RTLCheck: Verifying the Memory Consistency of RTL Designs

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

ABSTRACT

Paramount to the viability of a parallel architecture is the correct implementation of its memory consistency model (MCM). Although tools exist for verifying consistency models at several design levels, a problematic verification gap exists between checking an abstract microarchitectural specification of a consistency model and verifying that the actual processor RTL implements it correctly.

This paper presents RTLCheck, a methodology and tool for narrowing the microarchitecture/RTL MCM verification gap. Given a set of microarchitectural axioms about MCM behavior, an RTL design, and user-provided mappings to assist in connecting the two, RTLCheck automatically generates the SystemVerilog Assertions (SVA) needed to verify that the implementation satisfies the microarchitectural specification for a given litmus test program. When combined with existing automated MCM verification tools, RTLCheck enables test-based full-stack MCM verification from high-level languages to RTL. We evaluate RTLCheck on a multicore version of the RISC-V V-scale processor, and discover a bug in its memory implementation. Once the bug is fixed, we verify that the multicore V-scale implementation satisfies sequential consistency across 56 litmus tests. The JasperGold property verifier finds complete proofs for 89% of our properties, and can find bounded proofs for the remaining properties.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**;
• **Hardware → Functional verification**; **Assertion checking**;

KEYWORDS

Memory consistency models, automated verification, RTL, SVA

ACM Reference format:
Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLCheck: Verifying the Memory Consistency of RTL Designs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3124536

1 INTRODUCTION

Memory consistency models (MCMs) specify the values that can be legally returned by load instructions in a parallel program, and hence they form a critical and fundamental part of the specification of any shared-memory architecture. An ISA-level MCM serves as both a target for compilers and a specification for hardware to implement. Weaker ISA-level MCMs allow a variety of reorderings of memory operations in an effort to improve performance [6, 25], while implementations of stronger MCMs may use complicated speculation to retain performance while still enforcing stronger ordering [9, 15]. Regardless of the choice of ISA-level MCM, a hardware implementation of a multicore architecture must satisfy its MCM in *all possible cases*, or the correct execution of parallel programs on that architecture cannot be guaranteed.

With concurrent systems becoming both more prevalent and more complex, MCM-related bugs are more common than ever. Intel processors have experienced two transactional memory bugs in recent years [23, 52]. In another case, ambiguity about the ARM ISA-level MCM specification led to a bug where neither hardware nor software was enforcing certain orderings [4, 5]. With no cost-effective hardware fix available, ARM had to resort to a low-performance compiler fix instead. MCM-related issues have also surfaced in concurrent GPU programs [3, 45]. Finally, the computer security community has recognized that MCM bugs may lend themselves to security exploits [22].

Verifying an MCM's correct specification and implementation is increasingly critical given its fundamental importance. System verification is a general challenge, with verification costs now dominating total hardware design cost [20]. MCM verification is particularly challenging since it requires analysis across many possible interleavings of events. Testing all possible combinations of inputs to a system is infeasible, and dynamic testing of a design in simulation will by definition be incomplete and not capture all possible interleavings, even for the tested programs.

True completeness in MCM verification would entail spanning the stack from high-level languages (HLLs), through compiler mappings, the ISA, and ultimately to underlying hardware implementations. Prior work has enabled tractable automated memory model verification for provided litmus tests[1] from HLLs to axiomatic microarchitectural specifications [4, 31, 32, 35, 49], but not deeper down to RTL.

Meanwhile, more generic non-MCM RTL verification is also maturing [41, 47, 48], often based on languages for specifying temporal assertions such as Property Specification Language (PSL) [26] and SystemVerilog Assertions (SVA) [27]. There remains, however, a gap between early-stage microarchitectural MCM verification [31] and lower-level RTL verification. If *axiomatic* microarchitectural MCM verification could be linked to the *temporal* verification approaches of RTL, this link would allow full-stack HLL to RTL checking of a hardware and software system's memory ordering behavior. It

[1] Small programs used to test MCM implementations.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124536

would also encourage the use of early-stage (e.g., microarchitecture-level) MCM verification techniques by enabling and facilitating their subsequent testing against RTL designs when available.

To fill the MCM verification gap between microarchitecture and RTL, this paper proposes RTLCheck, a methodology and tool for checking that microarchitecture-level axioms are upheld by underlying RTL across a suite of litmus tests. Given an axiomatic microarchitectural specification, an RTL design, and a mapping between the axiomatic representation's abstract primitives and the corresponding RTL signals and values, RTLCheck automatically generates SystemVerilog Assertions (on a per-test basis) and inserts them into the RTL design. These assertions lower the abstract microarchitectural axioms into concrete temporal assertions and assumptions that help establish that the design's memory ordering behavior meets the microarchitectural specification. RTLCheck then uses JasperGold [12] (a commercial RTL verification tool from Cadence Design Systems, Inc.) to check these assertions. The results of this verification say whether the asserted properties have been proven for a given test, whether they have been proven for the test up to a bounded number of cycles, or if a counterexample (execution trace that does not satisfy the property) has been found. If a counterexample is found, a discrepancy exists between the microarchitectural specification and RTL, likely due to a bug in the implementation. RTLCheck's approach is incomplete in the sense that it only ensures that the litmus tests verified actually run correctly, rather than checking all programs. In other words, it is possible for a bug to still exist in the design even if the design has been verified across a suite of litmus tests. Nonetheless, automated litmus test-based approaches like RTLCheck have the benefit of concentrating verification effort on the scenarios most likely to exhibit bugs, and are viewed as effective and efficient tools commonly included in MCM verification approaches [4, 24, 31].

This paper demonstrates RTLCheck's usage on a multicore version of the RISC-V V-scale open-source processor design[2] [42]. In doing so, we discover a bug in the V-scale processor's memory implementation. After fixing the bug, we use RTLCheck to show that our multicore V-scale RTL satisfies a set of microarchitectural axioms that are sufficient to guarantee sequential consistency for 56 litmus tests. Overall, the contributions of our work are as follows:

- We present an automated flow from *axiomatic* microarchitectural ordering specifications to *temporal* assertions about RTL design operation for a variety of litmus tests. The automated axiomatic-to-temporal translation is complicated by the inherent differences between the two approaches. The generated assertions are inserted into RTL to support MCM verification of the RTL, narrowing the verification gap between microarchitecture-level MCM checkers and underlying RTL.
- Our work demonstrates that using RTLCheck's automatically generated assertions in RTL verification can be efficient and tractable. In particular, RTLCheck's assertion and assumption generation phase takes just seconds. The subsequent property verifier discovers complete proofs (i.e. true for all possible traces of a given litmus test) for 89% of the generated SVA properties in 11 hours of runtime, and can generate bounded proofs (i.e. true for all possible test traces up to a certain number of cycles) for the remaining properties. RTLCheck can also be used for iterative verification, allowing implementers to refine their design and its specification with respect to meeting MCM requirements.
- Our method is the first technique for RTL MCM checking that applies generally to an *arbitrary* Verilog design. Other work has formally proven the MCM correctness of specific RTL designs [51] for all programs. Our work is complementary: it proves correctness for litmus tests rather than all programs, but does not impose restrictions on the RTL design to be verified. Furthermore, it supports *arbitrary* ISA-level MCMs, including ones as sophisticated as x86-TSO, ARM, and IBM Power [6, 25, 28].
- With the link from microarchitecture to RTL covered by RTLCheck, the Check suite [33] can now support MCM verification from HLLs (C11, etc.) through compiler mappings, the OS, ISA, and microarchitecture, all the way down to RTL.

[2]V-scale was deprecated in the time between this work's submission and publication [34], but remains an interesting case study.

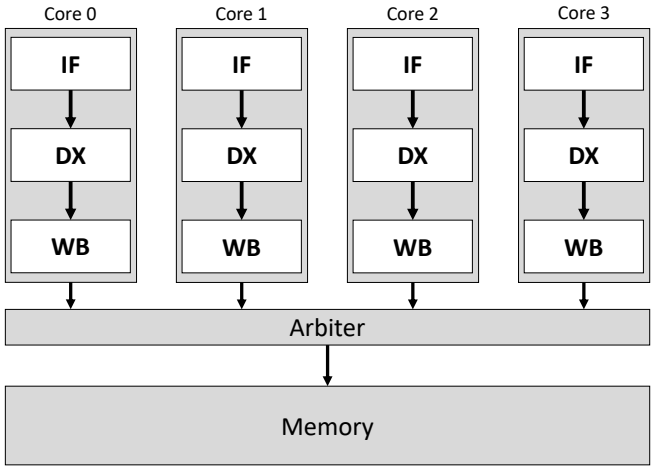


Figure 1: The Multi-V-scale processor: a simple multicore processor with four three-stage in-order pipelines. The arbiter allows only one core to access memory at a time.

