PipeSynth: Automated Synthesis of Microarchitectural Axioms for Memory Consistency

Chase Norman
c_@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Adwait Godbole
adwait@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Yatin A. Manerkar
manerkar@umich.edu
University of Michigan
Ann Arbor, MI, USA

ABSTRACT

Formal verification can help ensure the correctness of today's processors. However, such formal verification requires formal specifications of the processors being verified. Today, these specifications are mostly written by hand, which is tedious and error-prone. Furthermore, architects and hardware engineers generally do not have formal methods experience, making it even harder for them to write formal specifications. Existing methods for the automated synthesis of formal microarchitectural specifications utilise RTL implementations of processors for their synthesis, preventing their usage until RTL implementation of the processor has completed. This hampers the effectiveness of formal verification for processors, as catching design bugs pre-RTL can reduce verification overhead and overall development time.

In response, we present PipeSynth, an automated formal methodology and tool for the synthesis of μspec microarchitectural ordering axioms from small example programs (litmus tests) and microarchitectural execution traces. PipeSynth helps architects automatically generate formal specifications for their microarchitectures before RTL is even written, enabling greater use of formal verification on today's microarchitectures. We evaluate PipeSynth's capability to synthesise single axioms and multiple axioms at the same time across four microarchitectures. Our evaluated microarchitectures include an out-of-order processor and one with a non-traditional coherence protocol. In single-axiom synthesis, PipeSynth is capable of synthesising replacement axioms for 42 out of 46 axioms from our evaluated microarchitectures in under 2 hours per axiom. When doing multi-axiom synthesis, we are able to synthesise an entire microarchitectural specification for the in-order Multi-V-scale processor in under 1 hour, and can synthesise at least 4 nontrivial axioms at the same time for our other microarchitectures.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; • **Software and its engineering → Formal methods**; • **Theory of computation → Parallel computing models**.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582056

KEYWORDS

formal methods, synthesis, memory consistency, microarchitecture

ACM Reference Format:
Chase Norman, Adwait Godbole, and Yatin A. Manerkar. 2023. PipeSynth: Automated Synthesis of Microarchitectural Axioms for Memory Consistency. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582056

1 INTRODUCTION

Hardware architectures are becoming more and more complex. Today's Systems-on-Chip (SoCs) typically contain not just multicore processors, but also specialised accelerators for various types of computational tasks. This complexity makes it difficult for architects and hardware engineers to build correct processors. A number of high-profile bugs have been found in processors in recent years, including security vulnerabilities [23, 25, 44] and concurrency-related bugs [17]. Hardware verification is thus critical for ensuring the correctness of today's computer systems, with verification costs now dominating total hardware design cost [12]. With the advent of accelerator-level parallelism [18], this trend is set to continue.

Formal methods can provide strong correctness guarantees for computing systems (including hardware architectures) based on mathematical proofs. Given a formal specification of the system to verify, automated formal verification approaches can prove that the system satisfies a certain property (sometimes for a restricted set of programs). If verification fails, the verification procedure can often provide a counterexample (an execution that does not satisfy the property). There has been an uptick in formal verification research related to computer architecture in recent years, both in academia [20, 34, 35, 51, 53, 54, 57, 60] and industry [28, 45].

A key challenge in the use of formal verification for architecture and microarchitecture is that the vast majority of formal verification approaches require formal specifications (i.e., formal models of the system being verified) for their usage. Most of the time, these formal specifications must be written by hand, which is both tedious and error-prone for today's complex microarchitectures. In addition, most architects and hardware engineers do not have formal methods expertise, making it hard for them to even write formal specifications in the first place. A formal specification used for verification must be sound with respect to the design it is modelling, or this can lead to the verification succeeding even if the real design is buggy.

The need for formal specification and verification is especially pertinent for memory consistency models (MCMs) in parallel architectures. MCMs specify the ordering rules governing memory

and synchronization operations in parallel programs, and constrain the values that can be returned by load instructions. MCMs are notoriously difficult to specify, as the complexity of today's microarchitectures leads to many corner cases that need to be accounted for by such specifications. It is also very hard to verify MCM implementations, as the nondeterminism of today's multicore architectures requires formal verification to ensure that the MCM will always be respected by the hardware. In response, there has been much work on formal specification and verification for hardware MCMs, especially over the past decade [1, 2, 6, 8, 10, 11, 14, 27–29, 31, 34–36, 38, 43, 46, 48, 54, 56, 60].

To alleviate the difficulties of writing complicated code such as formal MCM specifications, *program synthesis* techniques use formal methods to automatically generate programs that match a high-level formal correctness specification. A state-of-the-art synthesis method is that of *syntax-guided synthesis* (SyGuS) [3]. In SyGuS, a context-free grammar (like those used by parsers) specifies the space of possible function implementations. A Satisfiability Modulo Theories (SMT) solver then searches over the space of possible implementations to find an implementation of the function that satisfies the high-level correctness specification. The solver can either return a correct implementation or state that no valid implementation of the function exists under the given parameters. There has been much work on program synthesis over the years [3, 21, 22, 40, 42, 49, 50, 52].

There has also been prior work on synthesising formal architectural and microarchitectural specifications [6, 19, 51, 58, 59]. MemSynth [6] can automatically synthesise formal ISA-level MCM specifications, but has no capability to generate formal microarchitectural specifications. Meanwhile, RTL2μspec [19] and the Instruction-Level Abstraction (ILA) line of work [51, 58, 59] are capable of automatically generating microarchitecture-level specifications, but they use an RTL implementation of the hardware design in order to accomplish their synthesis. As a result, the formal specifications they can create are generated quite late in the design timeline, after the RTL implementation of the processor has been created. If a formal microarchitectural model can only be generated post-implementation, then engineers cannot formally verify that the microarchitecture is a correct design (i.e., that it correctly respects the ISA) until the implementation has been completed. Such an implementation will likely have microarchitectural design bugs that must be subsequently fixed, so the effort spent by engineers to create the buggy parts of the implementation is effectively wasted. On the other hand, if formal methods are used to catch microarchitectural design bugs during early-stage design, then engineers will not waste time creating incorrect implementations. Such use of formal methods requires a formal specification of the design, leading to a need for *pre-RTL* formal microarchitectural specification synthesis.

In response, this paper proposes PipeSynth[1], an automated methodology and tool for the pre-RTL synthesis of formal microarchitectural ordering specifications. Given a set of litmus tests[2] (which function as input-output examples), execution traces (which can be generated by pre-RTL architectural simulators like gem5 [5]), a partial microarchitectural ordering specification, and a grammar specifying the space of possible ordering rules, PipeSynth can use SyGuS to automatically generate additional microarchitectural ordering axioms (invariants) that are needed to satisfy the litmus tests and execution traces. PipeSynth uses and synthesises ordering specifications in the μspec domain-specific language for microarchitectural orderings [29]. PipeSynth allows architects and engineers to specify allowed and forbidden MCM behaviours using formats that are more familiar to them (namely, litmus tests and simulator traces), and then automatically generates formal ordering axioms necessary to enforce those orderings. In this way, PipeSynth significantly reduces the barrier to entry for architects wishing to create formal microarchitectural ordering specifications pre-RTL. These formal specifications can then be used for early-stage design-time verification to catch design bugs that would otherwise result in MCM violations [15, 27, 29, 34, 36, 54].

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

(a)

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

(b)

Figure 1: (a) Code for the litmus test `mp`, which is forbidden under SC. (b) Code for a variant of `mp` that switches the order of the two stores on core 0 and is thus allowed by SC.

The contributions of this paper are:

- **Pre-RTL Synthesis of Ordering Axioms:** PipeSynth is the first automated methodology and tool for the pre-RTL formal synthesis of μspec axioms from litmus tests and execution traces.
- **Novel SyGuS Encoding for Axiom Synthesis:** The naive use of SyGuS cannot synthesise μspec axioms, so we develop a novel *conjunct-based encoding* that makes μspec synthesis with SyGuS feasible for multiple common types of μspec axioms.
- **Demonstration:** We synthesise axioms for a variety of microarchitectural specifications, including both in-order and out-of-order processors. We are able to synthesise the majority of axioms for these microarchitectural specifications.

The rest of this paper is organised as follows. Section 2 provides background on μspec-based microarchitectural MCM verification and SyGuS. Section 3 provides an overview of PipeSynth. Section 4 describes the notable challenges involved in μspec axiom synthesis. Section 5 explains PipeSynth's operation, focusing on its novel conjunct-based encoding. Section 6 covers auxiliary synthesis options that users of PipeSynth can utilise to improve PipeSynth's synthesis capabilities. Section 7 describes our experimental methodology and Section 8 presents our results. Section 9 covers related work, and Section 10 concludes.

2 BACKGROUND

2.1 μspec Microarchitectural MCM Verification

MCMs specify the ordering requirements on memory and synchronization operations in parallel systems. The simplest MCM is sequential consistency (SC) [24]. SC requires the results of memory

[1] open-source and publicly available at github.com/chasenorman/PipeSynth-AEC.
[2] Litmus tests are 4-8 instruction programs designed to test specific memory consistency scenarios.

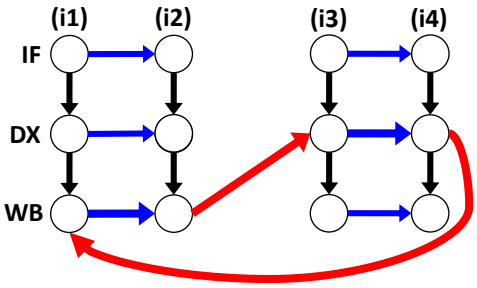

Figure 2: Example μhb graph for `mp` litmus test (Figure 1a) on Multi-V-scale processor [47].

```
Axiom "DecodeExecute_stage_is_in_order":
forall microops "i",
forall microops "j",
  (~SameMicroop i j /\
   AddEdge ((i, IF), (j, IF))) =>
     AddEdge ((i, DX), (j, DX)).
```

Figure 3: An example μspec axiom.

operations to be consistent with a total order on all memory operations, where instructions from individual threads or cores appear to execute in program order and each load reads from the last store to its address in the total order.

