adds an edge between their `Execute` stages. Instructions $i3$ and $i4$ also satisfy the axiom's conditions on `a1` and `a2`, and so an edge is added between their `Execute` stages as well.

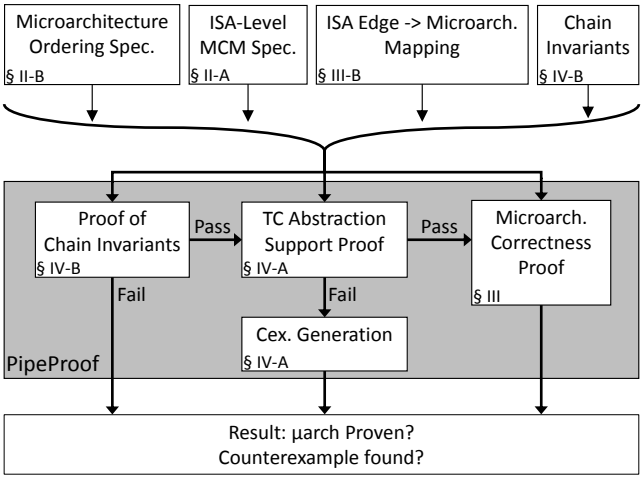


Fig. 3: High-level block diagram of PipeProof operation. Components are annotated with the sections in which they are explained.

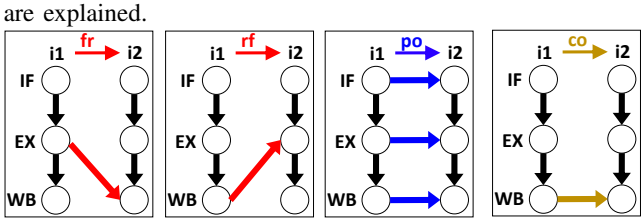


Fig. 4: ISA-level relations can be analysed in terms of how a given microarchitecture enforces them. These four μhb graphs show the μhb edges between instructions that are enforced (directly or indirectly) by the mappings of ISA-level edges to the `simpleSC` microarchitecture.

III. PipeProof Operation

Figure 3 shows PipeProof's high-level block diagram. The inputs to PipeProof are a set of μspec axioms describing microarchitectural orderings, an ISA-level MCM specification, mappings (to link ISA-level and microarchitectural execu-

```
Axiom "Mappings_po":
forall microop "i", forall microop "j",
HasDependency po i j => AddEdge ((i, Fetch), (j, Fetch), "po_arch").
```

Fig. 5: Example mapping axiom for *po* ISA-level edges on `simpleSC`.

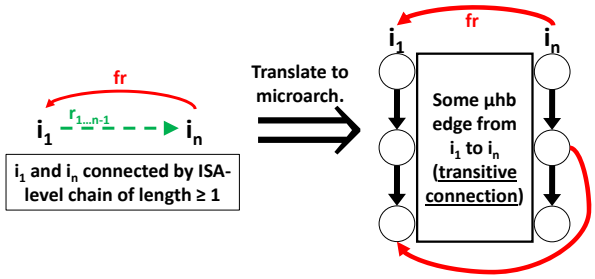


Fig. 6: A graphical example of the Transitive Chain (TC) Abstraction: all possible ISA-level chains connecting i_1 to i_n (left) can be abstracted as some μhb edge (the *transitive connection*) between the nodes of instructions i_1 and i_n (right). The red μhb edge is the microarchitectural mapping of the fr edge from i_n to i_1.

tions), and chain invariants (to abstractly represent repeating ISA-level patterns). As its output, PipeProof will either prove the microarchitecture correct for all possible programs or return an indication that the microarchitecture could not be verified.

PipeProof's overall operation has three high-level steps:

1) Prove chain invariants correct (Section IV-B).

Then for each forbidden ISA-level pattern in the ISA-level MCM specification:

2) Prove Transitive Chain (TC) Abstraction support for the microarchitecture and the ISA-level pattern (Section IV-A).
3) Prove Microarchitectural Correctness for the microarchitecture and the ISA-level pattern (this section).

The proofs of TC Abstraction support and chain invariants are supporting proofs on which PipeProof's main *Microarchitectural Correctness Proof* builds. The Microarchitectural Correctness Proof proves Theorem 1 below.

Theorem 1 (Microarchitectural Correctness): For each ISA-level execution $ISAExec := (Instrs, Rels)$ where $Rels$ exhibit a pattern forbidden by the ISA-level MCM, all microarchitectural executions $(Instrs, N, E)$ corresponding to $ISAExec$ are unobservable (i.e., their μhb graphs are cyclic).

This section describes the Microarchitectural Correctness Proof in detail, beginning with the structure of the ISA-level executions PipeProof verifies (Section III-A) and their translation to equivalent microarchitectural executions (Section III-B). The Microarchitectural Correctness proof uses an abstraction refinement approach that leverages the TC Abstraction (Section III-C) to model executions. PipeProof's refinement process involves examining abstract counterexamples (Section III-D) and refining the abstraction through concretization and decomposition (Section III-E). The section concludes by explaining when PipeProof's algorithm is able to terminate (Section III-F).

A. Symbolic ISA-Level Executions

PipeProof works with ISA-level executions that are similar to the ISA-level execution of `mp` in Figure 1b, but it uses *symbolic* instructions. In other words, the instructions in such ISA-level executions do not have specific addresses or values. The symbolic version of the ISA-level execution in Figure 1b would consist of four instructions $i1$, $i2$, $i3$, and $i4$, connected by the po, rf, and fr edges as shown, but nothing more would be known about the four instructions beyond the constraints enforced by the ISA-level relations. For instance, the instructions connected by po would be known to be on the same core, and the rf edge between $i2$ and $i3$ would enforce that $i2$ and $i3$ had the same address and value. However, the specific address and value of $i2$ and $i3$ could be anything. Such a symbolic ISA-level execution represents not only the ISA-level execution of `mp` in Figure 1b, but *any* ISA-level execution comprised of the cycle[3] $po; rf; po; fr$. Thus, verifying such a symbolic ISA-level execution checks the instance of the ISA-level pattern in that execution across all possible addresses and values, as required for complete verification.

B. Mapping ISA-level Executions to Microarchitecture

To verify that a forbidden ISA-level execution is microarchitecturally unobservable, one needs to translate the ISA-level execution to its corresponding microarchitectural executions[4]. PipeProof's complete verification requires such translation for any arbitrary ISA-level execution, not just a particular program instance. PipeProof's microarchitectural executions are similar to the μhb graph in Figure 2a, but like PipeProof's ISA-level executions, they operate on symbolic instructions which do not have specific addresses and values.

An ISA-level execution's instructions can be translated by instantiating the μspec microarchitectural axioms for those instructions. Translating ISA-level relations to microarchitecture is harder because the microarchitectural constraints implied by an ISA-level relation differ between microarchitectures. Thus, PipeProof requires user-provided *mappings* to translate individual ISA-level edges to their corresponding microarchitectural constraints. These mappings are additional μspec axioms that restrict the executions examined by PipeProof's solver to the microarchitectural executions where the mapped ISA-level edge exists between its source and destination instructions.

