Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations

Yao Hsiao
Stanford University
yaohsiao@stanford.edu

Dominic P. Mulligan
Arm Research
dominic.mulligan@arm.com

Nikos Nikoleris
Arm Research
nikos.nikoleris@arm.com

Gustavo Petri
Arm Research
gustavo.petri@arm.com

Caroline Trippel
Stanford University
trippel@stanford.edu

ABSTRACT

Modern hardware complexity makes it challenging to determine if a given microarchitecture adheres to a particular memory consistency model (MCM). This observation inspired the *Check* tools, which formally check that a specific microarchitecture correctly implements an MCM with respect to a suite of litmus test programs. Unfortunately, despite their effectiveness and efficiency, the Check tools must be supplied a microarchitecture in the guise of a *manually constructed* axiomatic specification, called a μSPEC model.

To facilitate MCM verification—and enable the Check tools to consume processor RTL directly—we introduce a methodology and associated tool, RTL2μSPEC, for automatically synthesizing μSPEC models from processor designs written in Verilog or SystemVerilog, with the help of modest user-provided design metadata. As a case study, we use RTL2μSPEC to facilitate the Check-based verification of the four-core RISC-V V-scale (multi-V-scale) processor's MCM implementation. We show that RTL2μSPEC can synthesize a complete, and *proven correct* by construction, μSPEC model from the SystemVerilog design of the multi-V-scale processor in 6.84 minutes. Subsequent Check-based MCM verification of the synthesized μSPEC model takes less than one second per litmus test.

CCS CONCEPTS

• **Hardware** → **Functional verification**; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

memory consistency, verification, concurrency, shared memory

ACM Reference Format:
Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. 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), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3466752.3480087

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480087

1 INTRODUCTION

Memory Consistency Models. In a multicore setting, multiple hardware threads concurrently execute while modifying a shared memory. A *memory consistency model* (MCM) is thus required, which describes *who* sees *what* and *when*—that is, the particular order(s) in which writes to this shared memory may become observable to different threads. MCMs are described using rules which restrict the ordering and visibility of shared memory accesses—either informally using natural language or formally [4, 45, 47]—with different architectures exhibiting different MCMs [5–7, 20, 21, 44].

Notably, a sound high-level programming language MCM is not sufficient to ensure correct execution of a parallel program. In particular, a program is only guaranteed to run correctly if a compiler correctly translates language-level MCM primitives to assembly instructions, and if the target microarchitecture is indeed implementing the MCM specified by its instruction set architecture (ISA). Despite the importance of correct hardware MCM implementations, a scalable, efficient, sound, and complete methodology for verifying processor MCM implementations remains elusive.

Verification of Hardware Memory Models. Formal verification of hardware MCM implementations is challenging for a variety of reasons. For example, ISA MCM correctness properties are generally articulated as ordering and visibility constraints on assembly instructions. Deducing whether or not they hold for a particular microarchitecture thus requires mapping these instruction-level properties to RTL-level assertions, such as SystemVerilog Assertions [43] (SVAs). These SVAs can then be proven or refuted by off-the-shelf RTL property verification tools, many of which are based on *model checking* [9, 14]. Not only is defining these assertions tedious and error prone, but checking that they hold of a design is extremely computationally intensive. Thus, it is common for assertions to be decomposed and/or for the hardware design itself to undergo abstraction for assertion checking to terminate.

These challenges have lead researchers to pursue other means of evaluating the adherence of a processor implementation to its MCM specification. *Litmus tests* [3, 27]—small concurrent programs that are carefully crafted, or automatically generated [11, 26, 45], to encode the implications of a given MCM on observable program outcomes—are a popular approach. They have been used for both *post hoc* formal specification of observable hardware behavior and for testing of hardware implementations against a particular MCM [13, 15, 17, 18, 28, 35, 38, 39]. For example, tools have been developed for running litmus tests on hardware with varied timings, interleavings, and system load imposed by a test harness, in order to coax out bugs in the hardware MCM implementation [3, 34, 40]. While sound, this approach is incomplete for failing to *prove* hardware will *always* execute litmus tests correctly even if no bug is found during validation testing.

The Check Tools. Building on litmus test-based *testing* approaches, prior work introduced the *Check* family of tools [24, 25, 31, 33], which incorporate *formal rigor*. Specifically, the Check tools provide an efficient mechanism for *proving* that a microarchitecture's MCM implementation is correct with respect to a suite of litmus test programs. Remarkably, recent work has shown that this approach can even be extended to prove correctness over the space of *all* programs [30].

Despite their success in finding bugs in real hardware, the Check tools possess a limitation: they require as input *manually-constructed* formal specifications of hardware designs, called *μSPEC models*, rather than Verilog implementations. A μSPEC model is an axiomatic model of a microarchitecture expressed in a DSL called μSPEC—essentially a specific theory, or collection of function and predicate symbols, in first-order logic. A gap therefore remains between the μSPEC models that support efficient Check-based verification and the RTL that hardware designers write and know.

The RTL2μSPEC Approach and Tool. In this paper, we pursue a new approach to scalable, Check-based verification of hardware MCMs by *automatically* synthesizing μSPEC models directly from user-supplied RTL written in Verilog or SystemVerilog,[1] with the help of modest user-provided design metadata (§4.2.1 and §4.3.4). We introduce the RTL2μSPEC tool,[2] which takes a Verilog processor design as input, and produces a complete μSPEC model as output, which can serve as input into any of the Check MCM verification tools. In designing RTL2μSPEC, our most fundamental challenge is bridging the inherently *operational* character of Verilog with the *axiomatic* specification style of μSPEC—the latter of which consists of axioms describing *happens-before invariants* (HBIs). HBIs capture causal *happens-before* relationships between hardware events that are preserved by a particular Verilog design for every executing program.

We bridge the *operational-axiomatic gap* with **our first insight**—μSPEC models can be decomposed into several categories of HBIs, with the two most general classifications being *intra-instruction HBIs* versus *inter-instruction HBIs*. Intra-instruction HBIs describe happens-before orderings that are localized to a single instruction's execution on a microarchitecture. Inter-instruction HBIs describe happens-before orderings relating the execution of a pair of instructions. This HBI decomposition (§3) ensures completeness of the RTL2μSPEC synthesis procedure. In other words, identifying the HBI building blocks of a complete μSPEC model is the first step in automatically synthesizing one.

Our second insight, which enables RTL2μSPEC to synthesize a complete set of HBIs from a Verilog design with minimal designer input, is that a control-flow dataflow graph (CDFG) representation of a Verilog design (i.e., a netlist) contains a subset of the target HBIs, which can be further used to construct *HBI hypotheses* for the remaining set of HBIs to be extracted. These hypotheses constitute an over-approximation of *all* HBIs implied by the Verilog design, and can be encoded as SVAs and evaluated with formal RTL property verification tools [12] to either prove or refute them.

Our third insight, which leads to RTL2μSPEC's efficiency over previous approaches, is a reliance on proving simple and localized HBIs when incrementally constructing the μSPEC model. In our case study (§5), RTL2μSPEC automatically generates and evaluates 122 SVAs when synthesizing a μSPEC model from a four-core version of the RISC-V V-scale (multi-V-scale) processor [29, 31]. Remarkably, each assertion is either proven or refuted in seconds—3.34 seconds on average. In contrast, prior work that aims to identify inaccuracies in hand-written μSPEC with respect to a Verilog design times out after 11 hours of runtime when evaluating the same microarchitecture [31]. We attribute this difference in verification time to the difference in assertion complexity between the two approaches.

Contributions. In this paper we make three major contributions:

(1) *The decomposition of μSPEC models into fundamental HBI building blocks*: We observe that μSPEC models can be decomposed into a collection of intra- and inter-instruction HBIs. Further, inter-instruction HBIs can be classified as resulting from either structural or dataflow dependencies between instructions during their execution on a microarchitecture. This decomposition facilitates a systematic procedure for synthesizing HBIs, and thus μSPEC models, directly from RTL. In summary, we are the first to define what constitutes a complete μSPEC model for a given RTL design.
(2) *The RTL2μSPEC tool for synthesizing complete, and proven correct by construction, μSPEC models from RTL*: RTL2μSPEC takes a processor design written in Verilog as input and outputs a μSPEC model by synthesizing all relevant HBIs. In doing so, RTL2μSPEC exhibits 100% proof coverage on the compliance of RTL to synthesized μSPEC model, advancing the state-of-the-art [31]. The resulting μSPEC model can serve as input to any of the Check MCM verification tools [24, 25, 30, 31, 33, 41, 42].
(3) *The verification of the RISC-V multi-V-scale MCM implementation:* We use RTL2μSPEC to facilitate the Check-based verification of the multi-V-scale processor [29, 31], *rooted in RTL*. In doing so, we identify a new bug in the RISC-V V-scale microarchitecture, and thus the multi-V-scale, that allows invalid instructions to update memory, and which was missed by prior work. RTL2μSPEC synthesizes a complete μSPEC model from the multi-V-scale design in 6.84 minutes. Subsequent Check-based MCM verification using the μSPEC model takes less than one second per litmus test to *prove* MCM compliance (with respect to said test).

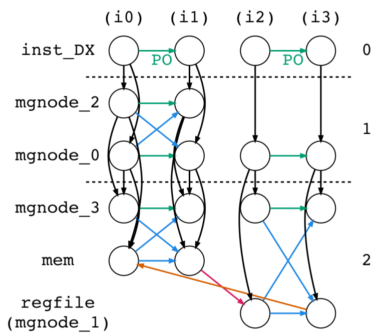
2 BACKGROUND

Encoding Ordering Behaviors with Litmus Tests. Simply put, MCMs specify the values that can be legally returned by shared memory loads in a concurrent program via constraints on the ordering and visibility of shared memory accesses. MCMs are a fundamental component of a processor's ISA specification, and the ability of a

[1]While RTL2μSPEC can accept either Verilog or SystemVerilog designs as input, we frequently refer to RTL2μSPEC's processor input as a *Verilog* design for brevity.
[2]RTL2μSPEC is open source and publically available at https://github.com/yaohsiaopid/rtl2uspec.

Core 0	Core 1
(i0) sw x, 1	(i2) lw r1, y
(i1) sw y, 1	(i3) lw r2, x
SC forbids r1 = 1, r2 = 0	

(a) Message passing (MP) litmus test with forbidden non-SC outcome. Memory locations are initialized to 0.



(b) μhb graph execution of the MP litmus test in (a) on the RISC-V multi-V-scale [29, 31] (Fig. 3a), corresponding to the non-SC outcome. The cycle signifies that this execution is *unobservable.*

Figure 1: A μhb graph, as in (b), can be used to represent the hardware specific execution of a litmus test program, as in (a). (b)'s μhb graph was generated by COATCheck [25] using an RTL2μSPEC-synthesized μSPEC model of the RISC-V multi-V-scale [29, 31]. mgnode_n row labels represent groups of state elements there were merged in the RTL2μSPEC-synthesized μSPEC model due to exhibiting the same ordering behaviors (see §4.4).

microarchitecture to correctly execute a program relies crucially on the correctness of its MCM implementation.

Prior work has proposed a number of tools for evaluating the correctness of hardware MCMs [15, 18, 28, 35, 36, 38, 39]. Litmus test programs [3, 27]—small programs designed to demonstrate constraints on shared memory ordering and visibility that are imposed by a given MCM—are central to this. They are used to concisely articulate the legal ordering behaviors of concurrent programs on hardware implementing a particular MCM.