MCM analysis often uses litmus tests, which are typically small 4-8 instruction programs designed to illustrate a specific MCM scenario. In litmus test convention, the initial values at all memory addresses are 0. For example, Figure 1a shows the litmus test `mp`, which depicts the idiom of message passing. In `mp`, core 0 writes to data variable `x` and then sets a flag variable `y`. Core 1, meanwhile, reads the value of `y` and then reads the value of `x`. Under SC, it is impossible for core 1 to see the write to `y` but not the write to `x`, as there is no total order on memory operations that obeys program order that would allow this. On the other hand, if we reordered the two writes on core 0 (as Figure 1b depicts), the test would become allowed.

Prior work from the Check suite [27, 29, 35, 36, 54] developed methodologies and tools for the formal verification of litmus tests on microarchitectures. This work represented microarchitectural executions as μhb graphs where nodes represent sub-events in instruction execution and edges represent happens-before relationships between such events. Figure 2 shows a μhb graph for the outcome of `mp` forbidden under SC for the open-source Multi-V-scale processor [47]. Multi-V-scale has 3-stage in-order pipelines of Fetch (`IF`), a combined Decode+Execute (`DX`), and Writeback (`WB`) stages, and aims to implement SC. Nodes in the μhb graph represent the different pipeline stages of instruction execution, with each column of nodes representing an instruction flowing through the pipeline. For instance, the second node in the first column represents instruction `i1` at its `DX` stage, while the third node in the second column represents instruction `i2` at its `WB` stage.

Edges in μhb graphs, meanwhile, represent happens-before relationships between the nodes that they connect. For instance, the edge between the first two nodes in the second row of the μhb graph

```
Grammar:
  f(x) := E
  E := E + E | E * E | x | 0 | 1

Constraints:
  assert(f(x) > x && f(3) == 4)
```

Figure 4: An example of a SyGuS synthesis problem.

represents that instruction `i1` goes through its `DX` stage before instruction `i2` does. Since each edge represents a happens-before relationship, a cycle in a μhb graph indicates that an event has to happen before itself, which is impossible. Thus, cyclic μhb graphs constitute executions that are unobservable on the modelled microarchitecture, while acyclic μhb graphs constitute executions that are observable on the modelled microarchitecture. The overall μhb graph in Figure 2 is cyclic, indicating that this particular execution of `mp` is unobservable on Multi-V-scale—as we would expect of an SC microarchitecture.

The decision of when and where to add edges to μhb graphs in microarchitectural MCM verification is dictated by a microarchitectural specification in the domain-specific μspec language. A μspec specification consists of a set of axioms or invariants dictating ordering properties that must be maintained by the microarchitecture. Each axiom represents one of the individual smaller orderings that combine to enforce the MCM of the overall processor. For example, Figure 3 shows one of the μspec axioms for Multi-V-scale. This axiom says that for any two distinct instructions `i` and `j` in the test, if `i` goes through its `IF` stage before `j` does, then `i` must also go through `DX` before `j` goes through that stage. Instructions `i1` and `i2` (as well as `i3` and `i4`) are fetched in order, so this axiom thus results in the addition of edges between the `DX` stages of `i1` and `i2` and between those of `i3` and `i4`. Other axioms in Multi-V-scale's μspec specification enforce the other blue horizontal edges enforcing in-order execution for instructions on the same core. Additional axioms cause the addition of the black vertical edges between the nodes of a single instruction and the addition of the red edges indicating the ordering of loads and stores on different cores with respect to each other.

To verify a μspec microarchitectural specification against a litmus test, the Check tools instantiate the specification's axioms for the litmus test and use an SMT solver to search for an acyclic μhb graph that satisfies all the axioms. If such an acyclic μhb graph is found, then the litmus test is observable on the microarchitecture. If the test was forbidden, its observability means that the microarchitecture is buggy. On the other hand, if the test is required to be observable, then an acyclic μhb graph satisfying the axioms must exist for it.

2.2 Syntax-Guided Synthesis (SyGuS)

Program synthesis aims to automatically generate implementations of functions (henceforth called *synthesis functions*) that match a high-level correctness specification. Syntax-guided Synthesis (SyGuS) [3] is a flavour of program synthesis that restricts the space of possible function implementations using context-free grammars (like those used in parsers).

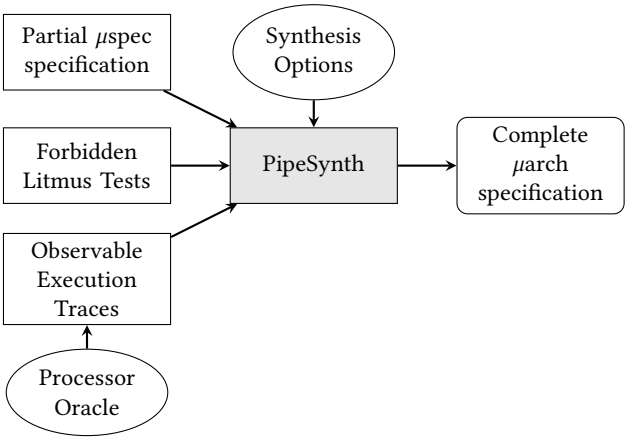

Figure 5: PipeSynth block diagram. Rectangles are required; circular nodes are optional. The rounded rectangle represents PipeSynth's output upon successful synthesis.

```
pc:0x8  stage:IF  t:4 unit:core0
...
pc:0x1c stage:IF  t:5 unit:core1
pc:0x18 stage:DX  t:5 unit:core1
pc:0x8  stage:DX  t:6 unit:core0
...
```

Figure 6: Execution trace fragment from Multi-V-scale.

SyGuS is a standardised format for the specification of syntax-guided synthesis problems [39], and it is supported by multiple synthesis solvers. A synthesis solver searches over a space of possible function implementations (often called *candidate solutions*) and tries to find an implementation that satisfies the correctness specification. In this paper we use the CVC5 solver [4].

Figure 4 shows a pedagogical example of a SyGuS problem formulation. The synthesis function $f(x)$ is restricted by the grammar. The grammar allows multiplication and addition operators over the function's input and the constants 0 and 1. In addition, the generated implementation must satisfy the constraint that $f(3) = 4$ and $f(x)$ must be greater than x for all x. When provided with such a SyGuS query, a synthesis solver like CVC5 will search over the space of possible implementations to try and find one that satisfies the examples and grammar constraints. In this example, it finds the implementation $f(x) = x + 1$, which satisfies the constraints and abides by the grammar. Thus, this implementation could be returned as the solution to the synthesis problem. In general, there may be several correct implementations and the solver may return any one of them.

Constraints on the synthesis functions may be provided as assertions that must hold (as in Figure 4), or they can be generated from *oracles* (human input or software/hardware runtimes). The flavour of synthesis where oracles are used to generate constraints is referred to as oracle-guided inductive synthesis (OGIS) [21].

3 PIPESYNTH OVERVIEW

Figure 5 shows PipeSynth's high-level organisation. The three required inputs to PipeSynth are a partial microarchitectural specification, a set of forbidden litmus tests, and a set of observable execution traces (all described below). PipeSynth then synthesises additional μspec axioms for the microarchitectural specification that are necessary to refute the negative tests while allowing observable executions. PipeSynth terminates when the synthesised axiom(s) + other provided axioms are sufficient to forbid all forbidden tests and allow all required execution traces, or when the solver concludes that synthesis is impossible for the given scenario and grammars. Users may leverage PipeSynth's synthesis options (Section 6) to improve their synthesis results. They may also use processor simulators as oracles[3] to generate observable execution traces (Section 5.5).

The first input to PipeSynth is the partial μspec specification. Users provide a partial μspec specification that consists of a set of microarchitectural axioms and placeholder functions for the remaining axioms. PipeSynth replaces these placeholder functions with synthesised μspec such that the overall μspec specification respects the constraints of the litmus tests and execution traces provided by the user. The synthesis of one axiom may require multiple such placeholder functions due to our novel synthesis encoding (Sections 5.1 to 5.3), so PipeSynth provides utility APIs to create the appropriate *templates* of placeholder functions for a number of common axiom types (Section 5.2). These built-in templates (either alone or in combination) can be used to synthesise replacement axioms for all but one of the 46 axioms in our evaluated microarchitectures for the litmus tests and execution traces we use. For axioms that do not conform to one or more of our built-in templates, users may create custom templates by putting placeholder functions in a custom μspec axiom structure (Section 6.3).

The second input is the set of forbidden litmus tests. We recommend that these tests exhibit a variety of ISA-level orderings so as to cover a wide range of possibilities. We use the `litmustestgen` tool [30] to generate our tests. Given a formal ISA-level MCM specification, `litmustestgen` can automatically generate all *boundary litmus tests* (of size up to a bound) for that ISA MCM. A boundary litmus test is a forbidden litmus test for which any relaxation would result in it being allowed. For example, the reordering of core 0's writes from Figure 1a to Figure 1b is a relaxation that makes the test allowed. (PipeSynth does not require the use of `litmustestgen`. Users can provide forbidden litmus tests from other sources, e.g., handwritten tests.)