[3]A semicolon (;) denotes relational composition. For example, $r_1; r_2$ denotes a sequence of two ISA-level edges r_1 and r_2 where the destination instruction of r_1 is the source instruction of r_2.

[4]There are usually multiple microarchitectural executions corresponding to a single ISA-level execution.

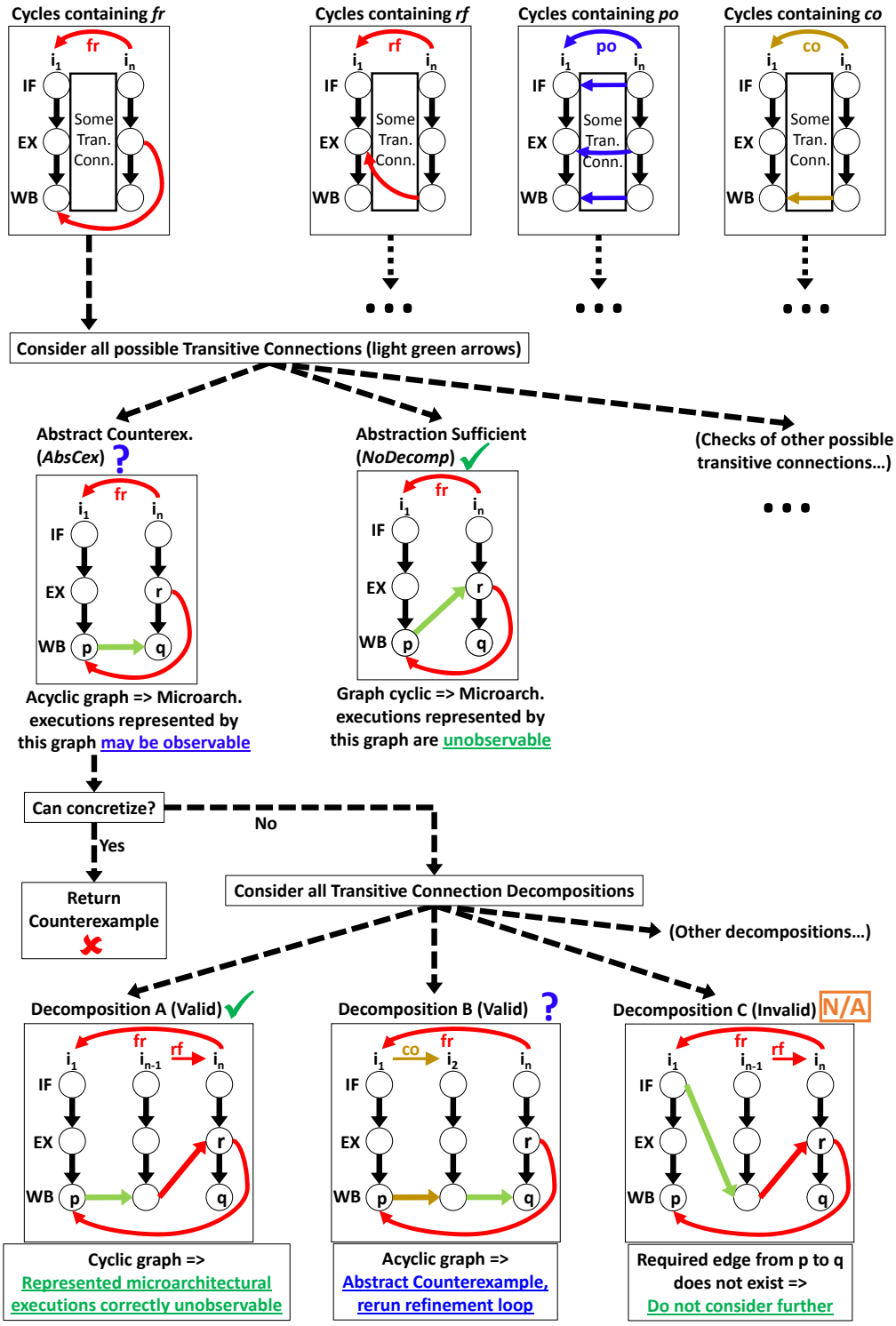


Fig. 7: PipeProof checking that non-unary ISA-level cycles forbidden by SC are unobservable on `simpleSC` with the help of the Transitive Chain Abstraction. This figure focuses on the cycles containing fr edges. Acyclic graphs like $AbsCex$ are abstract counterexamples: they may be concretized into real counterexamples or broken down into possible decompositions which are each checked in turn. Decompositions A and B are valid decompositions (subsets) of $AbsCex$, because they guarantee the edge from `p` to `q` in $AbsCex$. Decomposition C does not guarantee this edge, and is thus invalid and not considered further. Decomposition A strengthens the TC Abstraction enough to make the graph cyclic (and thus unobservable) as required. Decomposition B is valid but acyclic, so its abstraction needs to be refined further.

Figure 4 shows the μhb edges enforced by mappings of the ISA-level edges fr, rf, po, and co on `simpleSC`. Figure 5 shows the mapping axiom for po edges on `simpleSC`, which translates a po edge between instructions i and j to a μhb edge between the `Fetch` stages of those instructions. Such an edge can be seen between $i1$ and $i2$ in the po case of Figure 4. This edge between the `Fetch` stages indirectly induces edges between the `Execute` and `Writeback` stages of the instructions as well, through axioms like `IF_FIFO` from Figure 2b. These μhb edges are also shown in Figure 4, and reflect the in-order nature of `simpleSC`'s pipeline.

C. The TC Abstraction: Representing Infinite ISA-level Chains

Symbolic analysis and the use of mappings for ISA-level edges allow a single ISA-level cycle to be translated to the microarchitectural level for all possible addresses and values. However, there are an infinite number of possible ISA-level cycles that match a forbidden ISA-level pattern like $cyclic(po \cup co \cup rf \cup fr)$ (which are the executions forbidden by SC). Conceptually, all of these ISA-level cycles need to be verified as being unobservable in order to ensure that the microarchitecture is correct across all possible programs. Such unbounded verification is made possible by using inductive approaches. PipeProof achieves unbounded verification by inductively modelling executions using a novel *Transitive Chain (TC) Abstraction*. Specifically, PipeProof uses the TC Abstraction to efficiently represent ISA-level chains (defined below):

Definition 3: An ISA-level *chain* is an acyclic sequence of ISA-level relations $r_1; r_2; r_3 ...; r_n$. An example ISA-level chain is $po; rf; po$ from Figure 1b.

The TC Abstraction is the representation of an ISA-level chain of arbitrary length between two instructions i_1 and i_n as a single μhb edge between i_1 and i_n at the microarchitecture level. None of the intermediate instructions in the chain are explicitly modelled. *Abstract executions* are those executions where the TC Abstraction is used to represent an ISA-level chain. Meanwhile, *concrete executions* are those executions where all instructions and ISA-level edges are explicitly modelled; that is, where nothing is abstracted using the TC Abstraction. (Instructions in concrete executions are still symbolic.)