Fig. 1a gives an example of a litmus test program, commonly called the *message passing* (MP) test. Here, Core 0 writes some data x before setting a flag y, while Core 1 reads the flag y before reading the data x. In keeping with typical litmus test convention, all memory locations are initialized to 0 (i.e., x=0 and y=0). The *outcome* of a litmus test program denotes the values returned by the loads of the test—in this test, featuring two loads, there are four possible outcomes. The loads on Core 1 can return either the initial values of x and y (0s), or the values written by Core 0 (1s). For Sequential Consistency (SC) [23]—which requires that each legal program outcome must correspond to an execution where all threads' executions preserve program order, and there exists a total global order on all memory operations—all but one of the four possible outcomes is permitted. Specifically, r1 = 1 and r2 = 0 at the end of the test is a forbidden outcome according to SC. MCMs can be categorized by the non-SC outcomes that they permit or forbid for various litmus tests. In this example, the non-SC outcome is, by definition, forbidden by SC and TSO, e.g., x86-TSO [21].

Litmus tests are useful for conducting verification of hardware MCMs and aim to exercise behaviors most likely to exhibit bugs. Researchers have also proposed tools for efficiently generating complete (up to a bound in instruction count) suites of litmus test programs that encode all unique ordering behaviors imposed by a formally specified MCM [11, 19, 26]. Such comprehensive litmus test suites can be consumed by the Check family of tools [24, 25, 31, 33] to soundly and completely (with respect to the bound on litmus test program size) verify the correctness of hardware MCM implementations. In other words, given a collection of litmus tests, the Check tools will *prove* whether or not a specific microarchitecture is *guaranteed* to correctly execute every test, using *microarchitectural happens-before* (μhb) analysis, as described next.

Microarchitectural Happens-Before Analysis. The Check tools leverage a type of Lamport-style *happens-before* analysis [22], called μhb analysis, which relies on representing *hardware-specific program executions* as directed graphs, called *μhb graphs*. Fig. 1b presents an example of a μhb graph, depicting a non-SC execution of the MP litmus test of Fig. 1a on the RISC-V multi-V-scale processor [29, 31] (see Fig. 3a). Program order proceeds from left to right at the top of the graph. Nodes represent *hardware events* that take place during a program's execution, specifically an instruction (represented by a μhb graph column label) updating some particular hardware state element(s) (represented by a μhb graph row label) in the microarchitecture, such as a store updating a store buffer entry. A μhb graph node may represent an instruction updating either a single state element in the microarchitecture or a collection of state elements. Directed edges represent *happens-before relationships* between nodes, for example capturing that a store always updates an entry in its core-local store buffer *before* it updates the L1 cache.

Note that μhb nodes and edges are implied by the microarchitecture in combination with the executing program itself and may vary across executions of the same program on the same design. For example, the green PO edges in Fig. 1b result from the multi-V-scale's processor cores fetching instructions from instruction memory according to program order. Further, the pink edge ordering i1's update of mem before i2's update of regfile corresponds to the program-level data-flow between i1 which writes to y and i2 which reads the result of i1's write. The conditions under which μhb nodes and edges are instantiated in a μhb graph corresponding to a specific hardware design and program are elaborated on in §3.

μhb graphs enable efficient reasoning about whether a particular execution of a program (such as one that is expressly forbidden by the ISA MCM) is possible on a microarchitecture in question or not. Specifically, acyclic μhb graphs represent program executions that are possible on a given microarchitecture, whereas cyclic graphs represent impossible executions, since they would require events to be transitively causally related to themselves, implying a contradiction. The μhb graph in Fig. 1b features a cycle, indicating that the multi-V-scale (which implements SC [31]) correctly forbids the non-SC litmus test outcome, r1 = 1 and r2 = 0.

Axiomatic Specifications of Microarchitectures. Using an SMT solver [10, 25], the Check tools search the space of all possible executions of a litmus test on a given microarchitecture, with the intent of identifying executions that violate the ISA-specified MCM. Intuitively, this can be understood as enumerating all possible acyclic μhb graphs in search of ones which correspond to illegal program outcomes. To support this analysis, the microarchitecture is input as a μSPEC model, a series of *axioms* expressed in a specially-tailored typed first-order theory. These axioms describe how a legal hardware instruction flows through the microarchitecture, over the course of a program's execution, and how each instruction may interact with other instructions that are in-flight concurrently. In particular, the hardware state elements that an instruction updates and depends on, as well as the (partial) order on its state updates, must be specified. For example, a store instruction might first update the `fetch` pipeline register, followed by `execute` pipeline register, and lastly the `memory`. Or, a load's update to the `regfile` might depend on a prior store's update to `memory`, if the load and store access the same memory location.

With respect to μhb graphs, a μSPEC model describes μhb nodes and the *intra-instruction* happens-before edges required for modeling the execution of each instruction type, and which *inter-instruction* happens-before edges may exist between nodes corresponding to different instructions. In this paper, we define for the first time, what renders a μSPEC model complete with respect to a microarchitecture whose ordering behavior it is intended capture.

3 A TAXONOMY FOR CONSTRUCTING COMPLETE μSPEC MODELS

Establishing what constitutes a complete μSPEC model is the first step toward automatically generating one. Thus, a key contribution of our work is decomposing μSPEC models into a core set of building blocks, which we identify as four hierarchical categories of HBIs. In this section, we describe our taxonomy for categorizing these HBIs. In §4, we explain how we use this taxonomy to incrementally and systematically synthesize a complete set of HBIs (encoded as μSPEC axioms), and thus a complete μSPEC model, from RTL.

3.1 Happens-Before Invariants

Verilog is an *operational* description of how state updates take place in hardware. In contrast an axiomatic μSPEC model describes *happens-before invariants* (HBIs) that are preserved by a Verilog design in any executing program. A Verilog design might specify that the `fetch` pipeline register is updated with new a value at non-stall cycles. In contrast, a μSPEC model would assert an HBI stating that if some instruction `i0` precedes another instruction `i1` in program order, `i0` will update the `fetch` pipeline register before `i1` updates the `fetch` pipeline register. As mentioned in §1, μSPEC models can be decomposed into axioms that describe either execution paths of individual instructions (via intra-instruction HBIs, discussed in §3.2) or pairwise interactions between instructions during their execution on a microarchitecture (via inter-instruction HBIs, discussed in §3.3).

3.2 Intra-Instruction HBIs

Intra-instruction HBIs describe the execution paths of instruction types as they execute on a microarchitecture. Thus, a set of intra-instruction HBIs are required for each ISA instruction to encode their individual ordering behaviors in a μSPEC model. In our multi-V-scale case study (§5), RTL2μSPEC synthesizes a μSPEC model that encodes the behavior of RISC-V load and store instructions—`lw` and `sw`—only, given our focus on MCM verification in this paper.

Concretely, the set of intra-instruction HBIs for a particular instruction type specify which hardware state elements, at the granularity of sets of registers or memory cells, are updated on its behalf during its execution, along with a partial ordering on its induced state updates. In μhb graphs, the intra-instruction HBIs of an instruction type specify how nodes and intra-instruction edges (that is, edges that relate nodes corresponding to the same instruction instance) should be instantiated. For example, a set of intra-instruction HBIs corresponding to the execution path of `lw` on the multi-V-scale processor is shown below.

```
forall microops i, IsAnyRead i ⇒
  AddEdges [((i, inst_DX),  (i, mgnode_0)); % hbi0
            ((i, mgnode_0), (i, mgnode_3)); % hbi1
            ((i, mgnode_0), (i, regfile))]. % hbi2
```

Above, three HBIs have been encoded in a single axiom in the μSPEC DSL. `hbi0` specifies that for all instructions `i`, such that `i` is a memory read operation (`IsAnyRead i`), `i` will update the `inst_DX` state element before it updates the `mgnode_0` state element. Here, `mgnode_n` state elements each comprise several state elements that RTL2μSPEC deems equivalent in terms of ordering behaviors (see §4.4). The overall effect of the axiom above is to instantiate intra-instruction μhb nodes and edges for `lw` instructions in Fig. 1b.

3.3 Inter-Instruction HBIs

Inter-instruction HBIs describe how instructions can *interact with each other* during their execution. This characterization can be further refined by the type of interaction as detailed in §3.3.1 and §3.3.2. μSPEC model excerpts correspond to the multi-V-scale.

3.3.1 Structural Dependencies. A pair of instructions may be involved in a *structural dependency* if their accesses to a particular state element or a collection of state elements must be *serialized*. Structural dependencies take two forms—spatial and temporal.

Spatial Structural Dependencies. Spatial structural dependencies exist between a pair of hardware state updates that result from two instructions updating the *same* state element, which could be a single register or a single cell within a memory array. If two instructions `i0` and `i1` update the same hardware state element `s` during their execution—that is, their execution paths in μhb graph form both feature a node corresponding to an update of `s`—then their updates to `s` must be serialized. We therefore need some HBIs to describe this serialization order. As one possibility, `i0` and `i1` may update `s` in any order depending on the dynamic conditions of program execution. However, if `i0` and `i1` share a *reference order*, meaning they previously updated another common state element in a particular order or are ordered in the program, it is *possible* their updates to `s` will be constrained to take place in a way that either *always agrees with* or *always contradicts* the reference order. This amounts to three possible ordering behaviors.

The μSPEC excerpt below gives an example axiom that features a single inter-instruction HBI which corresponds to a spatial structural dependency, where the reference order is program order.

```
forall microops i0, i1,
  ProgramOrder i0 i1 ⇒
    AddEdge ((i0, inst_DX), (i1, inst_DX)). % hbi0
```

The above asserts that for all pairs of instructions `i0` and `i1`, if `i0` appears in program order before `i1` (`ProgramOrder i0 i1`, i.e., the reference order), then `i0` will update the `inst_DX` state element before `i1` does. The axiom instantiates inter-instruction μhb edges for pairs of instructions that are ordered in program order with respect to their updates on the `inst_DX` state element, such as the green edges labeled `PO` in Fig. 1b. Conceptually, this axiom enforces an in-order instruction fetch.

Temporal Structural Dependencies. Temporal structural dependencies exist between a pair of state updates that result from two instructions updating *distinct* state elements, where those state elements may only be accessed by a single instruction at any clock cycle. That is, temporal structural dependencies serialize the order in which instructions may update a state element within some set of state elements that is time-multiplexed between different instructions. For example, the horizontal dotted black lines in Fig. 1b illustrate the pipeline stage partitioning of the multi-V-scale, with `mgnode_0` and `mgnode_2` belonging to the same pipeline stage. Since only one instruction can access a pipeline stage at a time in this design, updates by different instructions to `mgnode_0` and `mgnode_2` are inherently serialized. As another example, single-ported processor memories serialize accesses that they process.

The serialization order of temporal structural dependencies has the same three ordering options as spatial structural dependencies—either order, consistent with a reference order, or inconsistent with a reference order. The μSPEC excerpt below gives an example of a single-HBI axiom that corresponds to a temporal structural dependency, where the reference order is the order in which a pair of instructions update the `inst_DX` register during their execution.

```
forall microops i0, i1, IsAnyWrite i0 ⇒ IsAnyWrite i1 ⇒
  EdgeExists ((i0, inst_DX), (i1, inst_DX)) ⇒
    AddEdge ((i0, mgnode_2), (i1, mgnode_0)). % hbi0
```

This axiom asserts that for all pairs of instructions `i0` and `i1`, such that `i0` and `i1` are both memory write operations, if `i0` updates `inst_DX` before `i1` does, then `i0` will update `mgnode_2` before `i1` updates `mgnode_0`.