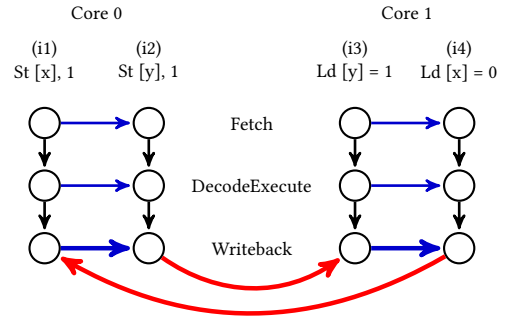
Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

Figure 2: Code for litmus test `mp`

2 MOTIVATING EXAMPLE

2.1 Microarchitectural Verification Background

Figure 1 shows the Multi-V-scale processor, a simple multicore where each core has a three-stage in-order pipeline. Instructions in these pipelines first go through the Fetch (IF) stage, then a combined Decode-Execute (DX) stage, and finally a Writeback (WB) stage where data is returned from memory (for loads) or sent to memory (by stores). An arbiter enforces that only one core can access data memory at any time. The read-only instruction memory



(a) μhb graph for the SC-forbidden outcome of Figure 2's `mp` litmus test on Figure 1's processor. The cycle in this graph shows that this scenario is correctly unobservable at the microarchitecture level.

```
Axiom "WB_FIFO":
forall microops "a1", "a2",
(OnCore c a1 /\ OnCore c a2 /\
  ~SameMicroop a1 a2 /\ ProgramOrder a1 a2) =>
EdgeExists((a1,DX)), (a2,DX))) =>
AddEdge((a1,WB)), (a2,WB))).
```

(b) Axiom expressing that the WB stage should be FIFO.

```
always @(posedge clk) begin
  if (reset | (stall_DX & ~stall_WB)) begin
     // Pipeline bubble
     PC_WB <= `XPR_LEN'b0;
     store_data_WB <= `XPR_LEN'b0;
     alu_out_WB <= `XPR_LEN'b0;
  end else if (~stall_WB) begin
     //Update WB pipeline registers
     PC_WB <= PC_DX;
     store_data_WB <= rs2_data_bypassed;
     alu_out_WB <= alu_out;
     csr_rdata_WB <= csr_rdata;
     dmem_type_WB <= dmem_type;
  end
end
```

(c) Verilog RTL responsible for updating WB pipeline registers.

Figure 3: Illustration of the verification gap between microarchitectural axioms and underlying RTL. The axiom in (b) states that the processor WB stage should be FIFO. Sets of such axioms can be used to enumerate families of μhb graphs such as the one in (a). RTLCheck translates axioms such as (b) to equivalent temporal properties at RTL. These properties can be verified to ensure that Verilog such as that in (c) upholds the microarchitectural axioms for a given test.

(not shown) is concurrently accessed by all cores. This processor is simple enough that it appears to implement sequential consistency[3] (SC) [29]. As a case study in this paper, our goal is to check that its RTL (Verilog) correctly satisfies SC for the litmus tests we examine. (Section 5 provides further details about the Multi-V-scale design.)

One litmus test to check for SC is the "message-passing" (`mp`) litmus test shown in Figure 2. A multiprocessor that implements SC will forbid the outcome of `r1 = 1, r2 = 0` for this code, as there is no SC order on memory operations that allows this outcome. Our approach needs to check that a provided RTL design will never produce executions exhibiting this forbidden outcome.

The Check suite [33] has developed methods for conducting microarchitectural MCM verification by exhaustively enumerating and checking microarchitectural happens-before (μhb) graphs that represent all possible executions for a given litmus test [31, 32, 35, 49]. RTLCheck extends the microarchitectural verification of μhb graphs to the level of RTL.

Figure 3a shows an example μhb graph for the `mp` litmus test running on two cores, each with the three-stage pipeline mentioned above. Each column of nodes represents a different instruction in the litmus test. The left two columns correspond to i1 and i2 executing on core 0. The right two columns correspond to i3 and i4 executing on core 1. Nodes in μhb graphs represent the individual microarchitectural events or pipeline stages in a particular instruction's execution on that microarchitecture. For example, the node in the bottom-right of the graph represents instruction `i4` at its Writeback stage, while the left-most node in the second row represents instruction `i1` at its Decode-Execute stage.

Edges between μhb nodes represent known happens-before relationships between those nodes, and are added based on ordering axioms that the designer specifies a correct microarchitecture should respect. As each edge represents a happens-before relationship, a cycle in the graph indicates that the depicted scenario *cannot occur*. This is because a cycle would imply that an event must happen before itself, which is impossible. Thus, the strategy with verification based on μhb graphs is to consider and cycle-check all possible scenarios. If the specification indicates that a particular outcome should be forbidden, then all μhb graphs for that test outcome on that microarchitecture must be cyclic.

Many μhb graphs are typically possible for each litmus test and microarchitecture, since parallel programs generally have many possible executions. Ordering rules specifying which edges are added and when are described in terms of *axioms* such as the one in Figure 3b. This axiom is written in μspec, a first-order logic-based modeling language used by the Check suite [33]. This axiom states that if the DX stage of an instruction `a1` happens before the DX stage of an instruction `a2` that is later in program order on the same core, then the WB stage of `a1` must also happen before the WB stage of `a2`.

For `mp`, under SC, an outcome of `r1 = 1, r2 = 0` should be forbidden on the processor from Figure 1. The microarchitectural analysis to verify this is performed as follows. First, for this processor's in-order pipelines, instructions proceed through the DX and WB stages in the order in which they were fetched. This behavior is represented by edges in the graph, including those between the WB stages of `i1` and `i2`, as well as between the WB stages of `i3` and `i4` (both added by the `WB_FIFO` axiom in Figure 3b). Other axioms can specify other ordering relationships that the designer stipulates for the implementation. For example, in order for `i4` to return a value of 0 for its load of `x`, it must complete its Writeback

[3] In SC, execution results are consistent with there being a total order on all memory operations where each load returns the value of the last store to the same address in this total order.

stage before the store of x in i1 reaches its Writeback stage and goes to memory, thus overwriting the old value of 0. This ordering is represented by the red edge from i4's Writeback node to i1's Writeback node in Figure 3a. Likewise, in order for i3's load of y to return a value of 1, it must occur after the store of y in i2 reaches the Writeback stage, shown by another red edge between those two nodes. The combination of the four thicker red and blue edges creates a cycle in the graph. Thus, this execution of mp is (correctly) not observable.

2.2 From Microarchitectural Models to RTL

The microarchitectural verification conducted by tools such as the Check suite is only valid if each of the individual ordering axioms is actually upheld by the underlying RTL. Thus, there is a need to verify that these axioms *are* respected by the RTL of the design. If they are not, then any microarchitectural verification assumes incorrect orderings and is invalid.

The key contribution of RTLCheck lies in verifying (on a per-test basis) that a given RTL design actually upholds the microarchitecture-level ordering axioms as specified. This verification is difficult due to the semantic mismatches between axiomatic μspec microarchitecture definitions and the temporal assertions used to verify RTL. Section 3 describes this semantic mismatch in greater detail, and then Section 4 discusses our approaches for overcoming these challenges.

A key contribution of this paper is to identify an approach to writing μspec that is "synthesizable" to SVA, much as previous work has spent effort to identify subsets of Verilog that are synthesizable to actual circuits. We expect that in the future, μspec axioms will become more tailored to restrict themselves more carefully to rules that are in fact synthesizable to SVA.

Returning to our example, Figure 3c shows the portion of the processor's Verilog RTL that updates the WB pipeline registers from the DX pipeline registers. The Verilog makes it appear as though instructions do indeed move to WB in the order in which they entered DX, but in general, it is challenging to tell from inspection whether this is *always* the case. For instance, what if an interrupt occurs between the two instructions? The axioms to be verified can be notably more complex than the one in Figure 3b. In such cases, without automated verification like RTLCheck, it is harder, if not impossible, to tell whether they are always upheld for a given test.

3 THE SEMANTIC GAP BETWEEN μhb GRAPHS AND TEMPORAL LOGIC

Axiomatic and temporal models can both be used to verify that systems correctly satisfy MCMs, but they do so in starkly different ways. These differences lead to challenges when translating between the two, which RTLCheck needs to do in its translation of μspec to SVA. This section highlights the challenges faced when translating between the axiomatic world of μspec and the temporal world of SVA. We begin by providing some background on the differences between axiomatic and temporal approaches.

3.1 Axiomatic/Temporal Differences

Consider an abstract machine atomic_mach which performs instructions atomically and in program order. Figure 4 shows the verification of the mp litmus test from Figure 2 on atomic_mach using both axiomatic (Figure 4a) and temporal (Figure 4b) models of the machine. This verification aims to check whether mp's forbidden outcome (r1=1,r2=0) is possible on atomic_mach.