Figure 6 illustrates the TC Abstraction for ISA-level cycles containing the fr edge. The left of the figure represents all possible non-unary ISA-level cycles (i.e. the cycles containing more than one instruction[5]) that contain an fr edge. In these cycles, i_1 may be connected to i_n by a single ISA-level edge or by a chain of multiple ISA-level edges (the *transitive chain*). These cycles include the ISA-level cycle $po; rf; po; fr$ from Figure 1b (since it contains an fr edge), as well as an infinite number of other cycles containing fr. Using the TC Abstraction, any microarchitectural execution (as seen on the right) corresponding to such an ISA-level cycle represents the chain between i_1 and i_n by some μhb edge (the *transitive connection*) from a node of i_1 to a node of i_n. (The red μhb edge from i_n to i_1 is the mapping of the fr edge to the microarchitecture.) Thus, if the μhb graph on the right is verified to be unobservable for all possible transitive connections from i_1 to i_n, then the microarchitecture is guaranteed to be correct for all possible ISA-level cycles containing the fr edge. PipeProof automatically checks that the microarchitecture and ISA-level pattern support the TC Abstraction (Section IV-A) before using it to prove microarchitectural correctness.

The Transitive Chain Abstraction's capability to represent ISA-level chains of arbitrary length using a single transitive connection from the start to the end of the chain is the key insight underlying PipeProof's complete verification across all programs. This representation allows PipeProof to conduct unbounded verification while only explicitly modelling as many instructions as needed to prove microarchitectural correctness. The transitive connection models the effects of intermediate instructions in the ISA-level chain without explicitly modelling the instructions themselves, allowing for efficient modelling and verification of all possible ISA-level cycles. The TC Abstraction is both weak enough to apply to a variety of microarchitectures and also strong enough (with appropriate refinements discussed below) to prove microarchitectural MCM correctness.

D. Abstract Counterexamples

The TC Abstraction guarantees the presence of some μhb edge between the start and end of an ISA-level chain, such as that between i_1 and i_n in Figure 6. To prove microarchitectural correctness using the TC Abstraction, PipeProof must show that for each possible transitive connection between i_1 and i_n, all possible microarchitectural executions corresponding to the ISA-level cycles being checked are unobservable.

Figure 7 shows PipeProof's procedure for proving Theorem 1 on `simpleSC`. The figure focuses on the verification of ISA-level cycles containing at least one fr edge; the process is repeated for other types of cycles in the pattern (for SC, this equates to cycles containing po, co, or rf edges). For some transitive connections, like the one in the $NoDecomp$ case, the initial μhb graph of the abstract execution is cyclic. This proves the unobservability of all concrete executions represented by $NoDecomp$, as required for microarchitectural correctness.

In most cases, however, the initial abstract execution graphs will be acyclic, as is the case for $AbsCex$ in Figure 7. This is because the TC Abstraction's guarantee of a single μhb edge (the transitive connection) between the start and end of an ISA-level chain is necessarily rather weak in order to be general across *all* possible ISA-level chains that match a forbidden ISA-level pattern. The weakness of the guarantee is also necessary in order for the TC Abstraction to be general enough to support a variety of microarchitectures.

In abstraction refinement [8], cases such as $AbsCex$ that appear to violate the property being checked but contain an abstraction are called *abstract counterexamples*. They may

[5]The number of unary cycles (those where an instruction is related to itself) is small, so PipeProof explicitly enumerates and checks them separately.

correspond to concrete (real) counterexamples. They may also be *spurious*. When spurious, all concrete cases represented by the abstract counterexample are in fact correct.

In PipeProof, an abstract counterexample such as $AbsCex$ represents two types of concrete executions. First, i_1 and i_n may be connected by a single ISA-level edge. On the other hand, i_1 and i_n may be connected by a chain of multiple ISA-level edges. To check whether an abstract counterexample is spurious or not, PipeProof conducts *concretization* and *decomposition* (the *refinement loop*) to handle the aforementioned two cases.

E. Concretization and Decomposition: The Refinement Loop

In the concretization step, PipeProof checks the case where i_1 is connected to i_n by a single ISA-level edge. PipeProof does so by replacing the transitive connection between i_1 and i_n with a single ISA-level edge that causes the resultant ISA-level cycle to match the forbidden ISA-level pattern. This concrete execution must be microarchitecturally unobservable. For example, when trying to concretize $AbsCex$ in Figure 7, PipeProof checks the ISA-level cycle $po; fr$, then $co; fr$, then $rf; fr$, and finally $fr; fr$, since each of these are ISA-level cycles forbidden by SC that arise from replacing the transitive connection of $AbsCex$ with a single ISA-level edge. If any of these concrete executions is found to be observable, then the microarchitecture is buggy and PipeProof returns the observable ISA-level cycle as a counterexample.

If executions where the transitive connection is replaced by a single ISA-level edge are found to be unobservable, PipeProof then inductively checks the case where i_1 is connected to i_n by a chain of multiple ISA-level edges through decomposition. PipeProof decomposes the transitive chain of $n-1$ ISA-level edges[6] into a transitive chain of $n-2$ ISA-level edges represented by a transitive connection, and a single concrete ISA-level edge. This also results in the explicit modelling of an additional instruction. The concrete ISA-level edge and instruction added by the decomposition are "peeled off" the transitive chain. The key idea behind decomposition of the transitive chain is that the explicit modelling of an additional instruction and ISA-level edge enforces additional microarchitectural constraints that may be enough to create a cycle in the graph. If a cycle is created, the decomposition is unobservable, completing the correctness proof for that case. If all possible decompositions of an abstract counterexample are found to be cyclic (unobservable), then the abstract counterexample is spurious and can be ignored. If any decomposition is found to be acyclic, then that decomposition constitutes an abstract counterexample itself, and concretization and decomposition are repeated for it. Decomposing the chain one instruction at a time improves efficiency by ensuring that PipeProof does not explicitly model more instructions than needed to prove microarchitectural correctness.

Figure 7 shows three (of many) possible decompositions of $AbsCex$. In Decomposition A, an rf edge has been peeled off the right end of the transitive chain. Peeling off the rf edge *strengthens* the abstraction by connecting node p to node r (an edge not present in $AbsCex$). This creates a cycle in the μhb graph, rendering the execution unobservable. This completes the correctness proof for this decomposition.

Decomposition B in Figure 7 shows a case where a co edge (rather than an rf edge) is peeled off the transitive chain. Furthermore, the co edge is peeled off the left end of the transitive chain rather than the right. Peeling off the co edge refines the abstraction, but this is still not enough to create a cycle in the μhb graph. Thus, Decomposition B is itself an abstract counterexample, and the process of concretization and decomposition will be repeated for it.

To ensure completeness of verification when decomposing transitive chains, PipeProof enumerates all possibilities for the concrete ISA-level edge that could be peeled off (using the procedure from Section IV-E) and for the transitive connection representing the remaining chain of length $n-2$. Verifying a single decomposition is equivalent to verifying a subset of the executions of its parent abstract counterexample. As such, any valid decomposition must guarantee the transitive connections of its parent abstract counterexamples. Decompositions that violate this requirement do not represent executions that are modelled by their parent abstract counterexamples, and hence they are discarded.