3.3.2 Dataflow Dependencies. A dependency may also exist between a pair of instructions because they share data, not just because they contend for shared resources. Specifically, a pair of instructions may possess a *dataflow dependency* if one instruction can update a state element that is read from and therefore influences the state update of the other instruction. For example, a `sw` instruction in the multi-V-scale writes to the processor's memory, `mem`, and its memory update can be read by a `lw` instruction accessing the same address. As a result, the `sw` influences the `lw`'s update of the register file, `regfile`. The following μSPEC excerpt describes a single-HBI axiom that corresponds to such a dataflow dependency.

```
forall microops i0, i1,
  IsAnyWrite i0 ⇒ IsAnyRead i1 ⇒ SamePA i0 i1 ⇒
  SameData i0 i1 ⇒ NoWritesInBetween i0 i1 ⇒
    AddEdge((i0, (0, mem)), (i1, regfile)). % hbi0
```

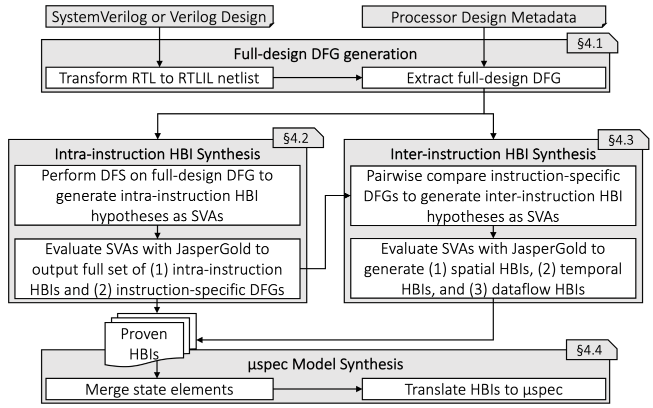


Figure 2: Overview of the RTL2μSPEC synthesis procedure, as detailed in §4.

Here, we assert that for all pairs of instructions `i0` and `i1`, where `i0` is a memory write and `i1` is a memory read, if both `i0` and `i1` access the same physical memory address with no intervening writes, and `i1` reads the value written by `i0`, then a dataflow dependency exists between `i0` and `i1` via `mem`. Since reads and writes can only communicate through main memory (`mem`) on the V-scale, the dataflow dependency implies that the write must update `mem` before the read accesses `mem` and writes the data it retrieves to `regfile`.

4 SYNTHESIZING μSPEC FROM RTL

RTL2μSPEC incrementally synthesizes a complete set of *proven* HBIs from an input Verilog design using a combination of static analysis and model checking. The synthesis flow is summarized in Fig. 2. We will refer to Fig. 3 throughout this section—a précis of the main stages of the synthesis procedure per our case study in §5.

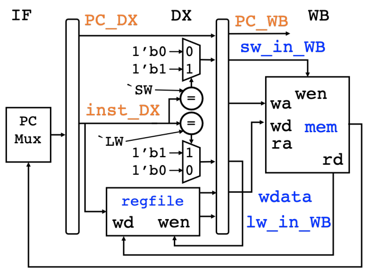
4.1 RTL to Full-Design Data Flow Graphs

The *data-flow graph* (DFG) representation of a Verilog design, referred to as a *full-design DFG* in this paper, contains all of the information needed for RTL2μSPEC to orchestrate the synthesis of a complete set of HBIs. Intuitively, this is because data-flow is a type of happens-before relation. Hence, RTL2μSPEC first extracts such a full-design DFG from the input Verilog.

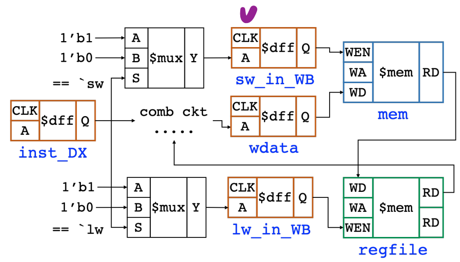
To extract a Verilog design's DFG, RTL2μSPEC uses two static analysis tools from the commercial Symbiotic EDA Suite,[3] Verific [8] and Yosys [46]. Verific is a parser that accepts Verilog or SystemVerilog as input and outputs a netlist. Yosys can then transform such a netlist into an intermediate representation (IR) called RTL Intermediate Language (RTLIL) which supports efficient Yosys-enabled netlist analyses and transformations.[4] Note that RTLIL is simply an alternate netlist representation.

Fig. 3b illustrates a simplified excerpt of the netlist that corresponds to the multi-V-scale design in Fig. 3a. The netlist was produced by running the multi-V-scale through Verific, and then running the Verific-generated netlist through Yosys. Observe that

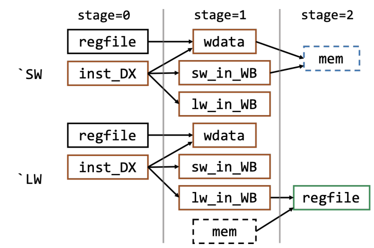
[3]We use Symbiotica to support SystemVerilog syntax with Verific.
[4]Yosys can also transform Verilog into RTLIL, but RTL2μSPEC uses Verific as its front end parser to support SystemVerilog syntax.



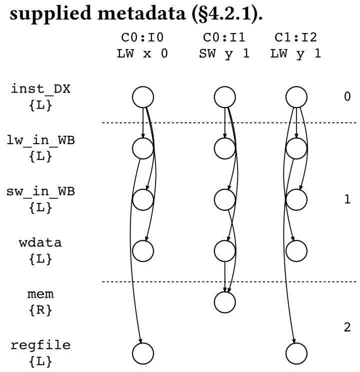
(a) Illustration of the RISC-V V-scale. Some relevant state elements are labeled in blue/orange. Orange state is part of RTL2μSPEC's designer-supplied metadata (§4.2.1).



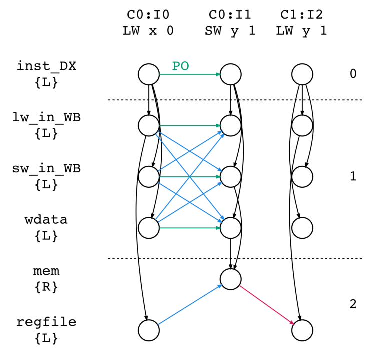
(b) Netlist generated by Verific+Yosys from (a). Brown/blue/green nodes are state elements updated during execution of both lw and sw/only sw/only lw.



(c) Per-instruction DFGs produced by specializing the netlist in (b). Solid (resp. dashed) boxes denote local (resp. remote) state elements.



(d) Intra-instruction HBIs (black nodes and edges). Dotted black lines and stage ID labels (§4.2.2) denote pipeline stage partitionings.



(e) Adds inter-instruction HBIs to (d). Green/blue/pink edges correspond to spatial/temporal/data-flow HBIs. PO denotes program order.

```
Axiom W_path: forall microops i1,
IsAnyWrite i1 ⇒ AddEdges [
((i1, inst_DX), (i2, sw_in_WB));
(i1, inst_DX), (i1, lw_in_WB));
(i1, sw_in_WB), (i1, mem))].
Axiom Temporal:
forall microops i1, i2,
IsAnyRead i1 ⇒ IsAnyWrite i2 ⇒
ProgramOrder i1 i2 ⇒ AddEdge(
(i1, regfile), (i2, mem)).
Axiom Spatial:
forall microops i1, i2,
IsAnyRead i1 ⇒ IsAnyWrite i2 ⇒
ProgramOrder i1 i2 ⇒ AddEdge(
(i1, wdata), (i2, wdata)).
```

(f) Three axioms from the RTL2μSPEC-synthesized μSPEC model of the multi-V-scale. The impact of these axioms on generating μhb graphs can be seen in(c).

Figure 3: Given the multi-V-scale in (a), represented as a netlist in (b), RTL2μSPEC generates per-instruction DFGs, as in (c), and deduces from them intra-instruction HBIs, as in (d), and inter-instruction HBI hypotheses, as in (e). HBI hypotheses are evaluated by JasperGold, and only proven hypotheses are included as axioms in the final μSPEC model, as in (f).

the netlist is simply a CDFG. Nodes are standard cells such as registers, memory arrays, and combinational logic gates. Edges represent wired connections between standard cells.

Since μSPEC models articulate HBIs at the granularity of hardware state elements, our target full-design DFG contains nodes that correspond solely to these state elements and edges which represent (potential) data-flow relationships between them. RTL2μSPEC's transformation of a Verilog design into RTLIL form enables it to easily produce such a DFG with the help of a new RTLIL analysis pass in Yosys. Specifically, this RTLIL analysis pass performs a depth-first-search (DFS) over all standard cells in the netlist, establishing a mapping between parent and child state elements that are connected via pure combinational logic. The full-design DFG is then constructed using the Verilog design's state elements as nodes, and the parent-to-child mappings as edges. In other words, the full-design DFG is constructed by collapsing out all combinational circuits separating state elements, including control flow, in the RTLIL netlist. Since this collapsing effectively assumes that all possible data-flows happen for every execution of every possible instruction, the full-design DFG represents an *over-approximation* of the hardware-level data-flow that can be induced by any microarchitecture-supported instruction. RTL2μSPEC uses this over-approximation to synthesize intra-instruction HBIs in §4.2.

Note that the analysis in this section need only consider the unique modules in the input design, such as a single core plus all shared resources in a homogeneous multi-core setting.

4.2 Synthesizing Intra-Instruction HBIs

A full-design DFG (§4.1) for a Verilog implementation contains the information needed by RTL2μSPEC to synthesize intra-instruction HBIs for each instruction type of interest; this can be reduced to specializing the full-design DFG for each instruction type, resulting in *instruction-specific DFGs*. An instruction-specific DFG captures (1) the precise set of state elements that are updated during the execution of a particular instruction type, expressed as DFG nodes, and (2) the relative (partial) order of these updates, expressed as DFG edges, since the data-flow edges represent the flow of information from one register to the next in time.

4.2.1 User-Supplied Core-Local Metadata. To support the construction of instruction-specific DFGs, RTL2μSPEC requires three pieces of user-supplied design metadata.

First, the **instruction fetch register** (IFR), which holds instructions when they are first fetched from memory, must be identified, and its signal name specified. Using this signal, RTL2μSPEC can reference the starting point of an instruction's execution life-cycle on the microarchitecture.

Second, per-pipeline stage **program counter registers** (PC registers or PCRs) must be identified, which are used by RTL2μSPEC to precisely reason about an instruction's presence in a particular pipeline stage and thereby attribute specific state updates to its execution. RTL2μSPEC refers to these registers via an array, called PCR, where PCR[0] is located in the same pipeline stage as the IFR by default, and PCR[i] corresponds to the PCR in the i^{th} pipeline stage with respect to the IFR's pipeline stage.

Third, a special PC signal, the **instruction memory PC** (IM_PC), which is used to index into and access instruction memory, must be identified. Note that all registers included in the PCR array should be reachable from the IM_PC in the full-design DFG.

In addition to the design metadata above, RTL2μSPEC requires the user to supply the **binary encodings** of all instructions which will be included in the synthesized μSPEC model. For example, in our case study in §5 we direct RTL2μSPEC to consider lw and sw instructions only, given our goal of MCM verification.