If we only provide forbidden litmus tests, an axiom which disallows all executions of all programs will be a consistent solution to them. This is clearly nonsensical, as a microarchitecture obeying such an axiom could never generate a result for a program. Thus, we must also provide the synthesis procedure with test cases that must be allowed by the microarchitecture. We generate such "required" litmus tests by taking our forbidden litmus tests and reordering one of the pairs of instructions in each of them. This constitutes a relaxation to each such test which is guaranteed to make it allowed (since `litmustestgen` only generates boundary litmus tests).

However, merely providing a synthesis procedure with permitted litmus tests is insufficient to synthesise realistic μspec axioms, because litmus test outcomes do not provide any information about *how* such litmus tests should execute on the microarchitecture. This

[3]In synthesis terminology, an oracle is an entity (including humans) that can be used by the synthesis procedure through a query-response interface [21, 39, 40].

issue is a key challenge in μspec synthesis (covered in Section 4.3). We solve this problem by annotating permitted litmus tests with timestamped execution traces like the one in Figure 6 (Section 5.5 provides details). These execution traces provide the synthesis procedure with a *witness* for permissibility of the litmus test. This allows PipeSynth to synthesise axioms that accurately reflect the event orderings in such executions. These execution traces can be generated in a straightforward manner by instrumenting pre-RTL simulators such as gem5 [5].

PipeSynth encodes the user's synthesis problem in SyGuS using our novel synthesis encoding. PipeSynth then passes this file to a SyGuS solver like CVC5 [4], which attempts to solve it. If the solver returns a synthesis result, this indicates that a completed microarchitectural specification was found which outlaws the forbidden litmus tests and permits the required execution traces. On the other hand, if the solver returns that the synthesis failed, it is impossible to create additional axioms matching the parameters used that outlaw the forbidden litmus tests while permitting the required traces. PipeSynth's use of SyGuS also lets us take advantage of advances in SyGuS solvers.

The user can ask PipeSynth to synthesise a single axiom or multiple axioms, depending on the number of templates they choose to use in the synthesis query. (Note that if an axiom is unnecessary, it can just be synthesised as `True`.) We look at both use cases in our experiments (Section 7.3). For single-axiom synthesis, we synthesise each axiom when given all other axioms of the microarchitecture. For multi-axiom synthesis, we remove increasing numbers of axioms from the microarchitecture and attempt to synthesise them all. Axioms are removed from the microarchitecture in the order of the high-level templates they conform to. Within a set of axioms that conform to a given template, we remove them in ascending order of their single-axiom synthesis runtimes.

Users could attempt to synthesise multiple axioms with PipeSynth iteratively. Specifically, they could first attempt to synthesise many axioms with PipeSynth. If this fails, they could add a handwritten axiom to the μspec, and then re-attempt to synthesise the remaining axioms, repeating as necessary. Our multi-axiom synthesis experiments (Section 7.3) form a mirror image of such a flow.

4 CHALLENGES IN μSPEC SYNTHESIS

This section describes the significant challenges involved in synthesising μspec axioms.

4.1 SyGuS for Quantified Expressions

Virtually all μspec axioms begin with quantifiers (`forall` or `exists`). However, synthesis with quantifiers is known to be very challenging and requires specialized, heuristic-based techniques [9, 41]. Furthermore, the SyGuS standard (and hence CVC5 [4]) does not allow grammars with quantifiers. While "unrolling" finite quantification into a conjunction (disjunction) for $\forall$ ($\exists$) respectively is a potential solution, all the conjuncts (disjuncts) need to be identical for this to work. This requires context-sensitivity, which is impossible for the context-free grammars required by SyGuS solvers like CVC5.

PipeSynth solves this problem by synthesising only the quantifier-free body of the axiom, and specifying the constraint as a finite conjunction/disjunction over the synthesised expression. However, using a naive SyGuS encoding of the quantifier-free portion of the axiom runs into performance issues as we now discuss.

4.2 Poor Performance of a Naive Encoding

A natural way to encode the axiom synthesis problem into SyGuS is to translate placeholder functions in the incomplete microarchitectural specification directly to a single synthesis function in SyGuS. This typically entails trying to synthesise the body of an axiom (i.e., its quantifier-free portion) as a monolithic SyGuS query. Figure 7a shows a graphical depiction of possible synthesis solutions when using such an encoding. The grammar of this synthesis function would include the logical connectives (e.g., $\wedge$, $\implies$) and predicates from μspec (e.g., `IsRead(i)`, `AddEdge((i, IF), (i, DX))`).

Empirically, this naive encoding leads to infeasible SyGuS problems. For instance, using such a naive encoding, synthesis of the axiom responsible for constraining load values on our simplest microarchitecture (Multi-V-scale [32]) fails to finish even after 20 hours on a 2016 MacBook Pro with a Core i7 processor and 16 GB of RAM.

In hindsight, this result is not very surprising. Figure 7a indicates how the space of expressions resembles a tree in this encoding. Even for expressions with just a few terms, there are a large number of possibilities for the solver to consider across the various logical connectives and predicates. Crucially, many of these possibilities are redundant. For instance, in this naive encoding, a solver would consider both $\neg a \vee b$ and $a \implies b$ for any a and b, despite them being semantically equivalent. Likewise, it would also consider both $a \wedge b$ and $b \wedge a$ for a given a and b, despite only needing to consider one of the two. PipeSynth addresses this redundancy by developing a novel encoding of our synthesis problem (Sections 5.1 to 5.3) that exploits the particular structure of μspec axioms and precludes the examination of many redundant synthesis candidates.

4.3 Execution Trace Requirement

As Section 3 covers, μspec axiom synthesis requires both permitted and forbidden litmus tests in order to work. However, knowledge of permitted (ISA-level) litmus tests does not contain enough information to infer microarchitectural execution details. This is because distinct behaviours at the microarchitectural level (distinct μhb graphs) may have identical ISA-level observations. Consider the allowed variant of `mp` (Figure 1b). In this outcome, it is clear that the load `i3` of `y` reads from the store `i1`. However, this says nothing about whether the value of `y` was propagated from core 0 to core 1 through a shared store buffer, a cache-to-cache transfer, the last level cache, main memory, or some other path.

Synthesis in the absence of such microarchitectural information can result in unreasonable μspec axioms. For instance, if required to enforce in-order pipeline execution to forbid litmus tests, a synthesis procedure could simply enforce that every instruction commits before any subsequent instruction is fetched. While this would enforce correctness for the relevant litmus tests, it is not an axiom that any reasonable pipelined microarchitecture would ever want to abide by. This suggests that in order to guide the synthesis towards more precise axioms, we need additional constraints on the microarchitectural executions.

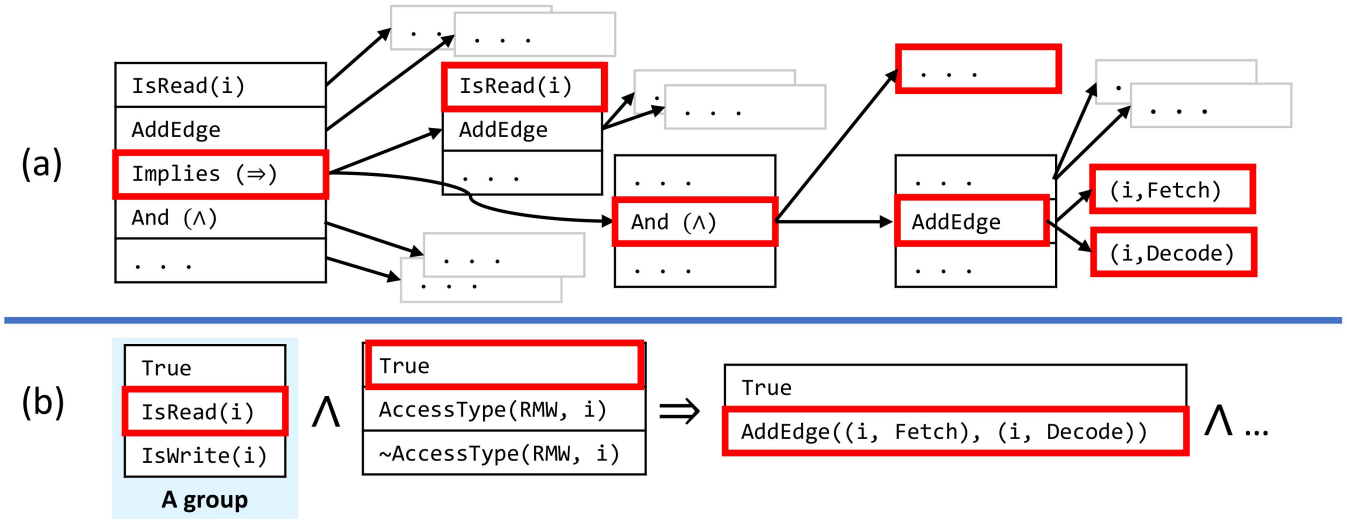

Figure 7: (a) A monolithic synthesis encoding contrasted with (b) our structured conjunct-based encoding of a path axiom. Red boxes represent the decisions made by the solver from groups (columns) of black boxes. Our conjunct-based encoding prevents a synthesis solver from examining numerous redundant and nonsensical candidate solutions. This greatly reduces the state space of solutions that the solver must search over and makes synthesis of μspec axioms feasible.

PipeSynth solves this challenge by augmenting permitted litmus tests with execution traces, and ensuring that its synthesis results respect these execution traces. Section 5.5 provides details.