For example, Decomposition C in Figure 7 is an invalid decomposition of $AbsCex$ because it does not guarantee the transitive connection of its parent $AbsCex$ (a μhb edge between nodes p and q) as Decompositions A and B do. PipeProof filters out any such decompositions that do not guarantee the transitive connections of their parent abstract counterexamples; it does not consider them further.

PipeProof alternates between peeling from the left and peeling from the right when inductively decomposing transitive chains. For example, Decomposition B was created by peeling from the left of $AbsCex$, so when concretization and decomposition is rerun for Decomposition B, the next ISA-level edge will be peeled from the right. PipeProof alternates in this manner because creating a cycle in the graph through decomposition often requires connecting more nodes to either side of the transitive connection.

F. Termination of the PipeProof Algorithm

In many cases, repeatedly peeling off instructions from the transitive chain strengthens the TC Abstraction enough to prove microarchitectural correctness. For the remaining cases, PipeProof requires user-provided *chain invariants* (Section IV-B) to abstractly represent infinite repeated peelings of a specific pattern of ISA-level edges and ensure termination of the refinement loop. For the SC and TSO case studies detailed in Section VI, peeling off a maximum of 9 instructions from the transitive chain was sufficient (along with chain invariants) to prove correctness of the microarchitectures.

IV. Supporting Proofs and Techniques

PipeProof's Microarchitectural Correctness proof relies on a number of supporting proofs and modelling techniques in or-

[6]Here, n is an abstract parameter used for induction. It has no concrete value.

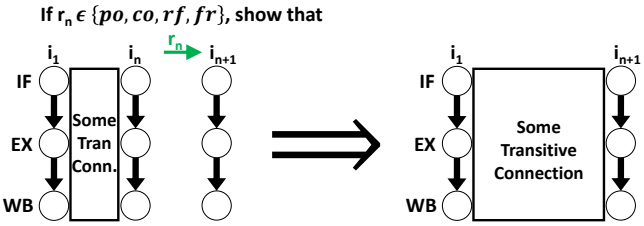


Fig. 8: Graphical depiction of the inductive case of the TC Abstraction support proof for `simpleSC`. Extending a transitive chain by an additional instruction and ISA-level edge r_n should extend the transitive connection to the new instruction as well.

der to prove correctness. This section explains these proofs and techniques, beginning with the TC Abstraction support proof (Section IV-A). This proof ensures that a microarchitecture and ISA-level pattern support the TC Abstraction, enabling it to be used in the Microarchitectural Correctness proof. Meanwhile, chain invariants (Section IV-B) are often required to ensure the termination of PipeProof's abstraction refinement loop. Theory Lemmas (Section IV-C) are required to constrain PipeProof's symbolic analysis to realisable microarchitectural executions. PipeProof must also use an over-approximation of microarchitectural constraints (Section IV-D) in order to guarantee soundness. Finally, Section IV-E describes how PipeProof inductively generates ISA-level edges matching a pattern when decomposing transitive chains.

A. Ensuring Microarchitectural TC Abstraction Support

As discussed in Section III-C, PipeProof uses the Transitive Chain (TC) Abstraction to represent the infinite set of ISA-level cycles that match a pattern like $cyclic(po \cup co \cup rf \cup fr)$. The TC Abstraction enables PipeProof to abstract away most of the instructions and ISA-level relations in these ISA-level cycles and represent them with a single μhb edge (the transitive connection) at the microarchitecture level.

The TC Abstraction is key to PipeProof's complete verification. However, to use the TC Abstraction in its Microarchitectural Correctness proof (Section III), PipeProof must first ensure that the microarchitecture and ISA-level pattern being verified support the TC Abstraction. If the TC Abstraction cannot be proven to hold for a given microarchitecture, then PipeProof cannot prove the microarchitecture correct. PipeProof's verification is sound; it will never falsely claim that the TC Abstraction holds without proving it.

The theorem for microarchitectural TC Abstraction support is provided below, following the definition of an ISA-level subchain:

Definition 4: An ISA-level chain $r'_1; r'_2; r'_3 ...; r'_k$ is a *subchain* of an ISA-level cycle or chain $r_1; r_2; r_3 ...; r_n$ if $k < n$ and $r_i = r'_i$ for $i = 1$ to k. In other words, the subchain is the first k edges of the chain. For example, in Figure 1b, $po; rf; po$ is a subchain of the cycle $po; rf; po; fr$.

Theorem 2: If instructions i_A and i_B are connected by a transitive chain (i.e. a subchain of a forbidden ISA-level cycle), then there exists at least one μhb edge (the transitive connection) from a node of i_A to a node of i_B in all microarchitectural executions in which that subchain is present.

PipeProof tries to automatically prove Theorem 2 inductively for each microarchitecture and ISA-level pattern for which the TC Abstraction is used.

Base Case: In the base case, we need to show that any single ISA-level edge $isaEdge$ that could be the start of the ISA-level chain to be abstracted guarantees a μhb edge between its source and destination instructions $i1$ and $i2$. For example, for `simpleSC`, PipeProof checks whether a po, co, rf, or fr edge between instructions $i1$ and $i2$ guarantees a μhb edge between them. As Figure 4 shows, the microarchitectural mappings of these ISA-level edges do indeed guarantee edges from $i1$ to $i2$ for `simpleSC`, so the base case passes.

Inductive Case: Figure 8 illustrates the inductive case for `simpleSC`. Given an ISA-level transitive chain between i_1 and i_n that implies a μhb transitive connection from i_1 to i_n, the inductive case must show that extending the transitive chain with an additional instruction i_{n+1} and ISA-level edge r_n matching the forbidden pattern extends the transitive connection. In other words, the inductive case must show that a μhb edge from some node of i_1 to some node of i_{n+1} exists.

If the combination of a transitive connection from i_1 to i_n and the microarchitectural mapping of r_n is not enough to guarantee a transitive connection from i_1 to i_{n+1}, this constitutes an abstract counterexample to Theorem 2. PipeProof then attempts to concretize and decompose the transitive chain between i_1 and i_n (as explained in Section III-E) to discover whether the abstract counterexample is spurious or whether a concrete ISA-level chain violating Theorem 2 exists. ISA-level chains that fail Theorem 2 are henceforth referred to as *failing fragments*.

As in the Microarchitectural Correctness proof (Section III), chain invariants (Section IV-B) are used to abstractly represent cases where an infinite number of edges could be peeled off without terminating. The abstraction refinement through decomposition continues until either the abstraction is strong enough to guarantee a transitive connection between i_1 and i_{n+1} in all cases (thus proving Theorem 2), or a failing fragment is found and returned to the user as failing the proof of Theorem 2.

Strength of Theorem 2: Theorem 2 is *stronger* than what the Microarchitectural Correctness proof (Section III) needs. Theorem 2 requires the transitive chain to guarantee a transitive connection both when the transitive chain is part of a forbidden ISA-level cycle in the overall execution (as the Microarchitectural Correctness proof requires) *and* when it is not part of such an ISA-level cycle (as seen in Figure 8). This enables Theorem 2 to be proven by induction. As Figure 8 shows, the inductive case consists of adding an extra instruction and ISA-level edge to the case guaranteed by the induction hypothesis, resulting in a proof by induction.