4.2.2 Filtering Front-End State Elements. To construct a specialized DFG for an instruction type, RTL2μSPEC must identify the subset of full-design DFG nodes whose corresponding state elements are updated on behalf of its execution. Since the IFR marks the start of an instruction's execution life-cycle, all nodes that *precede* the IFR (e.g., front-end predictor state) can be excluded from further consideration. To perform this filtering, RTL2μSPEC first identifies all nodes in the full-design DFG that are reachable from the IM_PC. During this identification process, RTL2μSPEC also tags each reachable node with an integer value, stage, capturing its distance from IM_PC in the full-design DFG. Since edges in the full-design DFG represent single-cycle data-flow relationships,[5] stage effectively associates each register with a pipeline stage, and it is used to precisely attribute hardware state updates to a particular instruction's execution (as it passes through some stage) as detailed further in §4.2.3. Directed cycles in the full-design DFG are handled by retaining the *shortest* distance from IM_PC as the stage for each node.

All nodes with a corresponding stage value less than that which is associated with the IFR (including IM_PC) are filtered from the reachable set, since they precede the IFR in the design. The remaining reachable nodes correspond to state elements that *may* be updated on behalf of instructions as they flow from the IFR through the various stages of execution. Note that stage values for nodes are updated at this point such that the IFR is associated with stage number 0.

4.2.3 Generating Intra-Instruction HBI Hypotheses. With the filtered set of candidate nodes, RTL2μSPEC can now construct specialized DFGs for each instruction type of interest. For each instruction type, RTL2μSPEC needs to determine which of the filtered nodes, that are also reachable from the IFR in the full-design DFG, are indeed updated on behalf of its execution. Related to this point, RTL2μSPEC currently assumes that each instruction type can exhibit at most one execution path through the design under verification—the *single-execution-path assumption*. In other words, the set of state elements updated by an instruction are always the same each time the instruction executes. Phrased differently, an instruction will always instantiate the same column of μhb nodes in a μhb graph. Thus, if RTL2μSPEC finds that a state element can *ever* be updated on behalf of a particular instruction's execution, it concludes that it is *always* updated on its behalf. §6.4 discusses the implications of this limitation, which we plan to alleviate in future work.

To isolate the set of nodes whose corresponding state elements are update by a specific type of instruction, RTL2μSPEC relies on a set of *automatically generated SVAs*, which encode *HBI hypotheses*.[6] In general, HBI hypotheses are are evaluated using the JasperGold property verifier [12], and proven hypotheses correspond to valid HBIs that will be inserted into final μSPEC model. In the case of intra-instruction HBI synthesis, HBI hypotheses are assertions designed specifically to determine whether or not an instruction's execution can ever update a particular state element (i.e., *always update*, per the single-execution-path assumption above).

4.2.4 Formulating Intra-Instruction HBI Hypotheses as SVAs. The RTL2μSPEC tool automatically synthesizes HBI hypotheses formulated as SVAs with the help of *SVA templates*. For intra-instruction HBI hypotheses, RTL2μSPEC makes use of two SVA templates, shown in Fig. 4. Both leverage the association between registers in the PCR array and other non-PC state elements established by identical stage labels (§4.2.2) to attribute the update of a non-PC state element s in stage i (i.e., stage(s) = i) to an instruction whose PC is contained in stage i's PCR, namely PCR[i]. We describe below how Fig. 4's SVA templates are used to synthesize intra-instruction HBIs for a *single instruction type*. The process is repeated for each instruction type of interest.

The first SVA template (Fig. 4a) is instantiated once for every node (i.e., state element) in the filtered set of candidate nodes (§4.2.2) that is reachable from the IFR in the full-design DFG. Thus, the property is parameterized by instruction type (op) and state element (s). It attempts to prove (via assertion A0) that when a particular instruction i0 (with a particular type—op) is passing through the stage associated with state element s ('PCR_<stage(s)> == pc0, where pc0 is the PC associated with i0), that s will never change its value. A failed proof signifies that s *can* be updated by the instruction type of interest when it passes through its corresponding stage. State elements that can never be updated on behalf of the instruction under evaluation are ignored henceforth.

While the first SVA template is able to deduce that a particular state element can be updated by a particular instruction once it progresses to a particular stage, the second SVA template (Fig. 4b) attempts to prove that said instruction will eventually make its way to the stage where it is capable of updating said state. For each stage that contains state element(s) that were retained after evaluating the first set of SVAs (Fig. 4a), the SVA in Fig. 4b attempts to prove (via assertion A1) that the instruction type of interest will eventually progress to and exit said stage when it executes. Thus, the property is parameterized by instruction type (op) and pipeline stage (stage), and a successful proof certifies forward progress.

[5] Data-flow relationships in the full-design DFG are "single-cycle," since they correspond to direct connections between state elements through combinational logic that was collapsed out (§4.1).

[6] We use the terms SVA and HBI hypothesis interchangeably in this paper.

```
P0: assume (first |-> ( (`PCR_0 != pc0 [*0:$]) ##1
        (`PCR_0 == pc0 [*1:$]) ##1 (`PCR_0 != pc0) ));
P1: assume (first |-> s_eventually(`PCR_<stage(s)> == pc0));
P2: assume (`PCR_0 == pc0 |-> `IFR == i0);
P3: assume (opcode(i0) == op);
A0: assert (`PCR_<stage(s)> == pc0 |-> s == $past(s));
```

(a) Assertion `A0` attempts to prove that state element `s` will never be updated by the execution of instruction `i0` with opcode `op`. `PCR_<stage(s)>` represents string concatenation of `PCR_` with the `stage` ID associated with `s`.

```
P4: assume (`PCR_0 == pc0 |-> `IFR == i0);
P5: assume (opcode(i0) == op);
P6: assume (first |-> strong((`IFR == `NOP &&
        `PCR_0 != pc0 [*0:$]) ##1 (`PCR_0 == pc0) ) );
A1: assert (first |-> s_eventually( (`PCR_<stage> == pc0) ##1
        (!(`PCR_<stage> == pc0)) ));
```

(b) Assertion `A1` attempts to prove that instruction `i0` with opcode `op` will eventually progress to and exit some pipeline stage, `stage`. It is used to prove precondition `P1` in (a) for `stage`s where instructions of type `op` can update state—i.e., where instructions of type `op` fail `A0` for some `s` in `stage`.

Figure 4: RTL2μSPEC uses the SVA templates in (a) and (b) to instantiate intra-instruction HBI hypotheses and ultimately synthesize intra-instruction HBIs (§4.2.3). Template parameters are blue. Symbolic values that correspond to the instruction under evaluation by the property are green.

Nodes which pass the HBI hypothesis evaluation of §4.2.3 are considered to be always updated on behalf of the instruction type under evaluation and are used to construct a specialized instruction-specific DFG. This is done by extracting a new DFG from the full-design DFG that is restricted to only contain nodes corresponding to these always-updated state elements. During extraction, DFG edges are retained if they directly relate extracted nodes. Immediate *parent nodes* of the always-updated state-elements in the full-design DFG are also extracted. These aid in synthesizing inter-instruction HBIs that result from data-flow dependencies as detailed in §4.3.5.

Fig. 3c gives an example of two simplified instruction-specific DFGs corresponding to the `sw` (top) and `lw` (bottom) instructions of the RISC-V V-scale. The *primary root node* of each graph is the `IFR`—the `inst_DX` signal for the V-scale—and all nodes reachable from it are always updated on behalf of the instruction that corresponds to the DFG. Other nodes with no incoming edges, such as `regfile` and `mem`, are reserved parent nodes.

Recall that the intra-instruction HBIs for a particular instruction type articulate which μhb nodes and intra-instruction μhb edges must exist in any μhb graph featuring an instance of said instruction. The nodes reachable from the primary root node in an instruction-specific DFG indicate relevant μhb nodes, while directed data-flow edges (relating the reachable nodes) indicate relevant intra-instruction μhb edges. In Figs. 3d and 3e, the nodes and black edges correspond to intra-instruction HBIs for `lw` and `sw` on the V-scale.

4.3 Synthesizing Inter-Instruction HBIs

After synthesizing a complete set of intra-instruction HBIs, the RTL2μSPEC tool synthesizes inter-instruction HBIs which result from structural or data-flow dependencies (§3.3). For each category of inter-instruction HBIs, RTL2μSPEC compares all pairs of per-instruction DFGs to identify all possible inter-instruction interactions, each of which requires an HBI to be instantiated. Whenever RTL2μSPEC determines that an HBI must be synthesized to describe a potential pairwise interaction between instructions, it formulates HBI hypotheses (as SVAs) so that the precise HBI can be deduced with the help of JasperGold. In this way, RTL2μSPEC ensures that the final μSPEC model contains a complete set of inter-instruction HBIs that have all been formally verified.

Notably, inter-instruction HBIs can describe interactions between instructions via *local* on-core resources (e.g., pipeline registers) or resources that are off-core and thus *remote* (e.g., memories, including on-chip caches). Furthermore, inter-instruction HBIs can describe interactions between instructions executing on either the same processor core (*intra-core HBIs*) or on different cores (*inter-core HBIs*). Inter-core HBIs inherently involve interactions via shared remote state whereas intra-core HBIs may be facilitated via interactions through either local or remote state elements.

When instantiating inter-instruction HBIs as SVAs, RTL2μSPEC distinguishes HBI hypotheses involving local versus remote resources. That said, the general structure of inter-instruction HBI hypotheses remains the same regardless of whether local versus remote state elements are involved. §4.3.1, §4.3.2, and §4.3.5 give the general procedure for generating relevant inter-instruction HBI hypotheses regardless of the types of state elements involved, while §4.3.3 describes how HBI hypotheses are instantiated in SVA form in slightly different ways for local versus global resources.

4.3.1 Generating Spatial Structural HBI Hypotheses. A spatial structural dependency exists between a pair of instructions if they both update an *identical* hardware state element during their execution. To identify these dependencies, RTL2μSPEC iterates over all pairs of instructions and compares their DFGs to find common nodes (representing identical state elements) which are reachable from the `IFR`s (the primary root nodes) in both. Given a pair of instructions, each such pair of common nodes constitutes a unique spatial structural dependency. In Fig. 3c, `inst_DX`, `sw_in_WB`, `lw_in_WB`, and `wdata` (four distinct state elements) are all updated by both `lw` and `sw`, since nodes representing these state elements are all reachable from the `IFR` nodes in their corresponding DFGs (recall that `inst_DX` is the `IFR` for the multi-V-scale). Four spatial structural dependencies therefore exist between `lw` and `sw` on the multi-V-scale. Note that the four spatial dependencies identified here all involve local state elements, but spatial dependencies can involve global state elements as well.

A spatial structural dependency between a pair of instructions always results in the inclusion of a corresponding HBI in the final μSPEC model. However, the *direction* of the HBI must be deduced. For each spatial structural dependency identified between all pairs of instructions (including same-instruction pairs), RTL2μSPEC either directly outputs an HBI or generates HBI hypotheses to determine the direction of the HBI corresponding to the dependency with respect to a reference order if one exists.

As discussed in §3.3.1, pairs of instructions cannot be constrained to update a common state element in a particular order *without* *a relevant reference order*. Thus, given a structural dependency involving such an instruction pair, RTL2μSPEC will synthesize an HBI indicating that while updates to the common state element on behalf of the instructions of interest are *ordered*, their *direction* is unconstrained. No proof effort is necessary. One such example arises when RTL2μSPEC is considering potential inter-core interactions between instructions and identifies a remote memory cell (e.g. one cell of `mem` in the multi-V-scale) as a common node between a pair of per-instruction DFGs (e.g., the DFGs corresponding to `sw` instructions in the multi-V-scale).