Axiomatic verification conceptually generates all possible executions and then checks each one for correctness. In the case of mp, there are four possible executions, each with a different outcome, as shown in Figure 4a. To check an execution for correctness, each axiom is applied to the execution as a whole. An axiom has *omniscience* of the execution it is being applied to; in other words, it can see all events in the execution. The axiom for SC on atomic_mach checks that an execution does not contain cycles in the combination of the po (program order), rf (reads-from), fr (from-reads), and co (coherence order) relations. Figure 4a shows such a cycle in the execution at its bottom-right, violating the axiom. The cycle indicates that this execution is impossible on atomic_mach. Figure 4a depicts this by a blue strike through the execution.

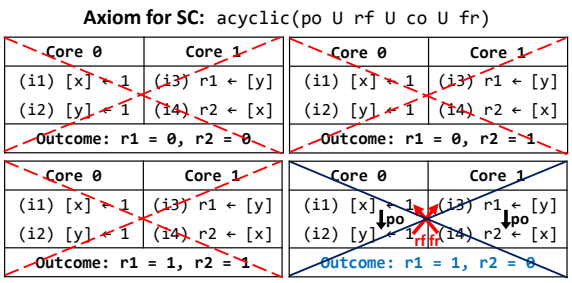
Since axiomatic approaches examine executions as a whole, they can also efficiently reason about the outcome of an execution. This makes it simple to exclude executions on the basis of their outcome. The remaining three executions in Figure 4a have outcomes distinct from the forbidden outcome under test (r1=1,r2=0), so they can be excluded from consideration when checking for the presence of the forbidden outcome. This exclusion is shown in Figure 4a by the dashed red strikes through these executions. There are no remaining executions that satisfy both the axiom for SC and the outcome under test. This indicates that the forbidden outcome of mp is correctly unobservable on atomic_mach.

Temporal approaches, meanwhile, conceptually generate and check executions step-by-step. As Figure 4b shows, temporal verification generates all possible first steps from a start state (the black dot). From each valid first step that is generated, the verifier generates all possible next steps, and so on, generating a "tree" of executions. For atomic_mach in Figure 4b, a step constitutes performing a single instruction atomically, while at the level of RTL, a single step is a clock cycle.

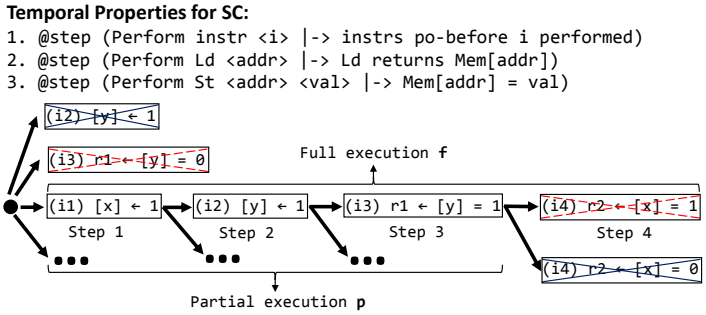
Executions correspond to paths from the start state through the tree. These paths can represent partial executions of the test (such as execution p in Figure 4b), which perform only some of the instructions of the test. They can also represent full executions of the test which perform all test instructions (such as execution f in Figure 4b).

The temporal properties required for SC on atomic_mach are listed in Figure 4b, and are noticeably different from the corresponding axiomatic description. Together, the three properties enforce that instructions perform in program order, and that loads read the value of the last store to their address. Temporal properties are specified in terms of steps rather than executions, and are thus applicable to *both* partial and full executions. If a step generated by the execution tree contravenes one of the properties of the model, that branch of the tree is eliminated as being impossible on atomic_mach. For instance, performing i2 from the start state violates property 1 in Figure 4b, as indicated by the corresponding blue strike-through.

Unlike in the axiomatic case, filtering outcomes is quite difficult for temporal methods. Temporal assumptions can be used to constrain the explorations of a temporal verifier to the outcome



(a) Axiomatic verification of `mp` on the abstract machine `atomic_mach`. All possible executions are generated, and then each is checked as a whole. Executions with an outcome not under test are eliminated (dashed red strikes), as are executions that violate axioms (blue strikes).



(b) Temporal verification of `mp` on the abstract machine `atomic_mach`. Executions are generated and checked step-by-step as a "tree". Steps exhibiting an outcome not under test are eliminated (dashed red strikes), as are steps that violate properties (blue strikes). `|->` denotes temporal implication.

Figure 4: Illustration of the differences between axiomatic and temporal approaches when used to check for the forbidden outcome of `mp (r1=1,r2=0)` on the abstract machine `atomic_mach`, which performs instructions atomically and in program order. The forbidden outcome is found to be unobservable in both cases.

under test, but the step-by-step generation of executions makes it difficult for the verifier to check whether a given step causes the future violation of an assumption. Consider a temporal assumption for `mp` which enforces that the load of `x` returns 0 as per the outcome under test. Performing the store to `x` in `i1` as step 1 makes it impossible for the load of `x` in `i4` to return 0 as required by the assumption. (Doing so would now violate property 2.) However, the only way the temporal verifier can (conceptually) deduce this is to examine all possible paths from `i1` to the end of the execution, and check if any of them contain a step where `i4` returns 0. This check is computationally intensive in the general case. Due to this cost, many SVA verifiers (including JasperGold, the verifier we use in this paper) do *not* check for the future violation of assumptions in this manner, contrary to actual SVA requirements[4]. Instead, they only guarantee that assumptions are not violated up to the present step.

Returning to `atomic_mach`, the lack of a check for future violation of assumptions means that executions *cannot* be filtered by test outcome. Instead, branches of the execution tree that violate assumptions are only removed from consideration after the offending event actually occurs. For example, in the execution tree of Figure 4b, the branch where the load of x in `i4` returns 1 is only eliminated when `i4` actually occurs at step 4 (as indicated by the corresponding dashed red strike-through), even though this outcome is enforced by the prior occurrence of the store of 1 to `x` at step 1. Meanwhile, the step where `i4` returns 0 in the displayed branch is impossible as it violates property 2 from Figure 4b. Since there is no complete execution that satisfies both the properties of the model and the outcome under test, the temporal verification deduces that the forbidden outcome of `mp` is correctly unobservable on `atomic_mach`. However, any properties of the model *must* still match partial executions like `p`, despite the fact that `p` cannot be extended to a full execution that satisfies both the model and the outcome under test.

The rest of this section explains the specific semantic mismatches between axiomatic μspec and temporal SVA that are relevant to RTLCheck, and the following section explains how RTLCheck surmounts these challenges.

3.2 μspec Omniscience and Load Values

Figure 5 shows a μspec axiom (and related macros) from the Multi-V-scale microarchitecture definition. This axiom splits the checking of load values into two categories: those which read from some write in the execution, and those which read from the initial value of the address in memory. The axiom is litmus-test-independent: it applies equally to any program that runs on the microarchitecture being modeled. However, for efficiency, the microarchitectural verification of the Check suite [31] uses *omniscience* about a proposed litmus test outcome to prune out all logical branches of the axiom which do not result in that outcome. Consider the application of this axiom to the load of `x` from `mp`, which returns 0 in the outcome under test. In μspec, the `SameData w i` predicate evaluates to true if instructions `w` and `i` have the same data in the litmus test outcome, while `DataFromInitialStateAtPA i` returns true if `i` reads the initial value of a memory location. For `mp`, the Check framework considers the entire execution at once (like the axiomatic verifier in Figure 4a), and evaluates all data-related predicates (highlighted in red in Figure 5) according to the outcome specified by the litmus test. For the load of `x`, the Check suite evaluates the `SameData w i` predicate in the `NoInterveningWrite` part of the axiom to false (as there is no write which stores 0 to `x` in `mp`). This causes the `NoInterveningWrite /\ BeforeOrAfterEveryWrite` portion of the `Read_Values` axiom to evaluate to false. Thus, the body of the `Read_Values` axiom is reduced to `BeforeAllWrites`, and Check knows from the start that it must add an edge indicating that `Ld x @WB` $\xrightarrow{hb}$ `St x @WB` (shown as one of the red edges in Figure 3a).

[4]Checking for future violation of SV assumptions requires complicated liveness checks even for simpler safety properties. See Cerny et al. [14] for further details.

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
    (IsAnyWrite w /\ SameAddress w i /\ ~SameMicroop i w) =>
    AddEdge ((i, Writeback), (w, Writeback), "fr", "red")).

DefineMacro "NoInterveningWrite":
  exists microop "w", (
    IsAnyWrite w /\ SameAddress w i /\ SameData w i /\
    EdgeExists ((w, Writeback), (i, Writeback)) /\
    ~(exists microop "w'",
      IsAnyWrite w' /\ SameAddress i w' /\ ~SameMicroop w w' /\
      EdgesExist [((w , Writeback), (w', Writeback), "");
                  ((w', Writeback), (i, Writeback), "")])).