On the other hand, proving the existence of a transitive connection only in the presence of a forbidden ISA-level cycle is not as straightforward. In the inductive case of such a

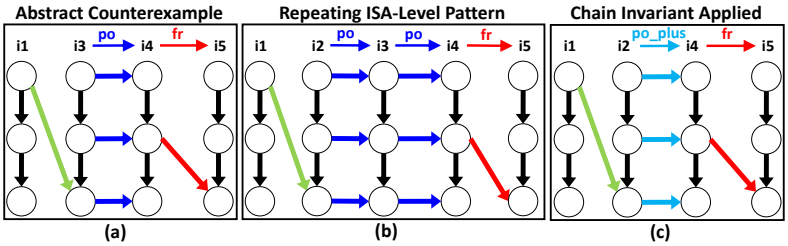


Fig. 9: Peeling off edges from abstract counterexamples like (a) may cause repetitions of the same pattern, like *po* in (b). Naively continuing to peel off repeated edges in this manner may prevent the refinement loop from terminating. *Chain invariants* efficiently represent an arbitrary number of repetitions of such ISA-level patterns, as shown by *po_plus* in (c), allowing PipeProof's refinement loop to terminate.

```
Axiom "Invariant_poplus":
forall microop "i", forall microop "j",
HasDependency po_plus i j => AddEdge((i,Fetch),(j,Fetch),"") /\ SameCore i j.
```

Fig. 10: Chain invariant for repeated *po* edges (i.e. *po_plus*) on the `simpleSC` microarchitecture.

proof, the induction hypothesis would guarantee a transitive connection for a chain between instructions i_1 and i_n only if the chain is part of a forbidden ISA-level cycle, similar to Figure 6. Extending this chain of length $n-1$ to a chain of length n (as required for an inductive proof) involves *removal* of one of the "loopback" edges connecting i_n to i_1 (*fr* in Figure 6). This is because a loopback edge connecting i_n to i_1 may not exist in arbitrary forbidden ISA-level cycles containing the extended transitive chain. If a loopback edge is removed, the induction hypothesis no longer guarantees a transitive connection between i_1 and i_n, and the proof cannot build on the guarantees for the chain of length $n-1$. In a nutshell, the induction hypothesis for such a proof is quite weak, so PipeProof cannot currently prove the necessary property, and attempts to prove the stronger Theorem 2 instead. This also means that some correct microarchitectures that do not satisfy Theorem 2 cannot be proven correct by PipeProof at present. We intend to address this issue in future work.

As a result of Theorem 2 being stronger than required, if a failing fragment is found, the microarchitecture may or may not be buggy. If the microarchitecture is buggy, PipeProof can generate an ISA-level cycle that exhibits the bug as a counterexample through its Cyclic Counterexample Generation procedure. This procedure checks all possible forbidden ISA-level cycles of length 1, then length 2, and so on for microarchitectural observability. At each iteration, if any of the cycles are microarchitecturally observable, the observable cycle is returned to the user as a counterexample. Otherwise, the procedure increases the size of the examined cycles by 1 and repeats the process.

B. The Need for Chain Invariants and their Proofs

When decomposing TC Abstractions instruction-by-instruction as outlined in Section III-E, it is possible to peel off concrete ISA-level edges that match a repeating pattern, but for the abstraction to *never* be strong enough to prove the required property (Theorem 1 or 2). For example, Figure 9a shows an abstract counterexample to Theorem 2 on `simpleSC` where there is no μhb connection between instructions $i1$ and $i5$. When decomposing this abstract counterexample, it is possible to peel off a *po* edge from the transitive chain, as shown in Figure 9b, and still have no μhb connection between $i1$ and $i5$. In fact, one can continue peeling off *po* edges in this manner *ad infinitum*, while never being able to guarantee a μhb edge between $i1$ and $i5$. Such a case will result in the refinement loop of the Microarchitectural Correctness proof or TC Abstraction support proof being unable to terminate.

For refinement loops to terminate in such cases, PipeProof needs a way to efficiently represent such repeating patterns. To do so, PipeProof utilises user-provided[7] *chain invariants*. These chain invariants are additional μspec axioms which specify microarchitectural guarantees about repeated ISA-level edge patterns. Users can examine PipeProof's status updates to detect when the peeling off of repeated edge patterns is preventing termination of a refinement loop. This is a sign that the user needs to provide PipeProof with a chain invariant for the repeated edge pattern in question.

Figure 10 shows an example chain invariant for `simpleSC` that abstracts a chain of successive *po* edges as a single *po_plus*[8] edge. This invariant states that if two instructions i and j are connected by a chain of *po* edges of arbitrary length, then at the μhb level, i and j are guaranteed to be on the same core and to have a edge between their `Fetch` stages (which in turn implies edges between their `Execute` and `Writeback` stages due to the in-order `simpleSC` pipeline). An ISA-level chain of successive *po* edges of arbitrary length on `simpleSC` can then be abstractly represented by a single *po_plus* edge (and the guarantees of its invariant), as seen in Figure 9c.

PipeProof automatically searches for concrete ISA-level patterns that can be abstracted by user-provided invariants in

[7]Future work could also use known invariant generation techniques to automatically discover invariants for a given concrete repeating ISA-level pattern. The search space of possible chain invariants is relatively small, and can be reduced further by restricting the search to specific invariant templates.

[8]The *plus* in *po_plus* is from Kleene plus.

each iteration of the refinement loop. The search for patterns matching available invariants is conducted edge by edge, similar to regex matching. Supported invariant patterns are repeated single edges (e.g., po) or repeated chains (e.g., $po; rf$). If PipeProof finds a concrete ISA-level pattern matching an invariant, it replaces the pattern with its invariant version. On subsequent decompositions, PipeProof's ISA Edge Generation procedure (Section IV-E) restricts the ISA-level edges that can be peeled off to those that cannot be subsumed within an adjacent chain invariant. For example, any edge peeled off from the right of the transitive chain in Figure 9c cannot be a po edge, as any such po edge is already subsumed within the po_plus edge between $i2$ and $i4$. This prevents edges matching an invariant pattern from being peeled off a transitive chain endlessly, allowing the refinement loop to terminate in such cases.

To help ensure verification soundness, PipeProof proves chain invariants inductively before using them in its proofs. If the proof of any chain invariant fails, PipeProof informs the user of the failure and does not proceed further. As an example of a chain invariant proof, consider Figure 10's invariant. PipeProof first checks the base case—whether a single po edge between two instructions i and j guarantees that they will be on the same core and have an edge between their `Fetch` stages. The po edge mapping and theory lemmas (Section IV-C) guarantee this. For the inductive case, PipeProof assumes that i and j are connected by a chain of a single po edge followed by a po_plus edge (i.e. a po-chain of arbitrary length), and that the invariant holds for the po_plus portion of the chain. It then checks if i and j are on the same core and have an edge between their `Fetch` stages. This property is guaranteed by the po edge mapping, theory lemmas, and the invariant from the induction hypothesis, completing the proof.