For pairs of instructions entwined in a structural dependency *with a relevant reference order* that RTL2μSPEC has identified (e.g., instructions executing on the same core which minimally have program order as a reference order), RTL2μSPEC generates HBI hypotheses in an attempt to prove that the instructions will always update the common state element in an order that is *consistent with* their reference order. These hypotheses attempt to prove that:

For instructions `i0` *and* `i1` *and state element* `s`*, if* `i0` *is ordered before* `i1` *with respect to some reference order (e.g., program order), then* `i0` *will update* `s` *before* `i1` *updates* `s`*.*

§4.3.3 gives more detail on precisely how inter-instruction HBI hypotheses are instantiated as SVAs, depending on whether `s` is local or remote. Regardless, to transform these HBI hypotheses into HBIs, the instantiated SVAs are evaluated by JasperGold, and proven hypotheses are translated by RTL2μSPEC into μSPEC axioms. On the other hand, invalid hypotheses require a second round of evaluation to check if the instructions always perform their updates in an order that is *inconsistent with* the reference order. Regardless of whether or not this final hypothesis is proven, an HBI can be deduced for inclusion in the final μSPEC model, as structural HBIs can be ordered in one of three ways (see §3.3.1), and structural HBI hypotheses always imply existence of a structural HBI.

4.3.2 Generating Temporal Structural HBI Hypotheses. Temporal structural dependencies occur when a pair of *distinct* state elements can only be accessed by one instruction at a time, and therefore updates by different instructions to these distinct elements are serialized by the hardware. RTL2μSPEC considers two sources of temporal dependencies: (1) state elements that belong to the same pipeline stage and are only accessible by a single instruction at any cycle, and (2) arrays of state elements (such as a register file or memory) whose access is constrained by a restricted interface.

To identify temporal dependencies, RTL2μSPEC iterates over all pairs of instructions and compares their corresponding DFGs. For each pair of DFGs, RTL2μSPEC looks for pairs of nodes (one in each DFG) that reside in the same pipeline stage (using `stage` labels from §4.2.2) or access the same register or memory array. Such node pairs *may* signify true temporal structural dependencies between instructions. *True* temporal structural dependencies identified by pairwise DFG analysis always result in the inclusion of a corresponding HBI in the final μSPEC model. As with spatial structural dependencies, the direction of the HBI must be deduced. *False* temporal structural dependencies involve instructions that can update a pair of state elements *concurrently*. For example, pairwise DFG analysis may determine that two instructions update a common memory array where the memory array is in fact multi-ported. RTL2μSPEC presently assumes single-ported memories, which is sufficient for our case study in §5, but supporting multi-ported memories is straightforward—one additional SVA check to filter out false temporal structural dependencies is all that is required.

As with spatial structural HBIs, if there is no relevant reference order that can be established for a given true temporal structural dependency, an HBI can be simply synthesized without any hypothesis generation or evaluation. True temporal structural dependencies for which a relevant reference order can be established require extra proof effort via temporal HBI hypotheses. The generated temporal HBI hypotheses attempt to prove that:

For instructions `i0` *and* `i1` *and state elements* `s0` *and* `s1`*, if* `i0` *is ordered before* `i1` *with respect to some reference order (e.g., program order), then* `i0` *will update* `s0` *before* `i1` *updates* `s1`*.*

§4.3.3 explains how hypotheses matching the format above are captured as SVAs and evaluated by JasperGold. Again, if the first hypothesis proof fails, RTL2μSPEC attempts to prove that the updates are sequenced in the *reverse* order with respect to the reference order. Also note, that spatial HBI hypotheses are simply a specialization of temporal HBI hypotheses, where `s0 = s1`.

4.3.3 Formulating Structural HBI Hypotheses as SVAs. This section explains how RTL2μSPEC instantiates the inter-instruction HBI hypotheses from §4.3.1 and §4.3.2 (and upcoming §4.3.5) as SVAs, depending on whether they involve local or remote state elements.

Structural HBI Hypotheses Involving Local State. When RTL2μSPEC instantiates structural HBIs involving local state as SVAs, designer-provided PCRs (§4.2.1) are again used to uniquely identify in-flight instructions and attribute particular state updates to their execution (§4.2.3). Recall that an update of local state element `s` is attributed to the instruction whose PC is contained in the PCR associated with `s`'s pipeline stage at the cycle `s` is updated. Notably, for a structural dependency involving local state, the two PCRs that are relevant for instantiating a structural HBI hypothesis are the same. Thus, the SVAs generated by RTL2μSPEC to deduce structural HBIs reduce to checks of the order in which two instructions, `i0` and `i1`, update a common PCR with respect to a reference order.

Notably, for all pairs of registers within the same pipeline stage (which all share a PCR), the direction of all relevant structural HBIs can be deduced by evaluating one or two SVAs—one (resp. two) if the structural HBIs associated with that stage are consistent (resp. inconsistent) with a reference order. This results in significant runtime savings for RTL2μSPEC which can evaluate, for structural HBIs involving local state, a number of SVAs that scale as a function of the number of pipelines stages rather than local state elements.

Structural HBI Hypotheses Involving Remote State. When an instruction updates a remote state element, the update is typically facilitated via a communication interface that connects the processor core executing the instruction to the remote resource. Thus, remote state updates are generally not attributed to particular instruction PCs, but rather to particular requests over the communication interface. This scenario necessitates a new approach for detecting state updates that are initiated by specific instructions, beyond associating state elements with same-stage PCRs.

To instantiate HBI hypotheses that require reasoning about the ordering of updates to remote state (e.g., memories, including on-chip caches), RTL2μSPEC assumes the existence of a generic *request-response interface*. §4.3.4 describes the structure of this interface, which the designer must expose to RTL2μSPEC for each remote state element (or array of state elements).

Given a request-response interface through which instructions can update a particular remote resource, RTL2μSPEC can instantiate (as SVAs) HBI hypotheses, like those in §4.3.1 and §4.3.2, involving said resource. The designer-exposed request-response interface (1) enables SVAs to attribute remote state updates to specific instructions without solely using PCRs, and (2) decomposes ordering proofs involving remote resources into multiple fine-grained and localized SVAs. For an HBI involving a remote resource, RTL2μSPEC evaluates it with the help of three SVAs, summarized as follows:

Req-Snd: *Requests corresponding to the instructions' state updates are sent from their local core to the remote resource in an order that is consistent with their reference order (e.g., program order).* ***Req-Rec:*** *For any two requests that are sent from the same core to the remote resource, they are received in the order in which they were sent.* ***Req-Proc:*** *For any two requests from the same core that are received by the shared resource, they are processed in the order received.*

Consider a temporal HBI hypothesis that aims to prove that a pair of same-core instructions always update a remote memory array in an order that is consistent with program order. Three SVAs will be instantiated. First, the **Req-Snd** SVA will be formulated, using PCRs to associate the sending of requests to the memory array with particular instructions. Second, the **Req-Rec** SVA will leverage the exposed request-response interface, which labels requests with IDs of the cores that issued them, to determine if the memory array receives same-core requests in the order in which they were sent. Finally, the **Req-Proc** SVA will also leverage requests' core IDs to determine if the memory array processes same-core requests in the order in which they are received.

If any of the three SVAs associated with an HBI hypothesis involving a remote resource are invalidated, they are re-evaluated with an inverted reference order. Further, RTL2μSPEC can refine hypotheses to detect ordering relationships that are only preserved for same-address accesses.

4.3.4 User-Supplied Interface Metadata. RTL2μSPEC requires communication interfaces that facilitate updates of remote state to be structured according to a generic request-response template. For each remote resource, RTL2μSPEC requires the designer to supply a mapping between output ports of unique processor cores and input ports of the remote resource with respect to five main signals—`transaction type`, `transaction size`, `address`, `data`, and `core ID`. Furthermore, RTL2μSPEC requires for each remote resource that any signals used to indicate the completion of processing a request are also identified (and their signal names specified).

4.3.5 Generating Data-flow HBI Hypotheses. A pair of instructions are entwined in a data-flow dependency if one instruction updates a state element that is read from and subsequently influences a state update of the other. To identify data-flow dependencies between instructions, RTL2μSPEC again considers all pairs of per-instruction DFGs. For a given DFG pair, RTL2μSPEC searches for common nodes, where one node instance is reachable from the `IFR` (the primary root node) in one instruction's DFG (the *writer* instruction) and the other constitutes a parent node (§4.2) in the other instruction's DFG (the *reader* instruction). Such a pair of nodes signifies a data-flow dependency from the writer's update of the common node to the reader's update of the common node's *child node* (in its DFG). In Fig. 3c, `mem` is one such common node in the `sw` and `lw` DFGs that is written by `sw` instructions but is read from and influences the state updates of `lw` instructions with respect to the `regfile`.

To deduce the HBIs that correspond to identified data-flow dependencies, RTL2μSPEC generates HBI hypotheses which try to prove that:

For instructions `i0` *and* `i1` *and state element* `s`*, where* `i0` *updates* `s` *which can pass data to* `i1`*, if* `i0` *is ordered before* `i1` *with respect to some reference order, then* `i0` *will write to* `s` *before* `i1` *reads* `s`*.*

4.3.6 Formulating Data-Flow HBI Hypotheses as SVAs. To instantiate data-flow HBIs as SVAs, RTL2μSPEC must again be able to attribute state updates to particular instructions. It does so with the help of user-identified PCRs (for local state elements) and request-response interfaces (for remote state elements), as in §4.3.3. Note that RTL2μSPEC assumes that memory operations are functionally correct. For example, a write of some data value `v` to some state element `s` (e.g., a memory location), will indeed write `v` to `s`. Likewise, a read of a state element `s` will return the exact value stored in `s`.

4.4 From Validated HBIs to a μSPEC Model

§4.2 and §4.3 detail RTL2μSPEC's procedure for collecting a complete set of *proven correct* HBIs to describe an input microarchitecture.

Node Merging. Thus far, all deduced HBIs operate at the granularity of *individual state elements*. To improve the efficiency and scalability of μSPEC model analyses, RTL2μSPEC agglomerates state elements into groups, and updates HBIs accordingly. This abstraction procedure is reducible to a μhb graph node merging problem. RTL2μSPEC merges a pair of intra-instruction nodes for an instruction if the two nodes reside at the same distance from the `IFR` node *and* are both involved in the same set of inter-instruction HBIs.

Syntax Translation. After node merging, the final μSPEC model is generated via syntactic translation of validated HBIs to μSPEC.

5 MULTI-V-SCALE CASE STUDY

We demonstrate the efficacy of RTL2μSPEC by using it to synthesize a complete μSPEC model from the multi-V-scale processor and thereby conduct Check-based verification of its MCM.

5.1 The RISC-V multi-V-scale

The multi-V-scale [29, 31] consists of four Sequentially Consistent [23] cores. Each core features a three-stage in-order pipeline implementing the RISC-V 32-bit base instruction set. The four cores interact with each other via a single shared memory module. The design features a single arbiter that connects all cores to the memory and allows one core to issue a data memory request per cycle, according to a round-robin policy. On concurrent memory requests, the arbiter services only one core and stalls all others looking to issue requests. The arbiter can accept a new memory request on each clock cycle due to the memory's pipelined design.