5 PIPESYNTH OPERATION

This section covers PipeSynth's operation, focusing on our novel *conjunct-based encoding* of μspec axiom synthesis. As Section 4.2 covers, a naive SyGuS encoding of the quantifier-free portion of a μspec axiom results in synthesis being infeasible. Our key insight with the conjunct-based encoding is that information about the structure of standard μspec axioms is lost in a naive SyGuS grammar. This causes a solver conducting synthesis with such a grammar to examine many nonsensical and redundant candidate solutions. Our conjunct-based encoding conveys more detailed information about the structure of μspec axioms to the synthesis procedure, allowing it to focus on likely candidate solutions and ignore many nonsensical and redundant ones.

Our encoding makes use of template-based synthesis as described in Section 5.1. Section 5.2 describes the specific set of templates we use. These templates contain conjunctions of μspec predicates, which we synthesise using Section 5.3's technique. Section 5.4 discusses the generality of our encoding, and Section 5.5 explains how to generate the execution traces that PipeSynth takes in as one of its inputs.

5.1 Template-Based Synthesis

First, we make the key observation that the high-level structure of the vast majority of μspec axioms conforms to three patterns (or combinations of those patterns). These patterns (and combinations of them) cover all but one of the axioms in our evaluated microarchitectures. Section 5.2 covers the three patterns in detail. In line with prior work on sketch-based synthesis, e.g. [6, 49, 52], we create three *templates* (also called *sketches* in the synthesis literature) corresponding to these high-level patterns, and enforce that any synthesis solutions considered by PipeSynth must conform to our built-in templates or to custom templates (Section 6.3) created by the user. The templates contain *holes* (i.e., synthesis functions) that must be filled by the synthesis solver with terms corresponding to specific grammars to generate the overall μspec axiom. Conducting such template-based synthesis prevents the synthesis solver from examining candidate solutions that do not correspond to the given templates. This results in a significant reduction in the state space that must be examined by the synthesis solver, which can notably reduce PipeSynth's runtime.

5.2 Axiom Templates

The majority of μspec axioms fall into one of three types: path axioms, FIFO axioms, and sourcing axioms. Path axioms (Section 5.2.1) govern the ordering of events within a single instruction (i.e., the path it takes through the microarchitecture). FIFO axioms (Section 5.2.2) govern orderings between pairs of instructions, particularly those where one edge between them implies the existence of another. Sourcing axioms (Section 5.2.3) govern orderings that arise due to the values written and read by instructions. These three types of axioms were initially outlined by PipeCheck [27] before the μspec language was developed. They were intended to form a general organisation of axioms capable of describing a wide variety of microarchitectures. We formalise the μspec patterns that correspond to these types of axioms as synthesis templates and use them to reduce the state space that our synthesis solver must examine. We believe that these three templates serve as a very good base of templates for microarchitectures in general. This is evidenced by the fact that all but one of the axioms in our evaluated

```
Axiom "ReadsPath":
forall microops "i",
  IsRead(i) => (AddEdge((i, IF), (i, DX))
    /\ AddEdge ((i, DX), (i, WB))).
```

Figure 8: An example μspec path axiom.

microarchitectures conform either to one or a combination of these templates (Section 5.4 provides details on the latter). Axioms that are not covered by our built-in templates or a combination of them can be handled using custom templates (Section 6.3).

5.2.1 Path Axioms. Path axioms impose constraints on the order of events seen in the execution of an instruction. For instance, Figure 8's axiom imposes constraints on the sequence of stages seen in Read instructions. Path axioms are of the following form:

$$\forall i.\ (P(i) \implies Q(i)) \tag{1}$$

where $P(i)$ is a conjunction of predicates on instruction i that does not involve the `AddEdge` predicate (`IsRead(i)` is $P(i)$ in Figure 8). $Q(i)$, on the other hand, is a conjunction of positive occurrences of `AddEdge` for instruction i. These restrictions are not problematic because path axioms are usually conditioned on the type of instruction (which does not require `AddEdge`) and add edges between nodes of that instruction (which requires only `AddEdge`). We encode path axioms by fixing the above template and synthesising the conjunctions P and Q using Section 5.3's technique.

5.2.2 FIFO Axioms. Now we consider axioms which have universal quantification over two instructions. The most common case amongst these is where axioms enforce one edge conditioned on another edge existing in the graph. Since these axioms preserve order across two pairs of events, they are called FIFO axioms. Figure 3 shows an example FIFO axiom from Multi-V-scale. The template for axioms of this form is:

$$\forall i\ \forall j.\ (\neg\texttt{SameMicroop}(i,j) \wedge P(i,j)) \implies Q(i,j) \tag{2}$$

Here we require P to be a conjunction over instructions i and j, and Q to be a conjunction of positive occurrences of the `AddEdge` predicate over i and j. The $\neg$`SameMicroop`(i,j) condition ensures that the `AddEdge` constraints in $Q(i,j)$ are between two distinct instructions. Unlike path axioms, we allow P to contain `AddEdge` and its negation. This allows FIFO axioms to express a disjunction of μhb edges (see also Section 5.4) through the following equivalence (where $AddEdge_1(...)$ and $AddEdge_2(...)$ are instances of the `AddEdge` predicate):

$$\begin{aligned} P(i,j) &\implies (AddEdge_1(...) \vee AddEdge_2(...)) \\ &\equiv (P(i,j) \wedge \neg AddEdge_1(...)) \implies AddEdge_2(...) \end{aligned} \tag{3}$$

5.2.3 Sourcing Axioms. Sourcing axioms often express constraints over the sourcing of read instructions. These axioms have a double quantifier alternation (i.e., a forall-exists-forall pattern). Figure 9 shows an example sourcing axiom, with some details elided for brevity.

Figure 9's axiom requires that for any read instruction `i` that does not read from the initial state of memory, there must be a write `w` (the "source") such that there is no other write instruction `w'` to the same address between `w` and `i` (enforced by ensuring

```
Axiom "Read_Values":
forall microops "i",
 (IsRead(i) /\
  ~DataFromInitialStateAtPA(i)) =>
   exists microop "w",
    IsWrite(w) /\ ReadsFrom(i, w) /\ (
     forall microop "w'",
       WriteOnSameAddr(w, w') =>
        (AddEdge(w', w) \/ AddEdge(i, w'))
    ).
```

Figure 9: Example sourcing axiom, with some details elided for brevity. The `DataFromInitialStateAtPA` predicate here can be part of a discriminator (Section 5.4).

appropriate edges among `i`, `w`, and `w'`). Again, it does not make sense for the instructions quantified over to be identical, so we explicitly enforce this by using the ¬`SameMicroop` predicate. This results in the following template for sourcing axioms where $P(i)$, $Q(i,j)$, and $R(i,j,k)$ are conjunctions. (Note that the $\neg\exists$ in the innermost quantifier is equivalent to a $\forall$.)

$$\begin{aligned} \forall i.\ (P(i) \implies \exists j.\ (&\neg\texttt{SameMicroop}(i,j) \wedge Q(i,j) \\ &\wedge \neg\exists k(\neg\texttt{SameMicroop}(i,k) \\ &\wedge \neg\texttt{SameMicroop}(j,k) \wedge R(i,j,k)))) \end{aligned} \tag{4}$$

5.3 Synthesising Conjunctions

In our synthesis templates (Section 5.2), we deliberately enforce that conjunctions like $P(i)$ and $Q(i,j)$ each correspond to some conjunction of μspec predicates, i.e., $C_0 \wedge C_1 \wedge C_2 \wedge ... \wedge C_n$ where each conjunct C_i is a μspec predicate (or its negation). In our encoding, each such conjunct corresponds to a hole (i.e., synthesis function) to be filled by the synthesis solver with a term from a specified grammar. For example, in Figure 7b, the antecedent of the implication corresponds to such a conjunction.

Focusing on conjunctions allows us to fix the logical structure of that portion of the axiom. This prevents the solver from examining other redundant logical structures for that portion of the axiom. For instance, for a given a and b, a solver only needs to examine one of $a \wedge b$ and $\neg(\neg a \vee \neg b)$, as both are semantically equivalent. Under our encoding, the solver would only examine the first. Our conjunct-based encoding is quite expressive; Section 5.4 provides more detail on how our encoding can capture different logical structures.

We further restrict the state space of possible conjunction solutions by only allowing a few predicates to be a solution to each conjunct. These constraints are enforced through appropriate SyGuS grammars for those synthesis functions. The possible solutions for a given conjunct are referred to as a *group*. For instance, in Figure 7b, the group of possible solutions for the first conjunct consists of `True`, `IsRead(i)`, and `IsWrite(i)`.

Groups prevent the synthesis solver from examining many nonsensical and redundant solutions, e.g., where two of the conjuncts are mutually exclusive (in which case the conjunction evaluates to false) or where one of the conjuncts implies the other (in which case one of the conjuncts is unnecessary). For instance, if we allow conjuncts to be filled with any μspec predicate, a solver could consider solutions like `IsRead(i) /\ ~IsRead(i)`, which will always

evaluate to false. As an example of a case where one predicate implies another, consider `ProgramOrder(i, j) /\ SameCore(i, j)`. If `i` and `j` are in program order with respect to each other, they must be on the same core as well, so the `SameCore` predicate here is unnecessary. We prevent such scenarios by ensuring that mutually exclusive predicates and predicates where one implies the other are always in the same group, so that only one of them can be selected.

Instances of the same predicate on different micro-operations will be placed in different groups. So for example, for two micro-operations i and j, `IsRead(i)` and `IsRead(j)` will be placed in separate groups as one does not directly depend on the other.

The assignment of different groups to different conjuncts also reduces redundancy. For instance, in Figure 7b, the solver will examine `IsRead(i) /\ AccessType(RMW, i)` as a potential solution to the antecedent conjunction, but not `AccessType(RMW, i) /\ IsRead(i)` (which is semantically equivalent). (`RMW` stands for read-modify-write.)