C. Theory Lemmas

The symbolic analysis conducted by PipeProof can allow inconsistent assignments to μspec predicates that are incompatible with *any* microarchitectural execution. For example, in any execution containing three instructions i, j, and k, if i and j have the same data (`SameData i j` is true), and j and k have the same data (`SameData j k` is true), then logically i and k must have the same data (`SameData i k` must be true). In other words, the `SameData` predicate is transitive. However, naive symbolic analysis will *not* require `SameData i k` to be true in such a case. Thus, to enforce such constraints, PipeProof provides a set of *Theory Lemmas*[9] for μspec predicates that is included in every call to PipeProof's solver. These constraints enforce universal rules (like the transitivity of `SameData`) that must be respected by *every* microarchitectural execution.

D. Over-Approximating to Ensure an Adequate Model

PipeProof verifies executions of an arbitary number of instructions while only modelling a small subset of those instructions and their constraints on the execution. Some of the instructions in an ISA-level cycle may be abstractly represented using the TC Abstraction or chain invariants, while other instructions in the execution that are not part of the ISA-level cycle are also not explicitly modelled. For its verification to be sound, PipeProof must ensure that the subset of an execution's constraints that it models is *adequate*: the subset must be an over-approximation of the constraints on the entire execution. In other words, it should never be the case that an execution is deemed to be unobservable when modelling only the subset of its constraints, but is actually observable.

[9]These lemmas are very similar to the lemmas produced by a theory solver in an SMT setup.

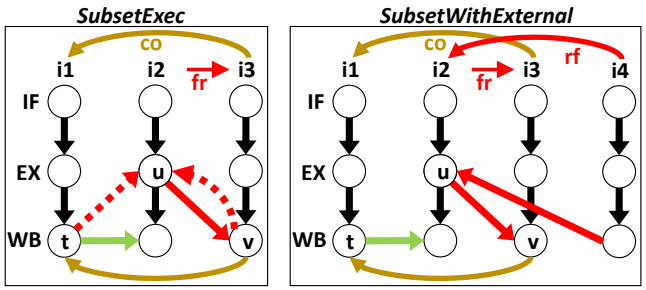


Fig. 11: The load $i2$ in $SubsetExec$ cannot read its value from the explicitly modelled stores $i1$ or $i3$ without adding one of the dotted edges and making the graph cyclic. This appears to make the execution unobservable. However, as shown in $SubsetWithExternal$, another instruction $i4$ outside the ISA-level cycle can source $i2$ while keeping the graph acyclic, making $SubsetWithExternal$ an abstract counterexample. PipeProof must over-approximate microarchitectural constraints to account for instructions like $i4$ that are not explicitly modelled.

For example, consider the abstract execution $SubsetExec$ on `simpleSC` in Figure 11, where an ISA-level cycle is abstractly represented with the help of the TC Abstraction. Consider also the constraint (henceforth called `LoadSource`) on every ISA-level execution that for every load l which does not read the initial value of memory, there exists a store s for which $s \xrightarrow{rf} l$, corresponding to s being the store from which l reads its value. Instruction $i2$ in $SubsetExec$ is a load (since it is the source of an fr edge), and so must satisfy the `LoadSource` constraint. If PipeProof attempted to satisfy `LoadSource` for $i2$ using only the explicitly modelled instructions, $i2$ could be sourced either from $i1$ (i.e. $i1 \xrightarrow{rf} i2$) or from $i3$ ($i3 \xrightarrow{rf} i2$). If sourcing from $i1$, the microarchitectural mapping of the rf edge adds a μhb edge from node t to node u, giving us a cycle in the μhb graph. Likewise, if $i3$ is used as the source, $i3 \xrightarrow{rf} i2$ maps to a μhb edge from node v to node u, once again creating a cycle in the graph. Thus, if the analysis only considered explicitly modelled instructions, it would deduce that all graphs for this case are cyclic (i.e. unobservable), and that this case need not be concretized and decomposed.

However, this reasoning would be incorrect. For instance, it is perfectly valid for an execution containing the ISA-

level cycle from $SubsetExec$ to have an additional instruction $i4$ that is not part of the ISA-level cycle but sources the value of $i2$ (i.e. $i4 \xrightarrow{rf} i2$). Figure 11 depicts this variant as $SubsetWithExternal$, which satisfies `LoadSource` while maintaining an acyclic graph. This indicates (correctly) that the ISA-level cycle from $SubsetExec$ is actually an abstract counterexample and must be concretized and decomposed.

To avoid unsoundly flagging executions such as $SubsetExec$ as unobservable, PipeProof conservatively over-approximates by replacing every `exists` clause in the μspec with a Boolean `true`. This suffices to guarantee an adequate model, since `exists` clauses are the only μspec clauses whose evaluation can change from `false` to `true` when an additional instruction is explicitly modelled. In the case of $SubsetExec$, the over-approximation results in `LoadSource` always evaluating to true, ensuring that $SubsetExec$ is treated as an abstract counterexample as required for soundness.

This over-approximation forces PipeProof to work only with a subset of the overall true microarchitectural ordering constraints; these may or may not be sufficient to prove the design correct. There exist microarchitectures for which this subset is not sufficient, and PipeProof currently cannot prove the correctness of those designs. However, the over-approximation of microarchitectural constraints is sufficient to prove the correctness of the designs in this paper.

E. Inductive ISA Edge Generation

There are often an infinite number of ISA-level executions that can match a forbidden ISA-level pattern. Thus, PipeProof must reason about these ISA-level executions inductively to make verification feasible. PipeProof's refinement loop inductively models additional instructions through concretization and decomposition (Section III-E and Figure 7). As such, PipeProof must also inductively generate the possible ISA-level relations that could connect these modelled instructions such that the overall execution matches the forbidden ISA-level pattern being checked.

Given an ISA-level pattern pat, PipeProof's ISA Edge Generation procedure returns all possible choices $(edge, remain)$, where $edge$ is a possible initial or final edge of pat, and $remain$ is the part of pat that did not match $edge$. If peeling from the left of a transitive chain, the procedure returns cases where $edge$ is an initial edge. If peeling from the right, the procedure returns cases where $edge$ is a final edge.

For example, if decomposing a transitive chain representing the pattern $(po \cup co); rf; fr$, the ISA Edge Generation procedure would return $(po, (rf; fr))$ and $(co, (rf; fr))$ if peeling from the left, so either po or co could be peeled off. Likewise, if peeling from the right, the procedure would return $(fr, ((po \cup co); rf))$, so only fr could be peeled off.

V. PipeProof Optimizations

PipeProof implements two optimizations that improve its runtime by greatly reducing the number of cases considered. This section explains these optimizations.