A single core of the multi-V-scale features 1,042 wires, 605 standard cells, 55 registers and 2 memories, and 1,088 D flip-flop bits. The four-core design features 15,616 wires, 3,185 standard cells, 200 registers and 5 memories, and 4,135 D flip-flop bits. To run RTL2μSPEC on the multi-V-scale, we slightly modify the design in two ways. First, to conform to RTL2μSPEC's structural requirements on request-response communication interfaces (§4.3.4), we extend the output port of the arbiter and all buffers holding memory requests with two bits each that tag memory requests with core IDs. The result is a 4-bit increase in design size with no additional logic. Second, we modify the multi-V-scale's memory module so that Yosys can recognize it as an addressable array with a restricted interface (§4.3.2). Yosys originally interpreted the multi-V-scale's memory as a collection of distinct memory cells. We suspect this was due to the complexity of the original memory module interface—a unified memory with split data and instruction access ports, where instruction memory access ports are parameterized by core count. Our split of instruction and data memory into distinct modules resolves this issue. [7]

We supply RTL2μSPEC with the slightly modified multi-V-scale design (in SystemVerilog), along with all required design metadata (§4.2.1 and §4.3.4). RTL2μSPEC is loaded as a C++ extension to the Symbiotic EDA Edition [20201202A] of Yosys v0.9+3715. HBI hypotheses are embedded in SVA 2009 [1] and evaluated with JasperGold v2016.09. All experiments are run on a compute node featuring a dual 32-core 2.9GHz Intel Xeon CPUs with 512GB RAM.

5.2 Verifying the multi-V-scale's MCM

We use the latest release of the Check MCM verification tools, called COATCheck [25], to verify the multi-V-scale's adherence to Sequential Consistency. For the litmus test input, we use a suite of 56 litmus tests composed of both hand-written tests from an x86-TSO litmus test suite [35] and tests that were automatically generated with the `diy` framework [2]. The μSPEC model input is synthesized by RTL2μSPEC from the multi-V-scale's RTL implementation. The correct-by-construction μSPEC model and litmus tests were supplied to COATCheck which determined that the synthesized model passed all 56 litmus tests, as detailed in §6.

Prior work has also sought to address the gap between μSPEC models and RTL, namely RTLCheck [31]. RTLCheck seeks to validate a manually-constructed μSPEC model against a Verilog implementation *with respect to a suite of litmus test programs.* The user supplies as input a μSPEC model, a Verilog design, a set of mappings to link to the two, and a suite of litmus tests. RTLCheck then simultaneously checks for each litmus test that the μSPEC model faithfully captures the Verilog behaviors exercised by the test and that the test exhibits the correct behavior when it runs on the microarchitecture. Similar to RTL2μSPEC, RTLCheck leverages SVAs and JasperGold.

We run the RTLCheck verification procedure on the multi-V-scale with the same suite of 56 litmus tests, both of which were acquired from the RTLCheck Github repository [32]. We compare the performance, scalability, and completeness of RTLCheck and RTL2μSPEC along two dimensions: (1) ability to deduce a correct μSPEC model, and (2) ability to conduct litmus test-based verification on Verilog designs. We note that we compare RTLCheck to RTL2μSPEC using the same JasperGold solver engines. Given this, our reported runtimes for RTLCheck are improved from the original paper [31], due to the presence of JasperGold's `Tri` engine that was released after RTLCheck's original publication.

[7] Note that RTL2μSPEC can handle multi-ported memories, e.g., the multi-V-scale's `regfile`.

6 RESULTS

6.1 Bug Discovered in the multi-V-scale

During multi-V-scale μSPEC model synthesis, RTL2μSPEC exhibited two assertion failures when trying to prove an intra-core temporal HBI involving a remote state array—memory. Specifically, RTL2μSPEC instantiated a set of SVAs in an attempt to prove that two memory requests from the same core will update the memory in an order concordant with program order (§4.3.3). One SVA asserted that, if a pair of memory requests from the same core are received by the memory, the memory will process them in the order in which they are received. This SVA was refuted for `sw/lw x` $\xrightarrow{\text{po}}$ `sw y` pairs, where `x != y`, implying that the final μSPEC model would have been unable to preserve program order for such instruction sequences.

The counterexample trace produced by JasperGold featured an undefined instruction—with an encoding *similar* to RISC-V's `sw` but where the `width` field has an undefined value (`funct3=3'b111`)—updating memory. Instead it should have triggered an exception. Since the erroneous `sw` encoding was undefined, it was not properly accounted for by the memory's request-tracking logic which tags requests with unique IDs. JasperGold was thus able to attribute the latter `sw` in a `sw/lw x` $\xrightarrow{\text{po}}$ `sw y` sequence to an invalid instruction with same (unconstrained) request ID that was actually received by the memory earlier in time. We fixed this issue in the multi-V-scale before re-running RTL2μSPEC to synthesize a fresh μSPEC model.

6.2 RTL2μSPEC Performance Breakdown

Fig. 5 summarizes the overhead of synthesizing a complete μSPEC model for MCM verification (`lw` and `sw` instructions only) from the multi-V-scale with RTL2μSPEC. Overall, it takes **6.84 minutes to synthesize the μSPEC model**, including 2.14 seconds of Verilog parsing and HBI hypothesis generation and 1.36 seconds of Python post-processing. JasperGold's evaluation of 122 RTL2μSPEC-synthesized SVAs accounts for the bulk of the run time—6.78 minutes in total. Running COATCheck on the RTL2μSPEC-synthesized μSPEC model takes 1.37 seconds in total for all 56 litmus.

Optimizing Structural HBI Hypotheses. When generating structural HBI hypotheses, RTL2μSPEC considers specific pairs of instruction types at a time. One such hypothesis might be instantiated to determine the order in which `lw` and `sw` instructions, specifically, update some common state element (e.g., `wdata` in Fig. 3). As an optimization, RTL2μSPEC relaxes instruction-specific structural HBI hypotheses to prune the number of SVAs that JasperGold must evaluate. In particular, instruction-specific properites are modified such that they refer to arbitrary pairs (rather specific pairs) of instructions. In other words, RTL2μSPEC tries to prove an

		Intra-Instruction	Structural (Spatial)	Structural (Temporal)	Dataflow	Total
# SVAs		107	1	12 (+1 spatial)	2	120
Runtime (s)		354.99	5.24	31.08	15.77	407.06
Runtime/SVA (s)		3.32	5.24	2.59	7.89	3.34
# HBI Hypo. / # HBI	Local	180 / 155	129 / 129	4,762 / 4,719	2 / 2	5,073 / 5,005
	Global	25 / 22	15 / 15	59 / 59	1/ 1	100 / 97
	Total	205 / 177	144 / 144	4,821 / 4,778	3 /3	5,173 / 5,102

Figure 5: Results for RTL2μSPEC's synthesis of a multi-V-scale μSPEC model. Some HBI hypotheses graduate to HBIs (by proving SVAs) and are included in final μSPEC model. The total runtime is 6.78 minutes, with a std. dev. of 8.60 seconds for proving SVAs. (+1 spatial) indicates that 1 spatial SVA served to validate the remaining temporal HBI hypotheses that are not covered by the 12. All runtimes are averaged over five runs of RTL2μSPEC.

instruction-specific property for all possible pairs of instructions simultaneously. If the relaxed property fails, RTL2μSPEC reverts back to the finer-grained instruction-specific encoding. This optimization reduced the number of properties evaluated by JasperGold (while synthesizing a μSPEC model of the multi-V-scale) by a factor of about i^2, where i is the number of instruction types evaluated.

6.3 Performance and Proof Coverage

Fig. 6 quantitatively and qualitatively compares RTL2μSPEC with RTLCheck on their ability to support verification of the multi-V-scale's MCM implementation. Both charts feature the 56 evaluated litmus tests along the `x-axis` and verification times on the `y-axis`.

Fig. 6a effectively compares the combined performance of validating a μSPEC model and proving that the multi-V-scale will execute a given litmus test correctly. Recall that RTLCheck simultaneously proves that a given μSPEC model is correct with respect to input litmus test *and* that the litmus test will execute as required by MCM specification on the microarchitecture. These proof times are represented by the left (light gray or patterned) bar for each litmus test. However, likely due to the complexity of SVAs generated by RTLCheck, not all litmus tests can be fully verified. Incomplete proofs are noted with patterned bars. On the other hand, RTL2μSPEC synthesizes a complete μSPEC model in one step, proving that it is correct with respect to the microarchitecture by construction. This cost can then be amortized over the number of litmus tests evaluated on the final model using the Check tools. The upper right (dark gray) bars for each litmus test represent the amortized overhead (over 56 litmus tests) of synthesizing a multi-V-scale μSPEC model. Meanwhile, the lower right (black) bars represent the overhead of evaluating each of the 56 litmus tests on the synthesized μSPEC model with COATCheck. The average latency of RTLCheck for evaluating one of 56 tests (including incomplete proofs) is 5,786.63 seconds (1.61 hours). In contrast the average amortized lifting time and litmus test evaluation time and for RTL2μSPEC are 7.33 and 0.03 seconds, respectively, for a total of 7.36 seconds.

Fig. 6b compares the runtime of evaluating the multi-V-scale's MCM with respect to each of the 56 litmus tests using RTLCheck and RTL2μSPEC. RTLCheck optimizes the procedure of proving that a hardware design correctly executes a given litmus test when proofs about the correctness of a user-supplied μSPEC model are not required. Run time results for this optimized variant of RTLCheck are shown in bars on the left. Again, patterned bars signify incomplete proofs. The bars on the right, representing runtimes for litmus test evaluation with RTL2μSPEC, are identical to those in Fig. 6a but redrawn for clarity. Overall, RTLCheck spends an average of 1,507.81 seconds (25.13 minutes) proving that a given litmus test cannot exhibit MCM bugs when they run on the microarchitecture (including incomplete proofs), whereas the RTL2μSPEC approach can leverage a synthesized μSPEC model to conduct verification a single test in 0.03 seconds on average.

Besides its apparent performance and coverage benefits, we note that RTL2μSPEC is the first tool capably of synthesizing a complete correct-by-construct μSPEC model from RTL.

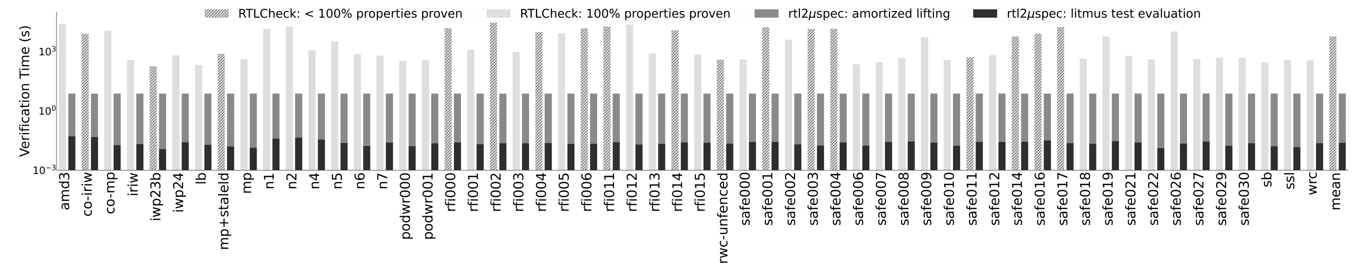
6.4 RTL2μSPEC Scope

In-Order, Out-of-Order, and Superscalar. RTL2μSPEC supports and has been evaluated on in-order processors. Theoretically, it can support a restricted class of out-of-order processors that do not speculate. Speculation violates the single-execution-path (§4.2.3) assumption. RTL2μSPEC can also handle superscalar designs, subject to the single-execution-path assumption. Such an in-scope design cannot feature multiple execution lanes for a single instruction type—this would directly violate assumption.