Finally, every group also includes `True`, since we instantiate the conjunctions in our templates with a large number of conjuncts (e.g., one per allowed edge for conjunctions that can contain edges), and not all of them may be necessary. Setting a conjunct to `True` effectively removes it from the conjunction (since $a \wedge True = a$), so this allows the solver to use more or fewer conjuncts as it sees fit.

5.4 Generality of Our Encoding

While our templates may seem restrictive, they can cover a wide variety of logical statements through equivalences. For instance, if an axiom requires a disjunction $a \vee b$ for some a and b, PipeSynth can synthesise this as $\neg a \implies b$. Equation 3 shows an instance of a similar transformation. Meanwhile, if the constraints to be synthesised must take a form like $(a \vee b) \wedge (c \vee d)$, this can be synthesised as two separate axioms (each expressed using one template), one being $(a \vee b)$ and the other being $(c \vee d)$. Note that the user does not need to do these sorts of rewrites manually; PipeSynth would simply synthesise the axiom(s) in a form matching the templates and groups provided to it. Overall, our three built-in templates (and combinations of them) are capable of representing 45 out of 46 of the axioms in our four microarchitectures.

Some axioms conform to a combination of our templates despite not appearing to do so. Consider trying to synthesise a constraint $(a \wedge F_1) \vee (b \wedge F_2)$, where a and b are μspec predicates and F_1 and F_2 are μspec fragments. On the surface, this does not seem to conform to any of our templates. However, if exactly one of a and b is always true, then this constraint is equivalent to two axioms: $(a \implies F_1)$ and $(b \implies F_2)$. In such cases, we refer to the set $\{a, b\}$ as a discriminator. If each of these two axioms corresponds to one of our built-in templates, PipeSynth can synthesise a replacement for the original axiom using a combination of our built-in templates.

We discovered that this pattern occurred in a number of the axioms which at first glance had appeared not to conform to our built-in templates. A common theme in such axioms was the use of {`DataFromInitialStateAtPA(i)`, `SameData(i, j)`} as a discriminator for a load `i` and a store `j`. A given load `i` must either read from the initial state of memory (making `DataFromInitialStateAtPA(i)` true) or must read from some write (in which case `SameData(i,j)` is true for some write `j`). Both cases cannot be true at the same time, making the combination of these predicates a discriminator. Many axioms that did not correspond to one of our built-in templates can be covered by combinations of our templates using discriminators or logical equivalences for the microarchitectures, tests, and execution traces we consider. Figure 9's sourcing axiom is an example of a sourcing axiom that makes use of a discriminator. Specifically, the use of `DataFromInitialStateAtPA` can be seen in Figure 9, while `SameData` appears in the implementation of the `ReadsFrom` macro.

We stress that the user does *not* need to specify the discriminator to get the synthesis to work. They can simply try combinations of various templates mechanically—this can even be done automatically and in parallel. If the axiom in question can be replaced by a combination of our templates, PipeSynth is capable of finding the rewrite and the corresponding synthesis result.

5.5 Generating Execution Traces

PipeSynth requires examples of both forbidden and permitted litmus tests to conduct its synthesis. Additionally, as Section 4.3 discusses, providing permitted tests alone can lead to unrealistic solutions. To solve this problem, we augment permitted litmus tests with timestamped *execution traces* of said tests. Execution traces provide examples of *microarchitectural* event orderings that must be allowed by the synthesised axioms, which litmus tests cannot directly encode. An execution trace can also be thought of as a single μhb graph that must satisfy all axioms (including those synthesised). This use of execution traces is an important difference between PipeSynth and MemSynth [6]. As MemSynth only synthesises ISA-level MCM specifications, it does not need to deal with microarchitectural execution details. PipeSynth, meanwhile, must take microarchitectural details into account in order to generate realistic axioms.

PipeSynth's use of execution traces means that users must provide them to the tool. It is desirable that the generation of such traces be as painless as possible for users. Thankfully, early-stage pre-RTL architectural simulators such as gem5 [5] can be instrumented in a straightforward manner to emit timestamped execution traces of programs. These traces can then be fed into PipeSynth to satisfy the execution trace requirement. Simulators like gem5 could generate diverse traces by taking in a random seed and using that seed to randomise event orderings of nondeterministic simulated microarchitectural events.

Execution traces may come from other sources as well. As an example, the Multi-V-scale processor [32, 47] from our case studies did not have an architectural simulator, but its RTL is publicly available. Thus, we instrumented Multi-V-scale's RTL and ran it through Verilator to generate execution traces for our Multi-V-scale case study. Figure 6 shows an example of such a trace. We stress that our use of RTL to generate Multi-V-scale execution traces is purely due to the absence of an architectural simulator for Multi-V-scale, and is not a fundamental limitation of our approach.

6 AUXILIARY SYNTHESIS OPTIONS

This section describes a number of auxiliary synthesis options that users can leverage to speed up or enhance PipeSynth's synthesis capabilities.

6.1 Removing Group Members or Groups

PipeSynth allows users to remove group members or entire groups from the synthesis encoding if they know that the group members or groups in question do not apply to the given scenario. This reduces the state space that the solver needs to cover and can make synthesis faster. Removal of irrelevant group members or groups can also result in more interpretable synthesis results.

For instance, consider the encoding of groups in Figure 7b. If we consider only the three groups shown, there are 3×3×2 = 18 possibilities for the solver to consider. A user might know that the axiom they wish to synthesise in this particular case does not deal with writes. In such a case, the user could remove the `IsWrite(i)` predicate from the leftmost group in Figure 7b. This would reduce the number of synthesis possibilities to 2×3×2 = 12.

Users can also eliminate entire groups from a particular synthesis instance, further reducing the state space. Consider again Figure 7b's group encoding. A user may know that the axiom they wish to synthesise for this particular case should not involve RMW operations. Thus, they can remove the entire second group in the encoding, as it only contains RMW predicates and `True`. This eliminates the second group's synthesis function, and reduces the number of synthesis possibilities by a factor of 3 (since there were 3 possible solutions to that group's synthesis function).

6.2 Edge Restriction

Just as users can restrict the predicates that are allowed in candidate solutions, they can also restrict the edges that can be used in candidate solutions. Specifically, PipeSynth allows the user to specify the μhb edges that may be referred to by a given conjunction. Architects will often have knowledge about which event orderings certain axioms should enforce, and this feature allows them to restrict synthesised axioms to use relevant μhb edges. In Fig. 7b, the third group restricts the edge to only be Fetch-Decode (or none at all).

Consider synthesising an axiom like that in Figure 3, which represents the in-order nature of the `DX` stage in Multi-V-scale. Architects can deduce that `WB` nodes and edges will be irrelevant to this axiom, as `WB` is later in the pipeline than `DX`. Consequently, for synthesis of this axiom, they can restrict the edges accessible to the synthesis procedure to those between `IF` and `DX` stages.

The restriction of edges accessible to a synthesis procedure reduces the space of possibilities that the solver must search over, which will likely improve synthesis time. It also gives users finer control over the form of the axioms generated, leading to more interpretable axioms. Consider the μhb graph in Figure 2. If an axiom needs to enforce an ordering between an instruction's `IF` stage and its `WB` stage, this can be accomplished by adding a single edge from the `IF` node to the `WB` node or with two edges: one from `IF` to `DX` and one from `DX` to `WB`, as is the case in Figure 2. While both enforce the `IF`-`WB` edge semantically, edge restriction allows for the synthesis of axioms that are better aligned with user intent.

Edge restriction can be iterative. An initial synthesis with PipeSynth may result in axioms which use edges that work, but which the user would prefer not to use. In such cases, the user can then restrict the synthesis to only use edges that they want to use and re-run the synthesis.

In our experiments, we restrict the μhb edges available to synthesis functions to those that are present in the original (i.e., solution) microarchitecture. Thus, in every synthesis query that deals with edges, the solver searches over all μhb edges that are present in the solution microarchitecture. When synthesising a single axiom, we do *not* further restrict the edges available to just those that are present in that axiom in the solution microarchitecture. However, a user could theoretically do so to further reduce the state space.

6.3 Custom Axiom Templates

PipeSynth's three built-in templates (Section 5.2) or a combination of them are sufficient to cover all but one of the axioms in our evaluated microarchitectures. However, in the general case, there will be a few axioms that do not belong to these categories. Expert users may create custom axiom templates for such axioms, as we do for the one axiom that does not fit our templates. PipeSynth provides APIs that allow users to create conjunctions, define the grammar of the conjuncts, and place them as desired in an axiom structure. This generalises our technique to work with axioms beyond the three default templates. If a particular custom template is found to be prevalent across a number of microarchitectures, we can add it to PipeSynth's built-in templates in the future.

This customisation can also be used to improve synthesis runtimes. If the user knows one part of an axiom and wishes to synthesise the rest, they may include the known part of the axiom to remove the need to synthesise it. For instance, a user synthesising a path axiom which only applies to reads may use the following axiom template:

$$\forall i \ (\texttt{IsRead}(i) \wedge P(i)) \implies Q(i)$$

This template hard-codes the known `IsRead` predicate into the structure of the axiom. This guarantees that it is present in the resulting axiom and reduces the synthesis problem to only involve unknown portions of the axiom.