DefineMacro "BeforeOrAfterEveryWrite":
  forall microop "w", (
    (IsAnyWrite w /\ SameAddress w i) =>
    (AddEdge ((w, DecodeExecute), (i, DecodeExecute)) \/
     AddEdge ((i, DecodeExecute), (w, DecodeExecute)))).

Axiom "Read_Values":
forall microops "i",
OnCore c i => IsAnyRead i => (
      ExpandMacro BeforeAllWrites
      \/
      (ExpandMacro NoInterveningWrite
        /\ ExpandMacro BeforeOrAfterEveryWrite)).
```

Figure 5: μspec axiom enforcing orderings and value requirements for loads on the Multi-V-scale processor. Predicates relevant to load values are highlighted in red. The edge referred to in Section 3.3 is highlighted in blue.

The use of omniscience poses the first major difficulty when translating μspec to SVA. As explained in Section 3.1, SVA verifiers do not check for *future* violation of assumptions. A temporal assumption stipulating that the load of `x` returns 0 for `mp` would only take effect from the cycle where the load of `x` receives its value. Thus, nothing would stop the temporal verification from creating a partial execution where the store of `x` reached its WB stage before the load of `x` did so (similar to partial execution `p` from Figure 4b). Since temporal properties must match all partial execution traces, the translation of the `Read_Values` axiom from Figure 5 must match this partial execution. If the Check suite's omniscience-based axiom simplification is conducted before translation to SVA, the equivalent SVA property would simply be a translation of `BeforeAllWrites`, which requires `Ld x @WB` $\xrightarrow{hb}$ `St x @WB`. Such a property would *fail* to match the partial execution, which has `St x @WB` $\xrightarrow{hb}$ `Ld x @WB`. This naive property would incorrectly report an RTL bug *despite* the design actually respecting microarchitectural orderings!

In short, the limitations of SVA verifiers prevent RTLCheck from enforcing the specified outcome of a litmus test at RTL. Thus, RTL verifiers will check (partial) executions corresponding to all possible outcomes of the litmus test. To overcome this challenge, properties generated by RTLCheck that involve checking the value of a load need to account for *all* outcomes of the litmus test, not just its specified outcome. Section 4.2 describes how RTLCheck generates such properties from μspec axioms.

3.3 Concretizing Abstract μhb Edges

An ordering edge `src` $\xrightarrow{hb}$ `dest` at the level of a μhb graph is a statement that `src` happens before `dest` for the execution in question. It says nothing about when `src` and `dest` occur in relation to other events in the execution, nor does it specify the *duration* of the delay between the occurrences of `src` and `dest`. A naive translation of the axiomatic edge to the temporal semantics of SVA would be to use the standard SVA mechanism of unbounded ranges (Section 4.3 discusses the mapping of the `src` and `dest` nodes themselves):

```
##[0:$] <src> ##[1:$] <dest>
```

This SVA *sequence* (specification of signal values over multiple clock cycles) allows an initial delay of 0 or more cycles (`##[0:$]`) before the occurrence of `src`, since `src` may not occur at the first cycle in the execution. It also includes an intermediate delay of 1 or more cycles (`##[1:$]`) between `src` and `dest`, as the duration of the delay between `src` and `dest` is not specified by the microarchitectural model.

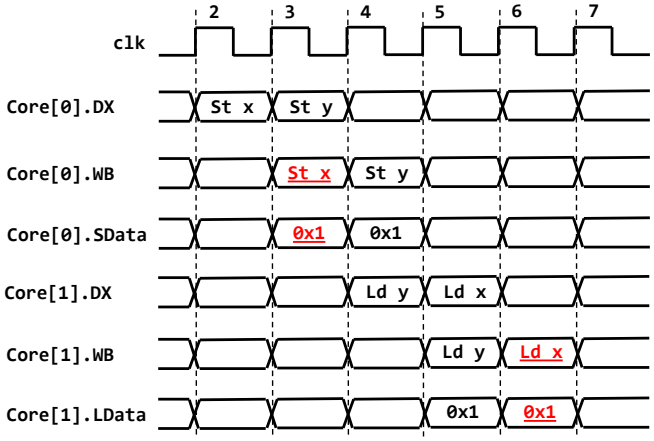


Figure 6: Example execution trace of `mp` on Multi-V-scale where `Ld y` returns 1 and `Ld x` returns 1. Signals relevant to the events `St x @WB` and `Ld x @WB` are underlined and in red.

Unfortunately, this standard mechanism is insufficient for checking that `src` does indeed happen before `dest` for all executions examined. Consider in isolation the edge in blue from Figure 5 in `BeforeAllWrites` enforcing that `Ld x @WB` $\xrightarrow{hb}$ `St x @WB`, with a constraint that the load of `x` must return 0. At the same time, consider the execution trace of `mp` in Figure 6 which reflects the outcome where `St x @WB` $\xrightarrow{hb}$ `Ld x @WB` and the load returns 1. (The relevant signal values are underlined and in red in Figure 6.) Since Figure 6's execution has the events occurring in the opposite order and the load values are different, Figure 6 should serve as a counterexample to the property checking the edge from `BeforeAllWrites`. However, if one simply uses the straightforward mapping above (i.e. `##[0:$] <Ld x=0 @WB> ##[1:$] <St x@WB>`), Figure 6 is *not* a counterexample for the property!

The reason that Figure 6 is not a counterexample is that the unbounded ranges can match *any* clock cycle, including those which

contain events of interest like the source and destination nodes of the edge. For this particular example, the initial `##[0:$]` can match the execution up to cycle 5. At cycle 6, since the load of `x` returns 1, `Ld x=0 @WB` does not occur, so the execution cannot match that portion of the sequence. However, nothing stops the initial delay `##[0:$]` from being extended another cycle and matching cycles 0-6. Indeed, even the entire execution can match `##[0:$]`, thus satisfying the property[5]. Since Figure 6's execution never violates the sequence, it is not a counterexample to the property.

To address this problem of incorrect delay cycle matches, the conditions on the initial and intermediate delays must be stricter to stipulate that they are in fact repetitions of clock cycles where *no* events of interest occur. Section 4.3 describes the specific SVA constructs RTLCheck uses to represent μhb edges in this manner.

3.4 Fire-Once vs. Fire-Always Assertions

An SV assertion or assumption has a clock signal which dictates what it considers a cycle. On every cycle of this clock, any properties move temporally forward one cycle. For example, if checking the following assertion with respect to the execution in Fig. 6[6]:

```
assert property (@(posedge clk) ##2 <St x@WB>);
```

one would expect the property to return true, as the WB stage of the store to x indeed occurs two cycles after the beginning of the execution. (Figure 6 elides the execution's first cycle for brevity.) However, SVA semantics are not that simple. In addition to dictating when the assertion should move forward one cycle, each cycle of `clk` also instantiates a unique copy of the assertion which checks the *entire* property from beginning to end, but starting from the current clock cycle. In other words, one match attempt of the property begins at cycle 1 and checks for `St x @WB` at cycle 3. Another match attempt begins at cycle 2 and checks for `St x @WB` at cycle 4, and so on for every cycle in the execution. If *any* of these match attempts fail, the entire property is considered to have failed. Here, the match attempt that begins at cycle 2 will fail, as `St x` will not be in `WB` at cycle 4 as the property requires.

The conceptual mismatch here is that the SVA notion of an assertion (taken as a whole) is something that holds true starting from every cycle in an execution, whereas a microarchitecture-level happens-before axiom is merely enforced once with respect to an execution as a whole. To bridge this gap, the properties generated by RTLCheck must explicitly filter out match attempts that do not start at the beginning of the execution. Section 4.4 describes our mechanism for doing so.

4 TRANSLATING μSPEC AXIOMS TO SVA

Figure 7 shows the high-level flow of RTLCheck. Three of the primary inputs to RTLCheck are the RTL design to be verified, a μspec model of the microarchitecture, and a suite of litmus tests to verify the design against. The other inputs to RTLCheck are the *program and node mapping functions* (described in Sections 4.1 and 4.3 respectively). Program and node mapping functions translate litmus tests and μhb nodes to initial/final state assumptions and equivalent RTL expressions respectively. RTLCheck has two main components. The **Assumption Generator** (Section 4.1) generates SV assumptions constraining the executions examined by a verifier to those of the litmus test being verified. The **Assertion Generator** (Section 4.2) generates SV assertions that check the individual axioms specified in the μspec model for the specific litmus test while surmounting the axiomatic/temporal mismatches discussed in the previous section. While this section focuses on μspec microarchitectural axioms and temporal SV assertions at RTL, the flow would be similar for other microarchitectural axiom formats and other temporal property languages.

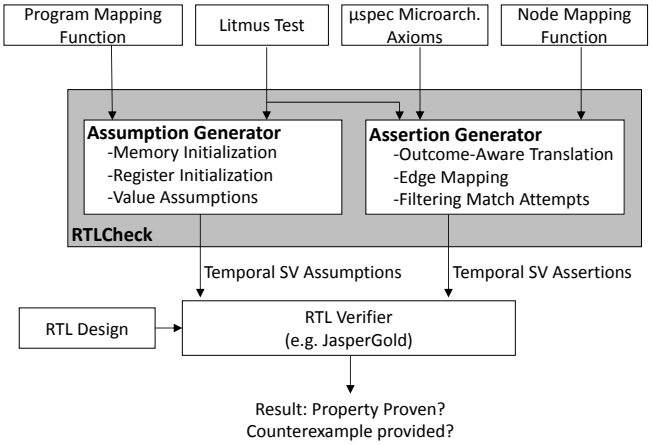


Figure 7: Overall flow diagram of RTLCheck.

4.1 Assumption Generation

RTLCheck's generated properties are litmus test-specific, so the executions examined by an RTL verifier for these properties need to be restricted to the executions of the litmus test in question. As seen in Figure 7, the Assumption Generator performs this task using a *program mapping function* provided by the user. Program mapping functions link a litmus test's instructions, initial conditions, and final values of loads and memory to RTL expressions representing these constraints on the implementation to be verified. The parameters provided to a program mapping function are the litmus test instructions, context information such as the base instruction ID for each core, and the initial and final conditions of the litmus test.

The assumptions generated for a given litmus test must accomplish three tasks:

(1) Initialize data and instruction memories to the litmus test's initial values and instructions respectively.
(2) Initialize registers used by test instructions to appropriate address and data values.
(3) Enforce that the values of loads and the final state of memory respect test requirements in generated RTL executions.

Figure 8 shows a subset of the assumptions that must be generated for the `mp` litmus test from Figure 2 for Multi-V-scale.

Memory Initialization: The first assumption in Figure 8 is an example of data memory initialization. It sets `x` to its initial value of 0 as required by `mp`. The assumption uses SVA implication (`|->`) triggered on the `first` signal being 1. An SVA implication `a |-> b` first checks `a`. If `a` evaluates to true, the implication then checks

[5] At an intuitive level, some readers may find this logic similar to regular expression matching.
[6] `##<i>` specifies a delay of `i` cycles