A. Covering Sets Optimization

The TC Abstraction guarantees at least one transitive connection between the start and end of an ISA-level chain that it represents. Thus, PipeProof needs to verify correctness for *each* possible transitive connection when using the TC Abstraction to represent an ISA-level chain. As seen in Figure 7, a new set of transitive connections comes into existence each time a transitive chain is decomposed. This can quickly lead to a large number of cases to consider. Even the `simpleSC` microarchitecture has 9 possibilities (3 nodes $*$ 3 nodes) for transitive connections between any two instructions. To mitigate this case explosion, PipeProof implements the *Covering Sets Optimization* to eliminate redundant transitive connections.

The key idea behind the Covering Sets Optimization is that if in a given scenario, a and b are possible transitive connections, and every μhb graph containing a also contains b, then it is sufficient to just check correctness when b is the transitive connection. In other words, b *covers* a. For example, $AbsCex$ in Figure 7 has a transitive connection between nodes p and q. This transitive connection covers other possible transitive connections such as the one from p to r used in $NoDecomp$. This is because there is no possible μhb graph satisfying the microarchitectural axioms that contains an edge from p to r without also having an edge from p to q (by transitivity). Given a set of transitive connections $conns$ for a given scenario along with all other scenario constraints, the Covering Sets Optimization eliminates transitive connections in $conns$ that are covered by other transitive connections in the set. This optimization significantly improves PipeProof runtime (details in Section VI-A).

B. Eliminating Redundant Work Using Memoization

Figure 7 shows PipeProof's procedure for proving that `simpleSC` is a correct implementation of SC. PipeProof first checks that all ISA-level cycles containing fr are microarchitecturally unobservable, and then does the same for cycles containing rf, po, and co. However, there is notable overlap between these four cases. For example, the ISA-level cycle $po; rf; po; fr$ from Figure 1b contains po, rf, and fr edges. A naive PipeProof implementation would verify this cycle (directly or indirectly through the TC Abstraction) 3 times: once as a cycle containing po, once as a cycle containing rf, and once as a cycle containing fr. The second and third checks of the cycle are redundant and can be eliminated.

PipeProof filters out cases that have already been verified by restricting the edges that can be peeled off during decomposition. For example, if all ISA-level cycles containing fr have been verified for `simpleSC`, then when checking all ISA-level cycles containing po, fr edges should be excluded from the choices of edges to peel off. This is because peeling off an fr edge would turn the ISA-level cycle being considered into a cycle containing fr (which has already been verified).

Stated formally, given an ISA-level MCM property $acyclic(r_1 \cup r_2 \cup ... \cup r_n)$, if all ISA-level cycles containing r_i have been verified $\forall i < j$, then the only choices for edges

to peel off when verifying cycles containing r_j should be $\{r_j, r_{j+1}, ..., r_n\}$. This optimization enables our TSO case study (Section VI-A) to be verified in under an hour.

VI. Methodology, Results, and Discussion

A. Methodology and Results

PipeProof is written in Gallina, the functional programming language of the Coq proof assistant [29]. PipeProof reuses the Check suite's μspec parsing and axiom simplification [19], and extends the Check suite solver to be able to model and verify executions of symbolic instructions. Writing PipeProof in Gallina allows for future formal analysis of the code, such as proving PipeProof's solver or proof procedures correct. PipeProof is an automated tool; we do not use the interactive theorem proving capabilities of Coq to prove microarchitectures. We use the built-in extraction functionality of Coq to extract our Gallina code to OCaml so it can be compiled and run as a standalone binary.

We ran PipeProof on two microarchitectures. The `simpleSC` microarchitecture has a 3-stage in-order pipeline and Store→Load ordering enforced. The `simpleTSO` microarchitecture is `simpleSC` with Store→Load ordering relaxed for different addresses. We verified `simpleSC` against the SC ISA-level MCM, while `simpleTSO` was verified against the TSO ISA-level MCM. The overall specification of TSO consists of two properties: $acyclic(po\_loc \cup co \cup rf \cup fr)$ and $acyclic(ppo \cup co \cup rfe \cup fr \cup fence)$ [2]. The po_loc relation models same-address program order, while ppo (preserved program order) relates instructions in program order except for Store→Load pairs (which can be reordered under TSO). The rfe (reads-from external) edge represents when a store sources a load on another ("external") core, and $fence$ relates instructions separated by a fence in program order.

Experiments were run on an Ubuntu 16.04 machine with an Intel Core i7-4870HQ processor. Table I breaks down PipeProof runtimes for five cases. We present results for `simpleSC` when (i) using vanilla PipeProof algorithms, (ii) with the Covering Sets Optimization (Section V-A), and (iii) with Covering Sets and Memoization (Section V-B). We also present results for `simpleTSO` when (iv) using Covering Sets, and (v) with Covering Sets and Memoization. (`simpleTSO` was infeasible without the Covering Sets Optimization.) PipeProof proves the correctness of `simpleSC` in under four minutes using vanilla PipeProof algorithms. The Covering Sets Optimization brings runtime down to under a minute, and Memoization reduces runtime further to under 20 seconds. Meanwhile, proving that `simpleTSO` correctly implements TSO takes just over five and a half hours with the Covering Sets Optimization. With the addition of Memoization, `simpleTSO` is verified in under 41 minutes.

While runtimes under an hour are quite acceptable, the verification of `simpleTSO` takes more time than the verification of `simpleSC` because TSO's additional relations increase the number of possibilities for ISA-level edges that can be peeled off a transitive chain. This has a multiplicative effect on the number of cases that need to be verified; each peeled-off instruction may require verification across many transitive connections, each of which may require further instructions to be peeled off. Nevertheless, with the help of its optimizations, PipeProof's verification of `simpleTSO` in under an hour shows that complete automated MCM verification of microarchitectures can indeed be tractable.

With regard to chain invariants, verifying `simpleSC` required one invariant (po_plus) to be provided to model repeated po edges. Meanwhile, verifying `simpleTSO` required five invariants, for repetitions of ppo, $fence$, po_loc, ppo followed by $fence$, and $fence$ followed by ppo.

PipeProof's detection of microarchitectural bugs was quite fast. As an example, we introduced a flaw into `simpleSC` relaxing Store→Load ordering. PipeProof produced a counterexample to that flaw in under a second (both with and without the Covering Sets optimization). Similarly, if Store→Load ordering for the same address was relaxed on `simpleTSO`, the bug was detected in under 2 seconds from the beginning of the check of the relevant ISA-level pattern.

B. Scalability

To scale performance to more complicated microarchitectures, we can parallelize the implementation of PipeProof's algorithm (Section III). The only dependency in the algorithm is that a given abstract execution (such as $AbsCex$ from Figure 7) must be checked before any concretizations or decompositions of it are checked. Apart from this dependency, each abstract or concrete execution can be checked independently of the others, making the algorithm $highly$ parallelizable and well-suited to the use of multicore machines and clusters to improve performance.

C. Future Work

We intend to build on PipeProof in future work by parallelizing it, handling read-modify-write instructions, the RISC-V [28], Power [13], and ARM [3] ISA-level MCMs, and chain invariant auto-generation. We would also like to modify the TC Abstraction support proof (Section IV-A) to only require that a microarchitecture guarantee a transitive connection when the transitive chain is part of a forbidden ISA-level cycle in the overall execution. Likewise, we also intend to derive a more accurate over-approximation (Section IV-D) that allows PipeProof to utilise more of the axioms from a microarchitectural ordering specification. These modifications would help PipeProof scale to more microarchitectures than it currently does.