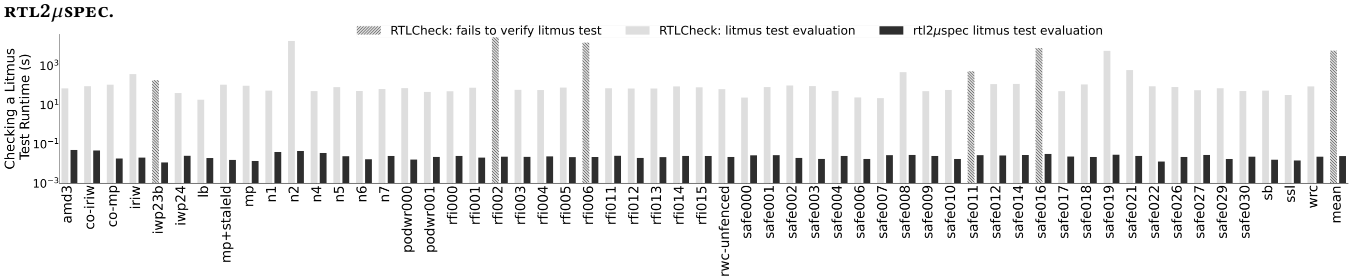
Single-Execution-Path and Single-Data-Source Assumptions. In addition to the single-execution-path assumption, RTL2μSPEC requires that designs feature a single data source per data-flow dependency that an instruction can be involved in (as the reader instruction, see §4.3.5)—the *single-data-source* assumption. A load whose read data can be sourced from either a store buffer *or* DRAM directly violates this. Handling designs that violate these constraints presently requires more user involvement; however, we think this is still an important advance over existing fully-manual approaches.

Memory Systems. Given the single-execution-path assumption, RTL2μSPEC cannot yet automatically identify cache structures which may be conditionally updated during the execution of an instruction (e.g., for cacheable versus uncacheable memory accesses). Regarding DRAM, RTL2μSPEC features no special restrictions outside of the request-response interface structure for remote state. Memory controllers are free to reorder requests. Multiple memory write ports and banked memories are also theoretically supported by RTL2μSPEC, but have not been evaluated.

User-Supplied Metadata. While RTL2μSPEC requires some designer-provided metadata to accompany the input design (§4.3.1 and §4.3.2), we expect annotations will be straightforward to provide, even with a complex design. In particular, many signals are likely to be involved in other standard property-based verification flows.



(a) Time to verify compliance of the multi-V-scale RTL with its μSPEC model on a per litmus test basis for RTLCheck (left), compared with the amortized time to synthesize a complete μSPEC model (upper right) plus the time to conduct litmus test verification (lower right) for RTL2μSPEC.



(b) Time to conduct litmus test-based MCM verification of the multi-V-scale using RTLCheck (left) versus a RTL2μSPEC-synthesized μSPEC model (right). Black bars are identical to those in (a).

Figure 6: Performance comparison (log scale) of RTL2μSPEC-assisted and RTLCheck [31]-based verification of hardware MCMs. Patterned bars represent incomplete proofs—instances where JasperGold returned *undetermined* or where a time out of 8 hours was reached.

Scalability. We cannot make definitive claims about the scalability of RTL2μSPEC, but have reasons to be optimistic. First, RTL2μSPEC generates highly localized properties which support low proof times with low variability. For example, RTL2μSPEC leverages the most recent reference ordering between a pair of instructions when instantiating HBI hypotheses as SVAs. This enables RTL2μSPEC to take advantage of RTL cut points that are already commonly used in commercial processor verification flows. Second, HBI hypotheses are independent and can be evaluated in fully in parallel. Finally, RTL2μSPEC's synthesis procedure features opportunities for optimization, like the elimination of redundant SVAs (§4.3.3 and §6.2).

7 RELATED WORK AND CONCLUSIONS

With minimal intervention, the RTL2μSPEC tool synthesizes an axiomatic description of hardware behavior—in the guise of a μSPEC model—from a Verilog design. To demonstrate its efficacy, we applied the tool to the multi-V-scale, thereby synthesizing a μSPEC model in 6.84 minutes. Subsequent verification of MCM litmus tests takes less than one second per test. Moreover, we identified a new, previously missed bug in the Verilog design of the V-scale.

Several tools are available for systematic litmus-based post-silicon testing of hardware, including `litmus` [3], `mcversi` [16], and `PerpLE` [34], and dedicated tools for GPU testing [40]. RTL2μSPEC, on the other hand, can be used to verify hardware before tape out.

The Check tools [24, 25, 30, 31, 33, 41, 42] are the most relevant prior work, especially RTLCheck. However, RTLCheck requires a user-provided μSPEC model, a processor implementation in Verilog, and a set of mappings from μSPEC primitives to signals in Verilog. In contrast, RTL2μSPEC only requires a Verilog implementation and modest design metadata. It also accomplishes a different goal—μSPEC model synthesis. ISA-Formal [37] checks RTL correctness by comparing states before and after the execution of an instruction against the machine readable definition of the Arm® Architecture [5]. In contrast to RTL2μSPEC, ISA-Formal does not verify the memory system and its concurrency implications.

We used the RISC-V V-scale in our case study for its simplicity and to ease the comparison with the RTLCheck tool, the current state-of-the-art. An obvious avenue for future work is applying our techniques to other processors—for example an Arm Cortex® design—which feature more complex microarchitectural features and also exhibit weak memory behaviors, in contrast to the multi-V-scale's strong consistency model. The Pipeproof [30] and Checkmate [41] tools could also be integrated with RTL2μSPEC. In the case of Pipeproof, this would allow us to conduct full proofs of MCM correctness, side-stepping litmus tests altogether. Checkmate, on the other hand, searches for security vulnerabilities in hardware designs using μhb analysis. Integrating both tools with RTL2μSPEC would allow them to work directly from source Verilog.

ACKNOWLEDGMENTS

We thank our shepherd and the anonymous reviewers for their helpful feedback. This work was supported by the National Science Foundation (under the grant CCF-2017863).

A ARTIFACT APPENDIX

A.1 Abstract

This artifact[8] uses RTL2μSPEC to produce a μSPEC model for the RISC-V multi-V-scale [29, 31], and COATCheck [25] to conduct formal MCM verification of the μSPEC model with respect to 56 litmus tests [31]. Overall, RTL2μSPEC requires two runtime environments:

- `rtl2uspecEnv`, where RTL2μSPEC runs as a C++ extension to Yosys
- `cadEnv`, where Jaspergold is installed and is able to evaluate RTL2μSPEC-generated SystemVerilog Assertions (SVAs).

A.2 Artifact check-list (meta-information)

- **Data set:**
  - *RISC-V multi-V-scale* SystemVerilog design for RTL2μSPEC to consume as input
  - *TCL and Python driver scripts* to support evaluation of RTL2μSPEC-generated SVAs on the multi-V-scale

  Data set components can be accessed here: https://github.com/yaohsiaopid/multicore_vscale_rtl2uspec_ae.git
- **Run-time environment:**
  Running RTL2μSPEC requires:
  - Symbiotic EDA Edition of Yosys
    Please contact edmund@symbioticeda.com and office@symbioticeda.com for academic license.
  - Cadence JasperGold

  Running the full end-to-end MCM verification case study featured in this artifact additionally requires:
  - COATCheck MCM verification tool (*included in container image*)

  To facilitate artifact evaluation, the compilation and execution environments for RTL2μSPEC, including RTL2μSPEC source code (https://github.com/yaohsiaopid/rtl2uspec), and COATCheck have been wrapped as a container image: yaohsiao/micro21:v0.2.3. The image requires the user to obtain Yosys access, as detailed below. *A run-time environment where JasperGold has been installed is required in addition to the container image.*
- **Output:**
  Given the multi-V-scale as input, RTL2μSPEC will produce a μSPEC model, called `vscale.uarch`, along with performance for various parts of the synthesis procedure. As a secondary output, COATCheck will produce qualitative and quantitative MCM verification results by indicating MCM compliance (or not) with sequential consistency (the multi-V-scale's MCM) and verification runtimes, respectively.

A.3 Installation

(1) **Setup the RTL2μSPEC execution environment.** (`rtl2uspecEnv`)
The below assumes that one has reached out to Symbiotica EDA and obtained instructions on how to download their software wrapped in a `tar.gz` file and a corresponding license file ending with `.lic`. Our artifact submission features a Docker image that includes all software dependencies, with the exception of JasperGold and Yosys, and requires users to provide the software and license file paths as mentioned (replace `<TARGZPATH>` and `<LICPATH>`). Run the commands as follows. **The last line should be executed within the container.**

```
$ export SYMBIOTIC=<TARGZPATH>
$ export SYMBIOTIC_LIC=<LICPATH>
$ docker run -itd --name microtest yaohsiao/micro21:v0.2.3
$ docker cp $SYMBIOTIC microtest:/home/symbiotic_bin.tar.gz
$ docker cp $SYMBIOTIC_LIC microtest:/home/symbiotic.lic
$ docker attach microtest
$ cd /home && . envsetup.sh
```

This step should end with the following result:

```
export PATH=/opt/symbiotic-20201202A-serp/bin:$PATH
export SYMBIOTIC_LICENSE=/home/symbiotic.lic
==================================
[success] yosys path is at /opt/symbiotic-20201202A-serp/
    bin/yosys
==================================
```

Path to multi-V-scale design: `/home/multicore_vscale_rtl2uspec`
Path to RTL2μSPEC: `/home/rtl2uspec`

(2) **Setup the JasperGold execution environment** (`cadEnv`):
- Confirm that JasperGold can be found in PATH
  `$ which jc`
- Install relevant python3 packages
  `$ yum install -y python3 && python3 -m pip install numpy pandas`
- Populate the multi-V-scale design

```
$ git clone https://github.com/yaohsiaopid/
      multicore_vscale_rtl2uspec_ae.git
    multicore_vscale_rtl2uspec &&
  mkdir multicore_vscale_rtl2uspec/gensva
```

A.4 Experiment workflow

(1) **Intra-instruction HBI synthesis.** In `rtl2uspecEnv`,
`$ cd /home/rtl2uspec && make init && make intra_hbi`
- `make init`: compiles RTL2μSPEC using source files located in `src_revised`. RTL2μSPEC's required user-provided design annotations are supplied as a header file, `src_revised/design.h`. For example, `src_revised/design.h` includes design information like the instruction fetch register (`IFR`) signal name, which is declared as a `string` type. The value of the `IFR string` is the hierarchical name in the RTL design of the state element that stores instructions when they are first fetched from instruction memory on a given core. For the multi-V-scale, the `IFR` is the `core_gen_block[0].vscale.pipeline.inst_DX` signal, and it is instantiated concretely in the multi-V-scale design files (`/home/multicore_vscale_rtl2uspec/src/main/verilog`).
  The `src_revised/design.h` header file is also used to specify which ISA instructions should have their behavior formalized and included in the final μSPEC model. This is done by enumerating (`opcodes_name, valid_exe_condition`) pairs, where `opcodes_name` is a `string` name for an instruction of interest, and `valid_exe_condition` describes the how to recognize the instruction of interest from its binary encoding. Given the focus of our paper is on extracting μSPEC models for conducting MCM verification, `src_revised/design.h` specifies two relevant ISA instructions for the multi-V-scale: `sw` (appears first, and thus will be referred to with ID 0 by RTL2μSPEC) and `lw` (appears second, and thus will be referred to with ID 1 by RTL2μSPEC).
- `make intra_hbi`: runs CDFG analysis over the Verilog design supplied in `script/multicore_yosys_verific.tcl`, namely the multi-V-scale located at `/home/multicore_vscale_rtl2uspec` in this artifact evaluation. CDFG analysis identifies the set of state elements that are reachable from the user-supplied `IFR` in the input design's netlist and generates corresponding intra-instruction HBI hypotheses in the form of SVAs. These SVAs are output into the folder `build/sva/intra_hbi/`. Metadata files `ever_update_[0-9]+.txt` for each instruction type list relevant state elements to be considered for inclusion in the instruction's execution path, pending the outcome of SVA evaluation. SVAs corresponding to an instruction metadata file can be found in a `ever_update_[0-9]+.sv` file with the same integer

[8] Official artifact can be found here: https://doi.org/10.5281/zenodo.5492990.