```
assume property (@(posedge clk) first |-> mem[21] == {32'd0});
assume property (@(posedge clk) first |-> mem[1] == {7'b0,5'd2,5'd1,3'd2,5'b0,`RV32_STORE});
assume property (@(posedge clk) (((core[1].PC_WB == 32'd24 && ~(core[1].stall_WB)) |->
   (core[1].PC_WB == 32'd24 && ~(core[1].stall_WB) && core[1].load_data_WB == 32'd1)) and
   ((core[1].PC_WB == 32'd28 && ~(core[1].stall_WB)) |->
      (core[1].PC_WB == 32'd28 && ~(core[1].stall_WB) && core[1].load_data_WB == 32'd0))));
assume property (@(posedge clk) (((core[0].halted == 1'b1 && ~(core[0].stall_WB)) &&
   (core[1].halted == 1'b1 && ~(core[1].stall_WB)) && (core[2].halted == 1'b1 && ~(core[2].stall_WB)) &&
   (core[3].halted == 1'b1 && ~(core[3].stall_WB))) |-> (1)));
```

Figure 8: A subset of the SV assumptions RTLCheck generates for `mp`. Some signal structure is omitted for brevity.

or enforces `b` beginning that same cycle. The `first` signal is auto-generated by the Assumption Generator, and is set up so that it is 1 in the first cycle after reset and 0 on every subsequent cycle. Thus, the assumption only enforces that the value of the address in memory is equivalent to the initial condition of the litmus test at the beginning of the execution. This distinction is necessary as the verification needs to allow the address to change value as an execution progresses. (The `first` signal is also used to filter match attempts as Section 4.4 describes.) The second assumption is an instruction initialization assumption. It enforces that core 0's first instruction is the store that is `(i1)` in Figure 2's `mp` code.

Register Initialization: The assembly encoding of litmus test instructions uses registers for addresses and data. Register initialization assumptions set these registers to the correct addresses and data values at the start of execution. They are similar in structure to memory initialization assumptions.

Value Assumptions: Load value assumptions cannot be used to enforce an execution outcome (see Section 3.1), but they can still be used to guide the verifier and reduce the number of executions it needs to consider. The third assumption in Figure 8 contains two such implications. Each one checks for the occurrence of one of the loads in `mp` and enforces that it returns the value in the test outcome (0 and 1 for `x` and `y` respectively) when it occurs. The last assumption in Figure 8 is a final value assumption. It contains an implication whose antecedent is the condition that all cores have halted their Fetch stages and all test instructions have completed their execution. The consequent of the implication stipulates any final values of memory locations that are required by the litmus test. Since `mp` does not enforce any such requirements, the consequent of the implication is merely a `1`.

A pleasantly surprising side effect of assumption generation is that for certain tests, assumptions alone turn out to be sufficient in practice to verify the RTL. For most assumptions, JasperGold (the commercial RTL verifier used by RTLCheck) can find *covering traces*, which are traces where the assumption condition occurs and can be enforced. For instance, a covering trace for an assumption enforcing that the load of `y` returns 1 would be a partial execution where the load of `y` returns 1 in the last cycle. A covering trace for a final value assumption in particular would by definition contain the execution of *all* instructions in the test. The covering trace must also obey any constraints on instruction execution stipulated by other assumptions, including load value assumptions. As such, a covering trace for `mp`'s final value assumption is an execution where the load of `y` returns 1 and the load of `x` returns 0. A search for such a trace is equivalent to finding an execution where the entire forbidden outcome of `mp` occurs. If JasperGold can prove that a covering trace for an assumption *does not exist*, it will label the assumption as unreachable. An unreachable final value assumption means that there are no executions satisfying the test outcome. This result verifies the RTL for that litmus test without checking the generated assertions. Thus, a final value assumption forces JasperGold to try and find a covering trace of the litmus test outcome, possibly leading to quicker verification. As such, final value assumptions are beneficial even when the test does not specify final values of memory, but we expect this benefit to be largest in relatively small designs. Section 7.2 quantifies our results.

4.2 Outcome-Aware Assertion Generation

As Figure 7 shows, the Assertion Generator is responsible for translating μspec axioms to SV assertions checking the corresponding properties at RTL. The Assertion Generator translates μspec primitives like `/\` and `\/` to their SVA counterparts `and` and `or`. It reuses most of the Check suite's μspec parsing engine and axiom simplification, but its translation must account for the axiomatic-temporal mismatch as outlined in Section 3.

Section 3.2 shows that the assertions generated by RTLCheck must account for *all* possible outcomes of the litmus test, not just its specified outcome. For example, the assertion generated for the axiom in Figure 5 for `mp`'s load of `x` must account for both the case where the load of `x` returns 1 *and* the case where it returns 0. (The load of `x` cannot return any other values in an execution of `mp`.)

If the load of `x` returns the initial value of 0, this corresponds to the `BeforeAllWrites` portion of the axiom. The SVA property for this case must check that `Ld x @WB` $\xrightarrow{hb}$ `St x @WB` and that the load returns 0. Similarly, if the load returns 1, this corresponds to the part of the axiom comprising `NoInterveningWrite` and `BeforeOrAfterEveryWrite`. The equivalent SVA property here must check that `St x @WB` $\xrightarrow{hb}$ `Ld x @WB` and that the load returns 1.

To accomplish this outcome-aware translation, the mapping onto RTL of any μhb edge containing nodes of load instructions must account for any constraints the edge's axiom enforces on the values of those loads. (These constraints are henceforth referred to as *load value constraints*.) For `mp`'s load of `x`, the μhb edge in Figure 5's `BeforeAllWrites` macro requires `DataFromInitialStateAtPA i` to be true, which in turn requires that the load returns the initial value 0. This requirement on the load's value is stipulated as a load value constraint when mapping the edge from `BeforeAllWrites`. Similarly, the μhb edges in Figure 5's `NoInterveningWrite` and