7 METHODOLOGY

7.1 Test Microarchitectures

To evaluate PipeSynth, we used it to try and synthesise axioms from four microarchitectures:

(1) **Multi-V-scale** [32, 47]: A microarchitecture with 3-stage in-order pipelines and an arbiter that only allows one core to access its single-cycle memory at any time.
(2) **FiveStage** [26]: A microarchitecture with a 5-stage pipeline and per-core store buffers.
(3) **FiveStageOoOSLR**: A microarchitecture that has a 5-stage pipeline and per-core store buffers and performs speculative load reordering (SLR). This microarchitecture is very similar to the `gem5.uarch` microarchitecture in the COATCheck public repository [26], though with 2 fewer pipeline stages.
(4) **FiveStagePeekaboo** [26]: A microarchitecture with a 5-stage pipeline and per-core store buffers that exhibits the Peekaboo scenario (with mitigations) [36, 37]. Modelling the Peekaboo scenario requires modelling invalidation-before-use, prefetching, and a livelock-prevention mechanism that allows limited use of stale data. This microarchitecture also

uses the ViCL (Value in Cache Lifetime) abstraction [36] to model cache behaviour.

All of our microarchitectures intend to implement the Total Store Order (TSO) consistency model [38], except for Multi-V-scale, which aims to implement SC. TSO relaxes orderings from writes to subsequent reads. TSO also allows a core to read its own write early, before the write is made visible to other cores.

All but one of the axioms in our tested microarchitectures were covered by our path, FIFO, and sourcing templates, or a combination thereof. The one axiom that required a custom template was the `Reads` axiom from FiveStagePeekaboo.

7.2 Litmus Tests and Execution Traces

We generated our forbidden litmus tests using `litmustestgen` [30]. With a bound of 4 instructions, we generated a set of forbidden litmus tests for all ISA-level axioms in the TSO memory consistency model. The tool generated 18 forbidden litmus tests. For Multi-V-scale, we removed the 4 litmus tests which involved RMW operations, as Multi-V-scale does not support RMWs.

For each of our forbidden litmus tests, we removed an ISA-level coherence edge or changed the data value of a memory read in order to create a permitted litmus test. For all microarchitectures except Multi-V-scale, we generated execution traces by taking the complete μspec for them, running the permitted litmus tests on the complete μspec, and using the acyclic μhb graphs generated for those tests as execution traces that the synthesis needed to obey. This produces 18 execution traces for each of those microarchitectures.

For Multi-V-scale, meanwhile, we used an RTL simulator to generate execution traces. We instrumented the RTL of Multi-V-scale to print out events of interest when they occurred in an execution. We then ran Verilator on the RTL to obtain an executable which could be invoked with a binary file containing a sequence of instructions for each core. By using binaries that represented litmus tests, we were thus able to simulate litmus tests on the processor. The output of these simulations on the instrumented RTL gave us execution traces like the one in Figure 6.

7.3 Single and Multi-Axiom Synthesis Flows

When evaluating the synthesis time for a single axiom, we remove it from the microarchitecture and replace it with one or more templates corresponding to that axiom's structure. We then attempt to synthesise that axiom with PipeSynth using the specified template(s). In other words, we try and synthesise each axiom independently when given all others. Our single-axiom synthesis flow can be described as follows for a μspec microarchitecture M:

```
For each axiom ax in M:
  Remove ax from M to get M'
  M' = M' + templates(ax)
  Synthesise ax using M'
```

For multi-axiom synthesis, we iteratively remove axioms from the microarchitecture and try and synthesise replacements for all axioms removed so far. When removing axioms, we do not immediately replace them with templates, as sometimes an additional template is not necessary. For instance, if two FIFO axioms are removed, PipeSynth may be able to find a single axiom that can replace both of them from just one FIFO template. If indeed an axiom does require its own template, we add the same template that we used when synthesising the axiom in the single-axiom case. Our multi-axiom synthesis flow can be described as follows for a μspec microarchitecture M:

```
For each axiom ax in M:
  Remove ax from M
  retVal = synthesise removed axioms using M
  if retVal == FAILED:
    Add template(s) for ax to M
    retVal = Retry synthesis using M
    if retVal == TIMEOUT:
      break
```

In other words, we try and synthesise as many axioms as we can at the same time. We remove axioms one at a time in the following order: first, axioms that proved unnecessary in the single-axiom synthesis (as they can be synthesised trivially), then FIFO axioms, then path axioms, then sourcing axioms, and then any others. Within each of these sets, we remove axioms in ascending order of their single-axiom synthesis runtimes (i.e., the axiom that was synthesised fastest is removed first). Section 8 contains the results of these experiments.

7.4 Experimental Parameters

PipeSynth produces a SyGuS file for each of our synthesis queries. We ran PipeSynth using Python 3.8. We used CVC5 version 0.0.4 to generate our SyGuS files, and we used CVC5 version 1.0.2 to solve those SyGuS files[4]. We allowed each synthesis query to run for a maximum of two hours. The experiments were performed on nodes of a 429-node cluster. Each synthesis query was given 16GB of RAM.

The conjuncts for our synthesis queries contained all μspec predicates used in the corresponding microarchitectures, with the exception of predicates related to fence operations (e.g., `IsFence`). This is because no fence operations were present in the litmus tests we used that were generated by `litmustestgen` [30]. Thus, fence-related predicates were irrelevant to our synthesis, and we excluded them from our synthesis possibilities. We do, however, synthesise axioms dealing with RMW instructions for microarchitectures other than Multi-V-scale. (Multi-V-scale does not support RMWs.)

8 RESULTS

8.1 Single-Axiom Synthesis Results

Figure 10 depicts our single-axiom synthesis results. It shows the number of axioms synthesised by PipeSynth over time, separated by microarchitecture. Each synthesis query is assumed to be run separately and in parallel, so for instance, the last axiom for FiveStagePeekaboo at the top of the graph takes just over 45 minutes (its absolute x-value) rather than about 4 minutes (the distance from its x-value to the previous x-value on the green line). PipeSynth was able to synthesise replacements for 42 out of the 46 axioms in our test microarchitectures, within 2 hours each. Table 1 provides a breakdown of the axioms across our microarchitectures by axiom template. It

[4]CVC5-1.0.2 contains a bugfix for a soundness error in its SyGuS solving, which was a key reason behind us using it to solve SyGuS files.

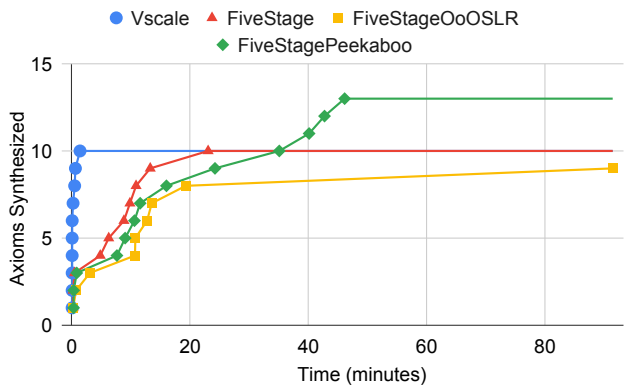

Figure 10: The number of successful syntheses of individual axioms against time, separated by microarchitecture. Each synthesis query is assumed to be run separately and in parallel.

Table 1: A breakdown of our single-axiom synthesis success/-failure results by template and microarchitecture, not including axioms pertaining to fences. ("Comb." = Combination.)

Template	Vscale	FiveStage	OoOSLR	Pkboo.
Path	2/2	0/0	0/0	1/1
FIFO	7/7	7/7	6/6	10/10
Sourcing	0/0	1/1	1/1	1/1
Comb.	1/1	1/2	1/2	0/1
Other	0/0	0/0	0/0	0/1

also lists how many axioms of each type in each microarchitecture succeeded in our single-axiom synthesis. (Combinations of our templates are as defined in Section 5.4.) Table 1 does not include 3 axioms in our microarchitectures that pertain to fence instructions. Our tests did not contain fence instructions, so these axioms can always evaluate to `True` no matter what template is used. Thus, we used an empty template containing no synthesis functions for these axioms.

Our synthesised axioms were often more verbose than necessary. A post-processing algorithm can remove their redundant conjuncts. Also, some synthesised path axioms had fewer conjuncts than expected. PipeSynth found 14 of the axioms to be unnecessary to satisfy the litmus tests and execution traces. In each of these cases, PipeSynth synthesised an empty axiom, with all conjuncts as `True`. Due to the simplicity of synthesising an empty axiom, these 14 axioms were each synthesised in under 1 minute. 5 of these axioms proved unnecessary because they were designed to implement features not seen in our litmus tests, including fence operations and postconditions on memory values. 2 other axioms that proved unnecessary are the path axioms for reads and writes in Multi-V-scale. These axioms are not necessary because the remaining axioms guarantee a total order on the `DX` stages and `WB` stages of all instructions in a test, as well as any necessary read value requirements. The other 7 axioms that proved unnecessary were FIFO axioms that added "convenience" edges which were not necessary to produce a cycle in any of our forbidden litmus tests. For instance, `STB_FIFO` in FiveStage was synthesised as an empty axiom because it adds edges between `StoreBuffer` stages, which (despite being a valid ordering) turn out not to contribute to μhb cycles that make the forbidden tests unobservable. This instance shows how PipeSynth can potentially be used in the future to optimise μspec specifications by reducing their size.

Table 2: Synthesis time (in minutes and seconds) versus the number of axioms removed from each microarchitecture. Blue entries correspond to trivial synthesis results. (T/O = Timeout.)