VII. Related Work

ISA-level MCMs are often specified in the format used by the `herd` tool [2], consisting of irreflexivity, acyclicity, and emptiness requirements for certain relational patterns. Such specifications can be created by hand based on consultation with system architects and dynamic or formal analysis of implementations using litmus tests (e.g.: x86-TSO [2], Power [2],

Component	simpleSC	simpleSC (w/ Covering Sets)	simpleSC (w/ Covering Sets + Memoization)	simpleTSO (w/ Covering Sets)	simpleTSO (w/ Covering Sets + Memoization)
Chain Invariant Proofs	0.008 sec	0.01 sec	0.008 sec	0.5 sec	0.5 sec
TC Abstraction Support Proofs	2.8 sec	0.9 sec	0.9 sec	71.1 sec	67.3 sec
Microarch. Correctness Proofs	223.1 sec	35.5 sec	18.2 sec	19813.8 sec	2379.5 sec
Total Time	**225.9 sec**	**36.4 sec**	**19.1 sec**	**19885.4 sec**	**2449.7 sec**

TABLE I: PipeProof runtimes for `simpleSC` and `simpleTSO` with and without Covering Sets and Memoization.

ARM [2, 27], and RISC-V [28]). Relational logic-based ISA-level MCM specifications can also be automatically generated based on litmus test outcomes [5].

MCM verification of hardware implementations has mostly been conducted using litmus test-based approaches, both static [2, 18, 19, 23, 24, 30] and dynamic [12]. Schemes for generating better suites of litmus tests have also been developed [5, 20, 21]. ISA-level MCM analysis has also been conducted using approaches which check all programs up to a certain bounded size [33]. All these approaches suffer from the flaw of being incomplete and possibly missing bugs. Mador-Haim et al. [22] established a bound on litmus test completeness when comparing certain consistency *models*, but it is still unknown how to detect whether a test suite is complete with respect to a given parallel system *implementation*. As such, there is no way to tell whether passing a suite of litmus tests means that a design is correct for all programs. PipeProof surpasses these approaches by conducting complete, unbounded verification of all possible programs on a microarchitecture, giving users the confidence that their microarchitecture is indeed completely correct.

Chatterjee et al. [6] verify operational models of processor implementations against operationally specified ISA-level MCMs. Their approach has two main steps. The first step creates an abstract model of the microarchitectural implementation by abstracting away the external memory hierarchy, and verifies it by checking a refinement relation between the two models using model checking. The second step verifies this abstract model against an ISA-level MCM specification using theorem-proving. They only verify small instances (restricted to two processors, addresses, and values), while PipeProof's verification is *complete* across different core counts, addresses, and values. They also target verification of operational models, while PipeProof targets axiomatic models. Finally, they handle visibility order specifications, whereas PipeProof uses more general MCM specifications such as the acyclicity of certain ISA-level edge patterns.

The only complete proofs of microarchitectural MCM correctness that have been conducted to date are those of the Kami project [7, 31]. However, Kami utilises the Coq interactive theorem prover [29] for its proofs, which requires designers to know proof techniques and requires manual effort. This is not amenable for many computer architects. In contrast, PipeProof *automatically* proves microarchitectural MCM correctness when provided with an ISA-level MCM specification, μspec axioms, mappings, and invariants.

VIII. Conclusions

Memory consistency models and their verification are paramount to correct parallel system operation. The MCM verification of a hardware design must be *complete* in order to guarantee the correct execution of parallel programs on that hardware. However, prior automated microarchitectural MCM verification approaches only conduct bounded or litmus-test-based verification, which can miss bugs.

In response, this paper introduces PipeProof, the first methodology and tool for automated *complete* verification of axiomatic microarchitectural ordering specifications with respect to axiomatic ISA-level MCM specifications. PipeProof can either automatically prove a specified microarchitecture correct with respect to its ISA-level MCM or it can inform the user that that the microarchitecture could not be verified, often providing a counterexample to illustrate the relevant bug in the microarchitecture. PipeProof's novel Transitive Chain Abstraction allows it to inductively model and verify all microarchitectural executions of all possible programs. This enables efficient yet complete microarchitectural MCM verification; PipeProof is able to prove the correctness of microarchitectures implementing SC and TSO in under an hour. With PipeProof, verification can be moved much earlier in the design process. Furthermore, architects no longer have to restrict themselves to manual proofs of correctness of their designs in interactive theorem provers to achieve complete verification. Instead, they can use PipeProof to automatically prove microarchitectural MCM correctness for all possible programs, addresses, and values.

Acknowledgements

We thank Kedar Namjoshi and the anonymous reviewers for their helpful feedback. This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, and in part by the U.S. National Science Foundation through the grants XPS-15-33837 and XPS-16-28926.

References

[1] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] 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.

[3] ARM, "ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile," 2017. [Online]. Available: https://static.docs.arm.com/ddi0487/b/DDI0487B_a_armv8_arm.pdf

[4] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2009, vol. 185, ch. 26, pp. 825–885. [Online]. Available: http://www.cs.stanford.edu/ barrett/pubs/BSST09.pdf

[5] J. Bornholt and E. Torlak, "Synthesizing memory models from framework sketches and litmus tests," in *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[6] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, "Shared memory consistency protocol verification against weak memory models: Refinement via model-checking," in *14th International Conference on Computer Aided Verification (CAV)*, 2002.

[7] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, 2017.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *12th International Conference on Computer Aided Verification (CAV)*, 2000.

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

[10] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 38–55.

[11] M. Hachman, "Intel finds specialized TSX enterprise bug on Haswell, Broadwell CPUs," 2014, http://www.pcworld.com/article/2464880/intel-finds-specialized-tsx-enterprise-bug-on-haswell-broadwell-cpus.html.

[12] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," in *19th International Symposium on Computer Architecture (ISCA)*, 2004.

[13] IBM, "Power ISA version 2.07," 2013.

[14] Intel, "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

[15] ISO/IEC, "Programming Languages – C," International Standard 9899:2011, 2011.

[16] ——, "Programming Languages – C++," International Standard 14882:2011, 2011.

[17] 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.

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

[19] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[20] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[21] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Generating litmus tests for contrasting memory consistency models," in *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[22] ——, "Litmus tests for comparing memory consistency models: How long do they need to be?" in *48th Design Automation Conference (DAC)*, 2011.

[23] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," in *50th International Symposium on Microarchitecture (MICRO)*, 2017.

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

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

[26] 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.

[27] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8," in *45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2018.

[28] RISC-V Foundation, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2," May 2017.

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

[30] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[31] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, "Modular deductive verification of multiprocessor hardware designs," in *27th International Conference on Computer Aided Verification (CAV)*, 2015.

[32] M. Walton, "Intel Skylake bug causes PCs to freeze during complex workloads," 2016, https://arstechnica.com/gadgets/2016/01/intel-skylake-bug-causes-pcs-to-freeze-during-complex-workloads/.

[33] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.