```
fun mapNode(node, context, load_constr) :=
  let pc := getPC(node.instr, context) in
  let core := node.core in
  match node.stage with
  | IF => return "core[" + core + "].PC_IF == "
          + pc + " && ~stall_IF"
  | DX => return "core[" + core + "].PC_DX == "
          + pc + " && ~stall_DX"
  | WB => let lc := get_lc(node, load_constr) in
          let str := "core[" + core + "].PC_WB == "
               + pc + " && ~stall_WB" in
          if lc != None then
            str += (" && load_data_WB == " + lc.value)
          return str
```

Figure 9: Multi-V-scale node mapping function pseudocode.

`BeforeOrAfterEveryWrite` macros require `SameData w i` to be true. In the case of `mp`'s load of `x`, this predicate enforces that the load returns 1, and this requirement is also encoded as a load value constraint when mapping the edge. The mapped edges of the two cases are combined with an SVA `or` (translated from the μspec `\/`), allowing the property to cater to both the executions where the load of `x` returns 0 and the executions where it returns 1, as required by RTL verifiers.

In μspec, the `DataFromFinalStateAtPA i` predicate returns true if `i` stores a value equivalent to the final value of its address in the litmus test. Accounting for this predicate at RTL requires ensuring that a given write is the last write to a particular address in an execution. However, the inability of SVA verifiers to check for future violation of assumptions also makes it impossible to ensure that a particular write happens last in an RTL execution. Thus, when translating this predicate, assertion generation always (conservatively) evaluates this predicate to false. Doing so ensures that the generated properties check all possible orderings of writes (i.e., a superset of the executions that the Check suite would examine), including the write ordering the litmus test focuses on.

4.3 Mapping μhb Edges to SVA

When mapping an edge between two μhb nodes to SVA, RTLCheck requires some notion of what the nodes represent at RTL. This functionality is provided by the node mapping function written by the user and provided to RTLCheck as input. Figure 9 shows pseudocode for a node mapping function for Figure 1's Multi-V-scale processor. The input parameters for a node mapping function are (i) the node to be mapped, which is a specific microarchitectural event for a specific instruction, (ii) context information, such as the starting program counter (PC) for each core, and (iii) any load value constraints that must be obeyed by the mapping of that node (as described in Section 4.2). The output of the node mapping function is a Verilog expression that corresponds to the occurrence of the node to be mapped in RTL.

As shown in Section 3.3, mapping μspec edges using standard SVA unbounded ranges misses bugs. Instead, the conditions on the initial and intermediate delays in an edge mapping must be stricter to stipulate that they are in fact repetitions of clock cycles where *no* events of interest occur. In this context, an event is of interest if it matches the node in question (i.e., if it matches the microarchitectural instruction and event), but *regardless of the data values themselves*. As such, for initial delays and intermediate cycles, we use this sequence[7]:

```
(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]
```

No load constraints are passed to the calls to the mapping function to generate the delay sequence. This prevents delay cycles from matching cases where events of interest occur with incorrect values, as in the case of the load of `x` from Section 3.3. The overall mapping RTLCheck uses for happens-before edges is:

```
(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]
##1 mapNode(src, lc) ##1
(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]
##1 mapNode(dest, lc)
```

This edge mapping is capable of handling variable delays while still correctly checking the edge's ordering.

Finally, some properties require simply checking for the *existence* of a μhb node, rather than an edge between two nodes. In these cases, we use a similar but simpler strategy to that used for mapping edges. The existence of a node is equivalent to an execution consisting of zero or more cycles where the node does not occur followed by a cycle where it *does* occur; in other words:

```
(~map(node, None)) [*0:$] ##1 map(node, lc)
```

4.4 Filtering Match Attempts

Section 3.4 showed how naive assertions generate multiple match attempts in contradiction to microarchitectural intent. Only the first match attempt is capable of matching all events from a microarchitectural axiom, so it is the only one that should be checked to match microarchitectural intent. To filter out match attempts other than the first attempt, RTLCheck guards each of its assertions with implications triggered by the `first` signal, which is auto-generated by the Assumption Generator (Section 4.1). The original assertion in Section 3.4 would attempt a match every cycle:

```
assert property (@(posedge clk) ##2 <St x@WB>);
```

Instead, we guard the assertion with an implication on `first`:

```
assert property (@(posedge clk) first |->
   ##2 <St x@WB>);
```

Now, any match attempt that begins at a cycle after the first one will trivially evaluate to true, as the `first` signal will be 0 and the implication consequent will never be evaluated. Meanwhile, the match attempt beginning at the first cycle will trigger the implication and cause evaluation of the consequent property as required.

Putting it all together, Figure 10 shows an example assertion for `mp` which checks for the existence of an edge `Ld x=0 @WB` $\xrightarrow{hb}$ `St x @WB` on Multi-V-scale. Note that the load's WB stage checks that the data value returned is 0, as required by the load value constraint. In general, assertions check multiple edges, and are larger than the one in Figure 10.

5 MULTI-V-SCALE: A SIMPLE MULTICORE

This section describes the relevant details of the Multi-V-scale processor, a multicore version of the RISC-V V-scale processor [42]. The

[7] `<x> [*0:$]` represents possibly infinite repetitions of `x`