#	Vscale	FiveStage	OoOSLR	Peekaboo
1	0m3s	0m12s	0m12s	0m17s
2	0m3s	0m12s	0m12s	0m17s
3	0m3s	0m10s	15m24s	0m16s
4	0m7s	12m3s	19m48	27m4s
5	0m7s	9m13s	14m22s	47m30s
6	0m6s	21m5s	98m20s	52m49s
7	1m1s	16m13s	T/O	106m58s
8	1m52s	T/O	T/O	69m56s
9	2m8s	T/O	T/O	90m41s
10	58m44s	T/O	T/O	T/O

We were able to individually synthesise all axioms for Multi-V-scale, with the longest individual synthesis time being 83 seconds. We failed to synthesise the `Reads` axiom from the remaining architectures and the `L1ViCLs` axiom from FiveStagePeekaboo within 2 hours each. These axioms each involved 4 or more quantifiers, making them rather complex, so it is not extremely surprising that we were unable to synthesise them. Three of these axioms can be replaced by combinations of our templates and 1 (`Reads` from FiveStagePeekaboo) did not fit our templates. We will investigate feasible synthesis of such axioms in future work.

8.2 Multi-Axiom Synthesis Results

To evaluate the performance of PipeSynth when synthesising multiple axioms simultaneously, we progressively removed axioms from each microarchitecture as outlined in Section 7.3. Table 2 shows the time of each such synthesis against the number of axioms removed for each microarchitecture. Trivial synthesis results (with all conjuncts as `True`) are highlighted in blue. For Multi-V-scale, PipeSynth found a suitable synthesis result for each query, including when all 10 axioms were removed. In other words, we were able to generate a complete microarchitectural specification for Multi-V-scale that satisfies the litmus tests and execution traces in under 1 hour. For FiveStage, FiveStageOoOSLR and FiveStagePeekaboo, we were able to simultaneously synthesise 7 (4 non-trivial), 6 (4 non-trivial), and 9 (6 non-trivial) axioms respectively within our 2 hour time limit. We believe that our results bode well for using PipeSynth to synthesise larger portions of microarchitectural specifications in the future.

Our multi-axiom synthesis times generally increase as we remove axioms from each microarchitecture, as the synthesis functions must replace more of the specification. However, they do not monotonically increase with respect to the number of axioms removed. This is because the removal of axioms can lead to a more flexible synthesis query where synthesis solutions do not need

to uphold the exact orderings of the original microarchitecture, potentially enabling faster discovery of a solution.

9 RELATED WORK

There has been much work on formal MCM specification and verification in recent years. Researchers have developed formal MCM specifications for many commercial ISAs by hand, including x86 [38], Power [2, 31, 48], ARMv8 [43], and RISC-V [46]. There has also been work on formal specifications of GPU MCMs [1, 28, 56].

For formal MCM verification of hardware implementations against their ISA MCMs, there are two main lines of work. The first is the automated Check suite [27, 29, 35, 36, 54] and tools based on it [34, 60]. The second line of work is Kami [7, 55], which conducts more manual verification using the Coq proof assistant. These tools take formal specifications as input, and cannot generate such specifications by themselves. In contrast, PipeSynth enables users to automatically generate formal specifications from litmus tests and execution traces.

Program synthesis is the automatic generation of programs matching a high-level correctness specification. In recent decades, notable variants of program synthesis include counterexample-guided synthesis (CEGIS) [50], oracle-guided synthesis (OGIS) [21], and syntax-guided synthesis (SyGuS) [3]. Popular synthesis frameworks include Sketch [49], PROSE [42], and Rosette [52]. More recently, researchers have proposed Semantics-Guided Synthesis (SemGuS) [22] and Synthesis Modulo Oracles (SyMO) [40] as alternative synthesis methods. We refer the reader to a recent survey on program synthesis [16] for a more detailed overview of the area. PipeSynth utilises SyGuS for its synthesis. However, it is well-placed to use techniques like OGIS and SyMO in the future. For instance, PipeSynth can use execution traces from simulators, which can thus function as oracles (Section 5.5).

MemSynth [6] takes in a set of allowed and forbidden litmus tests plus a sketch (i.e., template) of the overall structure of an ISA-level MCM specification, and synthesises a complete ISA-level specification matching the tests and sketch. Like MemSynth, PipeSynth also synthesises formal specifications related to MCMs using litmus tests. However, PipeSynth synthesises microarchitectural ordering axioms (as opposed to ISA-level MCMs). PipeSynth also requires microarchitectural execution traces in addition to litmus tests for its synthesis, as litmus test outcomes do not convey any specific information about the microarchitectural orderings of events that need to be maintained by its synthesised axioms. (See also Section 4.3.) In addition, while MemSynth's synthesis is useful only when creating or modifying an ISA, PipeSynth's synthesis is potentially useful in the development of any parallel microarchitecture, even if its ISA MCM was formalised long ago.

Prior work on synthesising formal specifications for hardware includes RTL2μspec [19] and the ILA line of work [51, 58, 59]. Meanwhile, Godbole et al. [13] convert μspec axioms to operational models (which are compilable to RTL). Both RTL2μspec and the ILA line of work are capable of generating microarchitectural specifications, with RTL2μspec generating μspec axioms like PipeSynth does. However, both of these approaches use an RTL implementation of the hardware to accomplish their synthesis, which means that they can only be used after the processor has been implemented. Formal verification can be much more effective when it is used before RTL is written, as doing so catches bugs earlier and can reduce overall development time [33]. PipeSynth enables pre-RTL synthesis of microarchitectural specifications, enabling users to leverage the benefits of early-stage design time verification without writing entire specifications by hand.

10 CONCLUSION

Formal methods can provide the strong correctness guarantees we require from our processors today. However, most automated formal verification approaches require formal specifications of the system being verified. These specifications are typically written by hand, which is tedious and error-prone. In addition, most architects and hardware engineers today do not have formal methods expertise, making it difficult for them to write formal specifications for their processors. Prior work on automated specification synthesis either does not apply to microarchitecture or requires an RTL implementation. This hampers the use of early-stage pre-RTL formal verification approaches, which can catch bugs earlier and reduce verification overhead and development time.

In response, we present PipeSynth, an automated formal methodology and tool for the pre-RTL formal synthesis of microarchitectural ordering axioms in the domain-specific μspec language. Given a partial μspec ordering specification, litmus tests, and observable execution traces, PipeSynth can automatically synthesise additional μspec ordering axioms that are necessary to guarantee correctness for the provided tests and traces. PipeSynth thus helps architects and hardware engineers automatically generate formal ordering specifications for their microarchitectures even if they do not have formal methods expertise. In doing so, it will enable the increased use of formal verification for emerging microarchitectures. This will help architects catch bugs faster and will substantially improve the correctness of the processors of tomorrow.

ACKNOWLEDGEMENTS

We thank Yan-Ru Jhou for help running benchmarks and fixing issues with our code infrastructure. We thank Andrew Bogdan for implementation effort on an early version of PipeSynth. This work was supported in part by Intel Corporation.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact[5] contains the code, litmus tests, and execution traces for PipeSynth. PipeSynth can synthesise μspec microarchitectural ordering axioms to be added to a partial μspec specification so that the overall specification respects the constraints of provided litmus tests and execution traces.

A.2 Artifact Check-List (Meta-Information)

- **Run-time environment: PipeSynth requires Cython, scikit-build, pytest, toml, CVC5-0.0.4 python bindings, and the CVC5-1.0.2 binary. We have tested PipeSynth on macOS Ventura and Monterey.**

[5] Official artifact DOI: https://doi.org/10.5281/zenodo.7592848.

- **Metrics: Execution time is our main metric. When multiple axioms are synthesised, another metric is also the number of axioms we were able to synthesise.**
- **Output: Our output is a log file consisting of the runtime of each experiment and the synthesis result, if successful.**
- **Experiments: The procedure for running our experiments is detailed in the Experiment Workflow section.**
- **How much disk space required (approximately)?: 2GB**
- **How much time is needed to prepare workflow (approximately)?: 15 minutes**
- **How much time is needed to complete experiments (approximately)?: 48 hours**
- **Publicly available?: Yes, our code is available at https://github.com/chasenorman/PipeSynth-AEC**
- **Code licenses (if publicly available)? MIT License**
- **Data licenses (if publicly available)? N/A**

A.3 Description

A.3.1 How to Access. PipeSynth is available at https://github.com/chasenorman/PipeSynth-AEC. Our official artifact DOI is https://doi.org/10.5281/zenodo.7592848.

A.3.2 Software Dependencies. PipeSynth requires Python 3.8, Cython, scikit-build, pytest, toml, Python bindings for CVC5-0.0.4, and the binary for CVC5-1.0.2.

A.4 Installation

A.4.1 Setup Python3.8 Required Build Libraries. Our artifact uses the CVC5 solver's Python APIs to generate SyGuS-IF files. The specific Python version we use is Python3.8. PipeSynth also requires the following Python libraries:

- Cython
- scikit-build
- pytest
- toml

Installation of the packages above can be accomplished by pip:

```
pip install Cython
pip install scikit-build
pip install pytest
pip install toml
```

A.4.2 Setup the CVC5 Execution Environment. Our artifact currently requires 2 versions of CVC5, one for generating SyGuS files and one for solving them. CVC5-0.0.4 along with its Python-Binding API is used to generate SyGuS-IF files, while CVC5-1.0.2 is used to solve these generated SyGuS-IF files. It is not completely straightforward for us to use just one version of CVC5. The Python APIs available in CVC5-0.0.4 (which we use for generating SyGuS-IF files) are not the same as those in CVC5-1.0.2. On the other hand, CVC5-1.0.2 contains a bugfix for a soundness error in SyGuS solving, so we can't use the old version for the purpose of solving. We are currently working on updating our codebase to just use one CVC5 version in an effort to make it easier to use, and will update our GitHub repository when this change is ready.