ID. These integer IDs match the order in which instructions were enumerated in the src_revised/design.h file. The result should be

```
build/sva/intra_hbi/
|-- ever_update_0.sv
|-- ever_update_1.sv
 -- .... several other files
```

(2) **Intra-instruction HBI hypothesis evaluation.**
- Copy the folder /home/rtl2uspec/build/sva/intra_hbi/ in rtl2uspecEnv to cadEnv under multicore_vscale_rtl2uspec/gensva/.
- Evaluate SVAs in cadEnv:
  $ python3 revised_script/intra_hbi.py
  The script invokes JasperGold to evaluate the SVA files in the folder and, based on the results (proven/cex), generates a modified version of metadata file ever_update_[0-9]+.txt, called ever_update_[0-9]+.txt.res.
  This file features a new field for each row (updated/fixed), which indicates whether the instruction of interest (denoted by the file ID) does/does not update the state element of interest (denoted by a row of the file).
  Upon termination of SVA evaluation, the script prints out total number of SVAs evaluated and the total runtime, **which should match the first two rows of the Intra-Instr. column in Fig. 5 in the paper.**

```
=================================================
Total time on intra-instruction HBI (sec) : 271.063000
Total number of SVA evaluated: 105
=================================================
```

- Copy the folder multicore_vscale_rtl2uspec/gensva /intra_hbi from cadEnv back to rtl2uspecEnv to replace original folder /home/rtl2uspec/build/sva/intra_hbi/ so that rtl2uspecEnv has the updated metadata files.

(3) **Inter-instruction HBI synthesis.** In rtl2uspecEnv,
$ cd /home/rtl2uspec && make inter_hbi
Based on the results from previous step (intra-instruction HBI evaluation), this step deduces per-instruction DFGs, and iterates over all pairs of per-instructions DFGs to generate all inter-instruction hypotheses. The result of inter-instruction HBI synthesis will be stored in build/sva/inter_hbi/ and be structured as follows:

```
gensva/
|-- inter_hbi
|   |-- 0.sv
|   |-- 1.sv
|-- |-- .... several other files
|   |-- hbi_meta.txt
|    -- hbi_meta.txt.detail
 -- intra_hbi
    |-- .... several other files
```

hbi_meta.txt.detail listed all generated inter-instruction HBI hypotheses (one per row) that will be evaluated along with their corresponding SVA file (in the file_# field of the list). One of the rows in hbi_meta.txt.detail should look like the following to indicate this hypothesis is validated by the SVA contained in 0.sv.

```
file_#,hbi_type,samecore,i0_type,i1_type,i0_loc,i1_loc,...
0,0,1,0,0,core_gen_block[0].vscale.pipeline.ctrl....
```

hbi_meta.txt contains metadata pertaining to all unique SVAs that will be used to validate all inter-instruction HBI hypotheses.

(4) **Inter-instruction HBI hypothesis evaluation.**
- Copy the folder /home/rtl2uspec/build/sva/inter_hbi/ in rtl2uspecEnv to cadEnv under multicore_vscale_rtl2uspec/gensva/.
- Evaluate SVAs cadEnv:
  $ python3 revised_script/inter_hbi.py
  As in intra-instruction HBI evaluation, this script invokes JasperGold for each SVA files in the inter_hbi/. Based on the results (proven/cex) a modified version of hbi_meta.txt, called hbi_meta.txt.res, is generated, which includes a new field for each row (updated/fixed). As before, the script prints out total number of SVAs evaluated and the total runtime, **which should match to first two rows of the Inter-Instr. column of Fig. 5 in the paper.**

```
============================================
        (Spatial)|  (Temporal)|   Dataflow|
cnt             1|          12|          2|
time     5.347000|   31.632000|  15.801000|
============================================
```

- Copy the folder multicore_vscale_rtl2uspec/gensva/ inter_hbi from cadEnv back to /home/rtl2uspec/build/ sva/inter_hbi/ in rtl2uspecEnv. rtl2uspecEnv should now have new files, namely /home/rtl2uspec/build/sva/inter_hbi/hbi_meta.txt.res

(5) **μSPEC generation.** In rtl2uspecEnv,
$ cd /home/rtl2uspec && make uspec .
This pass aggregates the results from previous steps, merges state elements having the same ordering behaviors into "mega-nodes," and generates the final μSPEC model, named vscale.uarch. The mega-nodes will be instantiated as single nodes during instruction execution path enumeration in the μSPEC model. Part of this pass also includes a syntatic translation of the proven HBI hypotheses to the μSPEC DSL. An excerpt of the μSPEC model generated by our artifact evaluation is included below for reference.

```
StageName 0 "IF_".
StageName 1 "mgnode_2".
StageName 2 "mgnode_0".
StageName 3 "hasti_mem_mem".
StageName 4 "mgnode_3".
StageName 5 "mgnode_1".

% ProgramOrder
Axiom "PO_man": forall microop "i1",  forall microop "i2",
    SameCore i1 i2 => ProgramOrder i1 i2 =>
      AddEdge ((i1, IF_), (i2, IF_), "PO", "orange").
```

A.5 Evaluation and expected results

Our artifact evaluates the synthesized μSPEC model against a suite of litmus tests using the COATCheck MCM verification tool. In rtl2uspecEnv,
$ cd /home/rtl2uspec && make eval_uspec
This step obtains a suite of litmus tests [31] to evaluate compliance of a μSPEC model with Sequential Consistency (the MCM of the multi-V-scale). It then uses COATCheck to evaluate the RTL2μSPEC-generated μSPEC model against these same litmus tests. An example of the results that should be generated is shown below. Each row features the name of a litmus test and the runtime (ms). Runtimes correspond to **blue performance bars Fig. 6 of the paper. The final line of output should also indicate that none of the litmus tests fail to execute in a Sequentially Consistent manner**, demonstrating that COATCheck has proven the multi-V-scale to implement Sequential Consistency with respect to the litmus tests considered.

```
.....
safe027.test,29.083897
safe029.test,16.207506
safe030.test,22.950519
sb.test,11.006003
ssl.test,16.676122
wrc.test,23.565418
--- 1379.073456 ms ---
======= ALL TESTS PASSES =======
```

REFERENCES

[1] 2009. Institute of electrical and electronic engineers (IEEE) standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language.

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. *Proceedings of the* 22nd *International Conference on Computer Aided Verification (CAV)* (2010). http://dx.doi.org/10.1007/978-3-642-14295-6_25

[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. *Proceedings of the* 17th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2011).

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 7:1–7:74.

[5] Arm. 2013. Arm Architecture Reference Manual.

[6] Arm. 2021. The Arm memory model tool. https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool Accessed 12th April 2021.

[7] Krste Asanović. 2017. The RISC-V Memory Consistency Model. *RISC-V Organization* (2017). https://riscv.org/2017/04/risc-v-memory-consistency-model/

[8] Verific Design Automation. 2019. Verific's Parser Platform.

[9] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series).* The MIT Press.

[10] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. 2009. Satisfiability modulo theories. In *Handbook of Satisfiability.* 825–885.

[11] James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. *Proceedings of the* 38th *Conference on Programming Language Design and Implementation (PLDI)* (2017).

[12] Cadence Design Systems, Inc. [n.d.]. Cadence JasperGold formal verification platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html Accessed 12th April 2021.

[13] Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the Arm Weakly Consistent Memory Model. In *Proceedings of the ACM SIGPLAN workshop on memory systems performance and correctness (MPSC).* 16–19.

[14] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. *Model Checking.* MIT Press.

[15] Francisco Corella, James M. Stone, and Charles Barton. 1993. A formal specification of the PowerPC shared memory architecture. *Technical Report Computer Science Technical Report RC 18638(81566), IBM Research Division, T.J. Watson Research Center* (1993).

[16] M. Elver and V. Nagarajan. 2016. McVerSi: A test generation framework for fast memory consistency verification in simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 618–630.

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

[18] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. 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).* 635–646.

[19] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. 2020. TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests. *Proceedings of the* 47th *International Symposium on Computer Architecture (ISCA)* (2020).

[20] IBM. 2013. Power ISA Version 2.07.

[21] Intel Corporation. 2007. Intel 64 architecture memory ordering white paper.

[22] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[23] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computing* 28, 9 (1979), 690–691.

[24] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2014. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. *Proceedings of the* 47th *International Symposium on Microarchitecture (MICRO)* (2014).

[25] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. *Proceedings of the* 21st *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[26] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. *Proceedings of the* 22nd *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[27] Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. 2010. Generating Litmus Tests for Contrasting Memory Consistency Models. *22nd International Conference on Computer Aided Verification (CAV)* (2010).

[28] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. *Proceedings of the* 24th *International Conference on Computer Aided Verification (CAV)* (2012).

[29] Albert Magyar. 2016. A Verilog implementation of the RISC-V Z-scale microprocessor. https://github.com/ucb-bar/vscale.

[30] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. 2018. PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications. *Proceedings of the* 51st *International Symposium on Microarchitecture (MICRO)* (2018).

[31] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLCheck: Verifying the Memory Consistency of RTL Designs. *Proceedings of the* 50th *International Symposium on Microarchitecture (MICRO)* (2017).

[32] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLCheck: Verifying the Memory Consistency of RTL Designs. https://github.com/ymanerka/rtlcheck.

[33] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: Using μhb Graphs to Verify the Coherence-consistency Interface. *Proceedings of the* 48th *International Symposium on Microarchitecture (MICRO)* (2015).

[34] Themis Melissaris, Markos Markakis, Kelly Shaw, and Margaret Martonosi. 2020. PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[35] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. *Proceedings of the* 22nd *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)* (2009).

[36] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying Arm Concurrency: Multicopy-atomic Axiomatic and Operational Models for Armv8. *ACM Programming Languages* (2017).

[37] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Arm® Processors with ISA-Formal. In *Proceedings of the* 28th *International Conference on Computer Aided Verification (CAV).*

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

[39] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[40] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the* 37th *Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 100–113.

[41] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. *Proceedings of the* 51st *International Symposium on Microarchitecture (MICRO)* (2018).

[42] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. *Proceedings of the* 22nd *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[43] Srikanth Vijayaraghavan and Meyyappan Ramanathan. 2014. *A Practical Guide for SystemVerilog Assertions.* Springer Publishing Company, Incorporated.

[44] Andrew Waterman and Krste Asanović (Eds.). 2018. *The RISC-V Instruction Set Manual Volume I: User-level ISA.* RISC-V International. Document version 2.2.

[45] John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. 2017. Automatically comparing memory consistency models. *Proceedings of the* 44th *Symposium on Principles of Programming Languages (POPL)* (2017).

[46] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys: a free Verilog synthesis suite. In *Proceedings of the* 21st *Austrian Workshop on Microelectronics (Austrochip).*

[47] Y. Yang, Ganesh Gopalakrishnan, G. Lindstrom, and K. Slind. 2004. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In *Proceedings of the* 18th *International Parallel and Distributed Processing Symposium.* 31–.