```
assert property (@(posedge clk) first |-> (((((~((core[1].PC_WB == 32'd28 && ~(core[1].stall_WB)) ||
   (core[0].PC_WB == 32'd4 && ~(core[0].stall_WB)))) [*0:$] ##1
   (core[1].PC_WB == 32'd28 && ~(core[1].stall_WB) && core[1].load_data_WB == 32'd0) ##1
(~((core[1].PC_WB == 32'd28 && ~(core[1].stall_WB)) || (core[0].PC_WB == 32'd4 && ~(core[0].stall_WB)))) [*0:$]
##1 (core[0].PC_WB == 32'd4 && ~(core[0].stall_WB)))))));
```

Figure 10: SV assertion checking Ld x@WB $\xrightarrow{hb}$ St x@WB in Multi-V-scale for `mp` where Ld x returns 0. Some signal structure is omitted for brevity.

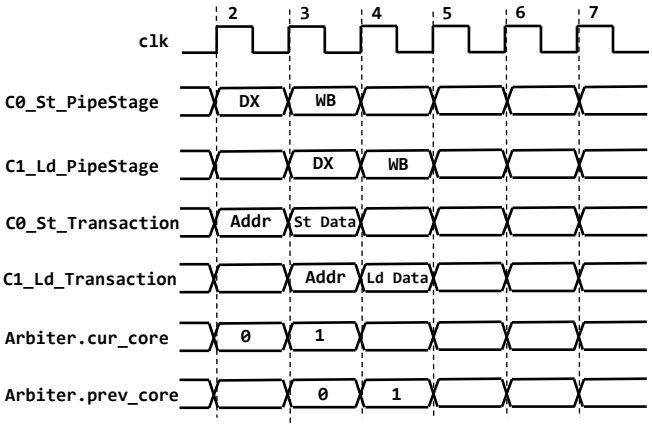


Figure 11: An example timing diagram for Multi-V-scale showing how a store on core 0 and a load on core 1 access memory through the arbiter in a pipelined manner.

V-scale processor itself is a Verilog implementation of the RISC-V Z-scale processor [30], which is written in Chisel.

5.1 V-scale Microarchitecture

The V-scale pipeline is suited for microcontrollers and embedded systems. It is similar in spirit to the ARM Cortex-M0/M3/M4 architectures [30]. The V-scale pipeline is three stages long, as shown in Figure 1. The three stages are Fetch (IF), a combined Decode-Execute stage (DX), and the Writeback stage (WB).

The source code of V-scale does not implement a cache, but does have an implementation of a memory array. When accessing memory, both loads and stores send their addresses to memory in the DX stage. In the WB stage, a load receives its data from memory, and a store provides its data to memory, to be clocked in on the next rising edge.

The V-scale memory is pipelined, and is able to start a memory transaction every cycle. Thus, it can start a memory transaction for an instruction `a` which is in DX while providing data to or reading data from an instruction `b` which is in WB. Figure 11 shows a timing diagram of how V-scale loads and stores operate. The memory does not always operate as expected; RTLCheck discovered a bug in its implementation in the course of our analysis. (See Section 7.1.)

5.2 Multi-V-scale

We created a multicore version of the V-scale processor by instantiating four V-scale pipelines and connecting them to data memory through a simple arbiter that we developed (Figure 1). The arbiter is connected to all four cores, and allows only one of them to access data memory every cycle. If cores other than the one currently granted access wish to access memory, they must stall in the DX stage until the arbiter grants them access.

The arbiter is capable of switching from any core to any other core in any cycle. The switching pattern of the arbiter is dictated by a top-level input to the design. This input stipulates which core the arbiter should grant memory access to in the next cycle. JasperGold tries all possibilities for this input, resulting in all possible switching scenarios between cores being examined when verifying properties generated by RTLCheck. The arbiter accounts for the pipelined nature of the V-scale memory; in other words, it can allow one core to start a request to memory in its DX stage while another core is receiving data from or sending data to memory in its WB stage. Figure 11 shows an example of this pipelining.

In addition to making V-scale multicore, we also added halt logic and a halt instruction to the V-scale implementation. This halt logic lets a thread be stopped once it has executed its instructions for a litmus test. (The RISC-V ISA does not currently have a halt instruction that we could have used.)

5.3 Modeling Multi-V-scale in μspec

We modeled our Multi-V-scale processor in μspec by having one node per instruction per pipeline stage (i.e. one each for IF, DX, and WB respectively). We included Figure 5's axiom, which states that loads must read from the last store to their address to complete its WB stage, or from the initial state of memory. This axiom should hold since stores write to and loads read from the same memory. We also included an axiom enforcing a total order on the DX stages of all memory instructions. This axiom is enforced by the arbiter allowing only one core to access memory at a time while the others stall in DX. We also included properties such as the one in Figure 3b stating that the pipeline stages were in-order. Another axiom we added required a total order on all writes to the same address (enforced through the arbiter's total order on memory operations). Figure 3a depicts an example μhb graph showcasing the edges added by some of these axioms.

6 METHODOLOGY

RTLCheck's Assertion Generator and Assumption Generator are written in the Coq proof assistant [17], which allows for formal analysis of the code. As with prior tools in the Check suite, we use Coq's capability to be extracted to OCaml to generate an OCaml version of RTLCheck that can be compiled and run as a standalone binary.

RTLCheck's generated assertions and assumptions are output as a single file per litmus test, taking only a few seconds per test. A shell script uses these files to create litmus-test-specific top-level modules of Multi-V-scale which are comprised of the implementation of the top-level module concatenated with all the assertions and assumptions for that specific litmus test. We based our changes to V-scale to make it multicore off commit `350c262` in the V-scale git repository [34].

6.1 RTL Verification Methodology

RTLCheck uses JasperGold [12] to verify the SV assertions subject to the SV assumptions for a given litmus test. For the temporal verification supported by SVA, the design itself and any SVA property to be proven on the design are compiled by JasperGold into finite automata, with state transitions at clock cycle boundaries. A property is valid with respect to the design if all execution traces that can be generated by the design satisfy the property [14]. For each property, JasperGold may: (i) prove it for all possible cases, (ii) find a counterexample, or (iii) prove it for all traces shorter than a specified number of cycles (bounded proof).

JasperGold has a variety of different proof engines which are tailored to different purposes. These engines employ SAT (satisfiabililty) and BDD (binary decision diagram)-based approaches to prove the correctness of properties [13]. Section 7 discusses our findings regarding the suitability of various engines for RTLCheck.

When verifying properties, JasperGold takes in a Tcl script as its configuration, which includes the choice of engines to use, how long to allot for the overall verification, and how often to switch between properties when verifying. We use a shell script to generate a Tcl script from a template for each litmus test. The Tcl scripts include a reference to the top-level module for their specific litmus test in the files provided to JasperGold for verification. The test-specific scripts and top-level modules allow instances of JasperGold to be run in parallel for different tests on the same Verilog design without duplication of most of the Verilog RTL. We ran instances of JasperGold on the Multi-V-scale design across litmus tests on a 240-node Intel cluster, allotting 4 or 5 cores and 64-120 GB of memory per litmus test, depending on the configuration used. We use a suite of 56 litmus tests, comprised of a combination of hand-written tests from the x86-TSO suite [39] and tests automatically generated using the `diy` framework [18]. Section 7 gives verification results.

7 RESULTS

This section presents the results of our evaluation of the Multi-V-scale processor RTL. We first discuss a bug we found, and then present RTLCheck runtimes under different JasperGold configurations to compare engines.

7.1 Bug Discovered in the V-scale Processor

In our evaluation of Multi-V-scale, JasperGold reported a counterexample for a property verifying the `Read_Values` axiom (Figure 5) for the `mp` litmus test. This property checks that each read returns the value of the last store to that address that completed WB (provided the read did not occur prior to all writes). Investigating the counterexample trace, we discovered a bug in the memory implementation of the V-scale processor. Namely, if two stores are sent to

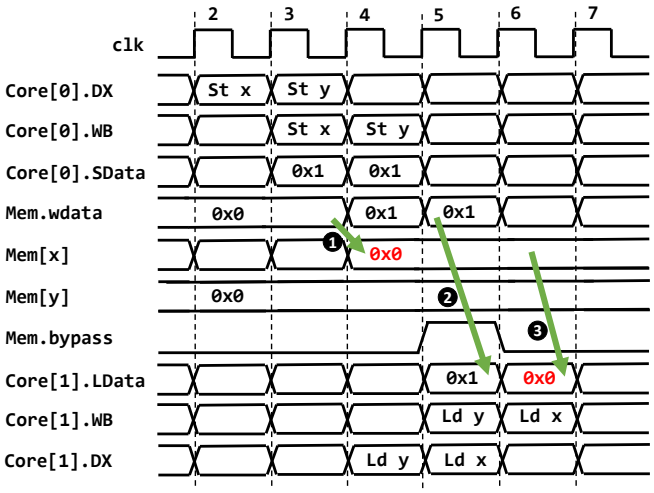


Figure 12: An execution of `mp` showcasing the bug RTLCheck found in the memory implementation of V-scale. The store to `x` is dropped by the memory, resulting in the subsequent load of `x` returning an incorrect value of 0.

memory in successive cycles, the first of the two stores is *dropped* by the memory. The V-scale memory presents a ready signal to the pipeline (or in the multicore case, to the arbiter), and the implementation currently hard-codes this signal to be high. This hard-coded value implies that the memory is ready to accept a new value to be stored every cycle, and so the dropping of values is a bug. This bug would occur even in a single-core V-scale.

Internally, the memory clocks data from stores into a register `wdata`, and only writes the contents of `wdata` to the address of the store in memory when another store initiates a transaction. If a load requests data from the address whose latest store is in `wdata`, the data is bypassed to the load by the memory. Thus, `wdata` functions in a manner akin to a single-entry store buffer.

Figure 12 shows a counterexample trace which violates the RTLCheck-generated property. Here, the two stores of `mp` (to `x` and `y`) initiate memory transactions in cycles 2 and 3 respectively. The `wdata` register is consequently updated to 1 in cycles 4 and 5, one cycle after the stores provide their data values. However, when the second store initiates its transaction at cycle 3, the memory implementation incorrectly pushes the value of `wdata` from cycle 3 into memory at address `x` (arrow ❶ in Figure 12) to make room in `wdata` for the store of `y`. At cycle 3, `wdata` has not yet been updated with the data of the store to `x`, so `x` gets incorrectly set to 0 in memory. The data of the store to `y` is then clocked into `wdata` at the start of cycle 5, overwriting the data of the store to `x` and causing it to be lost. When the load of `y` occurs, it gets its value bypassed from `wdata` (arrow ❷ in Figure 12). This is because no subsequent store has occurred to push the contents of `wdata` to memory. Meanwhile, the load of `x` returns the value of memory at address `x` (arrow ❸ in Figure 12), which is incorrectly 0, violating the property.

We corrected the dropping of stores by eliminating the intermediate `wdata` register. Instead, we clock a store's data directly into memory one cycle after the store does its WB stage. Load data is

Config	Covering Trace Run	Proof Engine Runs	Memory/ Test	Cores/ Test
Hybrid	1 hour	Autoprover (1 hr) K I N AM AD (9 hrs)	64 GB	5
Full_Proof	1 hour	I N AM AD (10 hrs)	120 GB	4

Table 1: JasperGold configurations used when verifying Multi-V-scale with RTLCheck.