Run the following commands, all of which should be executed under the project folder. First, install CVC5-0.0.4 for Python API Bindings:

- Download CVC5-0.0.4 source code
- Build CVC5-0.0.4 and Python-Binding from source code with the following terminal commands

```
./configure.sh --python-bindings
    --auto-download
cd build
make
sudo make install
sudo cp -r ./build/lib/*
    $( python -c 'import sysconfig;
    print(sysconfig.get_paths()["purelib"])')
```

Next, install the CVC5-1.0.2 binary for SyGuS solving. **If you are on Linux, change** `cvc5-macOS` **in the command below to** `cvc5-Linux`. **If you are on an ARM Mac, change** `cvc5-macOS` **in the command below to** `cvc5-macOS-arm64`**.**

```
wget -O cvc5  https://github.com/cvc5/cvc5/
    releases/download/cvc5-1.0.2/cvc5-macOS
```

Place this binary in the project root directory such that `./cvc5` **executes the binary.**

A.4.3 Basic Test. The following terminal commands, executed in the PipeSynth root directory, should produce a file that is named `out/vscale-pofetch-fifo.sy` and then run CVC5 on the file. The synthesis result and runtime of CVC5 will be written to a file called `out/results.txt`.

```
python3 main.py Vscale pofetch fifo
./benchmark
```

Synthesis should take no more than 2 minutes. CVC5 should not report that the query is infeasible, nor should CVC5 crash. To reset to a clean slate, empty the `out` directory.

A.5 Experiment Workflow

A.5.1 Single-Axiom Synthesis. Figure 10 in our paper shows our single-axiom synthesis runtimes for four microarchitectures (Vscale, FiveStage, FiveStageOoOSLR, and FiveStagePeekaboo). Table 1 shows how many axioms of each type we were successfully able to synthesise during our single-axiom synthesis. These results can be reproduced using the following procedure.

The following commands generate SyGuS files for the synthesis of each of the axioms in our four microarchitectures. Each SyGuS file attempts to synthesise one axiom in a microarchitecture, given all other axioms and litmus tests and execution traces.

```
python3 main.py Vscale all
python3 main.py FiveStage all
python3 main.py FiveStageOoOSLR all
python3 main.py FiveStagePeekaboo all
```

The above commands will generate the SyGuS files for single-axiom synthesis and put them in the `out/` directory. Next, use CVC5 to attempt synthesis of the queries in the files with our benchmarking script:

```
./benchmark
```

This script will run CVC5 on each SyGuS file in the `out/` directory, with a time limit of 2 hours each. The synthesis results and runtimes will be found in the `out/results.txt` file.

A.5.2 Multi-Axiom Synthesis. Table 2 details our multi-axiom synthesis results for our four microarchitectures. To reproduce our multi-axiom synthesis results, please follow the procedure below for each microarchitecture. For demonstration, we use `FiveStage0o0SLR`. We will remove axioms from `FiveStage0o0SLR` in the order specified by `axioms.txt`, which corresponds to the order in which we removed axioms when generating Table 2 for each microarchitecture.

Our multi-axiom synthesis flow removes axioms iteratively from a microarchitecture and only adds synthesis templates for them if necessary. (See Section 7.3 in our paper for details.) The first axiom we removed from `FiveStage0o0SLR` was `stbfifo`. To generate and run CVC5 on a SyGuS file corresponding to the removal of `stbfifo` from `FiveStage0o0SLR`, do the following:

```
python3 main.py FiveStage0o0SLR stbfifo
time ./cvc5 out/FiveStage0o0SLR-stbfifo.sy
```

CVC5 should be successful here, indicating that additional templates are not necessary at this point. Continue by removing both the first and second axioms, namely `stbfifo` and `mfence`:

```
python3 main.py FiveStage0o0SLR
    stbfifo mfence
time ./cvc5 out/FiveStage0o0SLR-
    stbfifo-mfence.sy
```

Continue until a synthesis failure. In this case, we expect the first failure to occur after the removal of the third axiom, `pofetch`, due to an insufficient variety of templates in the SyGuS file to allow the synthesis to succeed:

```
python3 main.py FiveStage0o0SLR
    stbfifo mfence pofetch
time ./cvc5 out/FiveStage0o0SLR-
    stbfifo-mfence-pofetch.sy
```

On synthesis failure, add the names of the template(s) corresponding to the last axiom removed as arguments. The templates are specified by `axioms.txt`. In this case, the template for the `pofetch` axiom is `fifo`.

```
python3 main.py FiveStage0o0SLR
    stbfifo mfence pofetch fifo
time ./cvc5 out/FiveStage0o0SLR-
    stbfifo-mfence-pofetch-fifo.sy
```

Continue to remove axioms (in the order specified by `axioms.txt`) and re-run CVC5, adding templates corresponding to the last removed axiom where necessary. Stop when the synthesis times out (i.e., takes longer than 2 hours).

A.6 Evaluation and Expected Results

For our single and multi-axiom synthesis cases, we expect our synthesis results (synthesis success/timeout) to correspond to those in Tables 1 and 2 respectively. We expect our runtimes for single and multi-axiom synthesis to correspond to those in Figure 10 and Table 2 respectively, making allowances for any differences in performance between the machines we used and those you use, and for any nondeterminism in CVC5's operation.

The runtimes for single-axiom synthesis runs can be directly read from the `results.txt` file. For multi-axiom synthesis, they will be output as a result of the `time` command for successful synthesis queries, or the timeout will be reached.

A.7 Experiment Customization

Any set of axioms can be removed from architecture `<arch>` and replaced with a set of templates by following this form:

```
python3 main.py <arch> <axiom_1> ... <axiom_n>
    <template_1> ... <template_k>
```

REFERENCES

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591. ACM, 2015.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36, July 2014.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[4] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS*, 2022.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comp. Arch. News*, 39(2), August 2011.

[6] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[7] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), 2017.

[8] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, power, arm, and c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 211–225. Association for Computing Machinery, 2018.

[9] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *CAV*, 2019.

[10] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 608–621, 2016.

[11] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris)*, pages 429–442, January 2017.

[12] Harry D. Foster. Trends in functional verification: A 2014 industry study. *52nd Design Automation Conference (DAC)*, 2015.

[13] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. Automated conversion of axiomatic to operational models: Theory and practice. In *22nd Conference on Formal Methods in Computer-Aided Design (FMCAD)*, page 331, 2022.

[14] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 635–646, 2015.

[15] Martonosi Research Group. Check research tools and papers website, 2017. http://check.cs.princeton.edu.

[16] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[17] Mark 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.

[18] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *CoRR*, abs/1907.02064, 2019.

[19] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 679–694, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. ILAng: A modeling and verification platform for SoCs using instruction-level abstractions. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–357, Cham, 2019. Springer International Publishing.

[21] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[22] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL), Jan 2021.

[23] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[24] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 28(9):690–691, 1979.

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018.

[26] Daniel Lustig. Coatcheck, 2016. https://github.com/daniellustig/coatcheck.

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

[28] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 257–270. Association for Computing Machinery, 2019.

[29] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek 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.

[30] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 661–675. ACM, 2017.

[31] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *24th International Conference on Computer Aided Verification (CAV)*, 2012.

[32] Yatin A. Manerkar. Vscale.uarch, 2017. https://github.com/ymanerka/rtlcheck/blob/master/uarches/Vscale.uarch.

[33] Yatin A. Manerkar. *Progressive Automated Formal Verification of Memory Consistency in Parallel Processors.* PhD thesis, Princeton University, Princeton, NJ, USA, 2020.

[34] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated memory consistency proofs for microarchitectural specifications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 788–801. IEEE Press, 2018.

[35] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. In *50th International Symposium on Microarchitecture (MICRO)*, 2017.

[36] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. In *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[37] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

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

[39] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The SyGuS language standard version 2.1, 2021. https://sygus.org/assets/pdf/SyGuS-IF_2.1.pdf.

[40] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. Satisfiability and synthesis modulo oracles. In *Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings*, page 263–284, Berlin, Heidelberg, 2022. Springer-Verlag.

[41] Elizabeth Polgreen and Sanjit A. Seshia. Synrg: Syntax guided synthesis of invariants with alternating quantifiers. *CoRR*, abs/2007.10519, 2020.

[42] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 107–126, New York, NY, USA, 2015. Association for Computing Machinery.

[43] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In *45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2018.

[44] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: Attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 685–698, New York, NY, USA, 2022.

[45] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 42–58, Cham, 2016. Springer International Publishing.

[46] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20190608-Base-Ratified*, June 2019.

[47] RISC-V Foundation and Yatin A. Manerkar. Multi-v-scale, 2017. https://github.com/ymanerka/multi_vscale/tree/multicore.

[48] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER microprocessors. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[49] Armando Solar-Lezama. *Program Synthesis by Sketching.* PhD thesis, University of California, Berkeley, USA, 2008. AAI3353225.

[50] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News*, 34(5):404–415, oct 2006.

[51] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based synthesis of instruction-level abstractions for SoC verification. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 160–167, Austin, TX, 2015. FMCAD Inc.

[52] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[53] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960, 2018.

[54] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret 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.

[55] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In *27th International Conference on Computer Aided Verification (CAV)*, 2015.

[56] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-scope promotion: Clarified, rectified, and verified. *SIGPLAN Not.*, 50(10):731–747, October 2015.

[57] Yue Xing, Huaixi Lu, Aarti Gupta, and Sharad Malik. Leveraging processor modeling and verification for general hardware modules. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1130–1135, 2021.

[58] Yu Zeng, Aarti Gupta, and Sharad Malik. Automatic generation of architecture-level models from rtl designs for processors and accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 460–465, 2022.

[59] Yu Zeng, Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. Generating architecture-level abstractions from rtl designs for processors and accelerators part I: Determining architectural state variables. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[60] Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. ILA-MCM: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.

Received 2022-10-20; accepted 2023-01-19