combinationally returned in WB as the value of memory at the address the load is accessing. This organization allows data written by a store in one cycle to be read by a load in the next cycle. Once we fixed the bug, RTLCheck was able to completely prove or provide a bounded proof for all assertions. This bug was also independently reported [16], but that report does not correctly diagnose the bug as only occurring upon successive stores. RTLCheck's counterexample trace offered detailed analysis to pinpoint the bug's root cause.

This example highlights an interesting and important use case for RTLCheck: it is most directly intended to catch memory ordering bugs, but is also able to catch bugs that fall on the boundary between memory consistency bugs and more basic module correctness issues. This is because formal verification of RTL takes into account all signals that may affect the property being checked, whether they are explicitly modeled at the microarchitectural level or not. Thus, *any* behavior that causes the property to be invalidated for a litmus test will be flagged as a counterexample by a property verifier checking assertions generated by RTLCheck.

7.2 RTLCheck Runtimes

We ran the properties generated by RTLCheck for the 56 litmus tests in our suite under the JasperGold commercial RTL verifier. JasperGold has many configuration options and solver settings. Space constraints preclude an exhaustive discussion, but we present results for two options.

Table 1 provides the details of the configurations. Each configuration spends one hour trying to verify tests by finding covering traces for assumptions (see Section 4.1), and then runs proof engines on the assertions for the remaining 10 hours. The Hybrid configuration uses a combination of bounded engines (which aim to prove correctness up to a bounded number of cycles) and engines which aim to find full proofs, and also utilizes JasperGold's autoprover. Our second configuration (Full_Proof) uses exclusively full proof engines, which can theoretically improve the percentage of proven properties at the cost of increased runtime. We allocate one core per engine used for each verification job.

Figure 13 presents the runtime to verification of JasperGold for the 56 litmus tests in our suite. For the 22 tests where assumptions were proven to be unreachable through covering traces, the runtime to verification is simply the time taken to check the assumptions. This time can be quite small, as seen for tests like `lb`, `mp`, `n4`, `n5`, and `safe006`, which are verified in under 4 minutes by either configuration. For tests where assumptions were not found to be unreachable, the total runtime is either the time taken to prove all properties, or the maximum runtime of 11 hours per test (if some properties remained unproven). The average runtime is 6.2 hours per test for both the Hybrid and Full_Proof configurations. The total CPU time for the Hybrid run (which used 5 threads per test) is 1733 hours, and that of the Full_Proof run (which used 4 threads per test) is 1390 hours.

Figure 14 shows the percentage of all assertions generated by RTLCheck that JasperGold was able to find complete proofs for within the time limits provided for the 56 litmus tests. In most cases, the Full_Proof configuration can find complete proofs for an equivalent or higher number of properties than the Hybrid configuration can. However, there are tests where the Hybrid configuration does better, such as `n2`, `n6`, and `rfi013`. On average, the Hybrid configuration was able to completely prove 81% of the properties per test, while the Full_Proof configuration found complete proofs for 90% of the properties per test. Overall, the Hybrid configuration found complete proofs for 81% of all properties, while the Full_Proof configuration found complete proofs for 89% of them. Given the negligible difference in average runtime between both configurations, using exclusively full proof engines has clear benefits as it can find complete proofs for a larger fraction of the properties.

For properties that were not completely proven, JasperGold provides bounded proofs instead. The average bounds for such properties for the Hybrid and Full_Proof configurations were 43 and 22 cycles respectively. As litmus tests are relatively short programs, many executions of interest fall within these bounds, giving the user considerable confidence that the implementation is correct.

8 RELATED WORK

Our work is distinct from general RTL verification research in that we focus on MCMs. This allows us to identify strategies that are efficient and tailored enough to give reasonable runtimes, while also being general enough to handle relaxed memory models [6, 25, 46] often not handled by other approaches.

MCM Specification and Verification: A large amount of recent work has considered MCM specification and verification at various layers of the hardware-software stack. This includes specification of the x86-TSO, ARM, and Power memory models at the ISA level [4, 19, 21, 39, 44], as well as specifications of programming language memory models like those of Java [37] and C11 [7, 8, 10]. Specifications of memory models have also been automatically generated from example tests and partial specifications [11]. Compiler mappings from C11 to various ISAs have been formally proven [7, 8, 43], though the proofs have been shown to be flawed at times [36].

The Check suite of tools provides static automated verification of MCM implementations. The suite covers microarchitecture [31], coherence protocol features [35], virtual memory and the OS [32], and HLL memory models [49]. Other tools, such as DVMC [38] and TSOTool [24], perform dynamic MCM verification.

Formal Verification of RTL: Aagaard et al. [1] propose a framework for microprocessor correctness statements that compares specification and implementation state machines. They also propose a formal verification methodology [2] for datapath-dominated hardware using a combination of lightweight theorem proving and model checking, integrated within the FL functional language.

There has been work in the CAD/CAV communities on assertion-based verification (ABV) [50, 53]. However, there is no prior work (to our knowledge) on using such assertions for MCM verification. In addition, such work focuses on handwritten assertions,

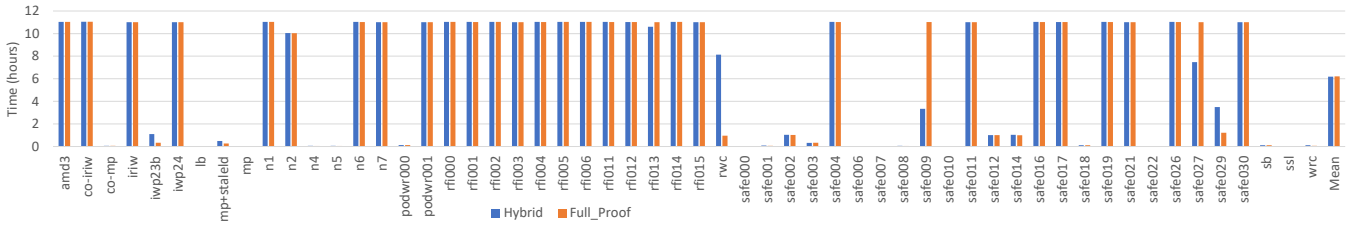


Figure 13: JasperGold runtime for test verification across all 56 tests and both engine configurations.

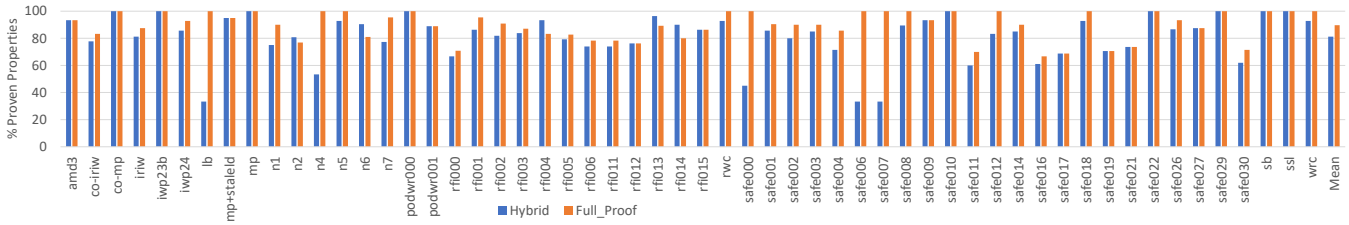


Figure 14: Percentage of fully proven properties (in a max. of 11 hours) across all 56 tests and both engine configurations.

in contrast to RTLCheck's automatic assertion and assumption generation. In that regard, RTLCheck is closer to the ISA-Formal verification methodology created by Reid et al. [41] to verify ARM microprocessors. They use a processor-specific abstraction function (similar to our node mapping function) which extracts architectural state from microarchitectural state, and check correctness by comparing the difference in the architectural state before and after an instruction executes/commits with what a machine-readable specification says the instruction should do to architectural state. They do not verify the entire design. For instance, the memory subsystem and floating-point units are not verified, and they do not address memory consistency issues.

Pellauer et al. [40] provide a method for synthesizing SVA into finite state machine hardware modules, which can then check for the desired behavior as the processor runs. Stewart et al. [47] proposed DOGReL, a language for specifying directed observer graphs (DOGs). These DOGs describe finite state machines of memory system transaction behavior. Users also define an *abstractor* per interface (similar to our node mapping function) that interprets signal-level activity as transaction-level events, whose properties can then be verified. DOGReL compiles down to RTL and SVA, similar to RTLCheck. However, RTLCheck specifically focuses on MCM properties in multiprocessors, which are not discussed in the DOGReL paper. In addition, RTLCheck's μhb graphs represent executions while DOGs represent finite state machines.

Vijayaraghavan et al. [51] formalise the idea of components in a hardware design such as reorder buffers (ROB), register files, store buffers, and caches as labelled transition systems (LTSes). They provide a machine-checked proof in Coq that if the individual components obey certain rules, then the combined system implements sequential consistency for all programs. LTS semantics match those of Bluespec, providing a method to generate RTL from such a formally-proven component behavior specification. While RTLCheck proves correctness for litmus tests rather than all programs, its methodology is capable of handling arbitrary RTL designs that may implement a variety of ISA-level memory models, not just SC. Furthermore, RTLCheck does not require complicated proofs as the LTS setup does, and does not require the design to be written in Bluespec to be synthesizable.

9 CONCLUSION

Memory consistency models are notoriously difficult to specify, let alone to implement properly. Hardware implementations may reorder memory operations to improve memory performance, or use speculative approaches in an effort to extract performance while maintaining ordering. The consequence of such approaches is implementation complexity, which results in the types of memory ordering bugs that continue to appear regularly in processors even today. Furthermore, no prior general-purpose memory consistency verification toolchain has been able to extend its reach all the way down to the level of RTL, forcing designers to instead rely largely on empirical black-box testing.

RTLCheck closes the final missing link in the verification stack: it enables verification of the memory ordering behavior of arbitrary RTL against an arbitrary microarchitecture-level ordering specification for a suite of litmus tests. This verification, when combined with the rest of the Check suite, constitutes a top-to-bottom verification stack that spans from high-level software, through the ISA, all the way down to RTL, within a single unified framework. RTLCheck is open-source (apart from the commercial JasperGold verifier) and is available at github.com/ymanerka/rtlcheck.

10 ACKNOWLEDGEMENTS

We thank Pramod Subramanyan, Yakir Vizel, Hongce Zhang, our shepherd T.N. Vijaykumar, and the anonymous reviewers for their helpful feedback. This work was supported in part by C-FAR (under the grant HR0011-13-3-0002), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, and in part by the National Science Foundation (under the grant CCF-1117147).

REFERENCES

[1] Mark Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. 2001. A Framework for Microprocessor Correctness Statements. In *Correct Hardware Design and Verification Methods (CHARME)*. 433–448.

[2] Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. 2000. A Methodology for Large-Scale Hardware Verification. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000*. 263–282. https://doi.org/10.1007/3-540-40922-X_17

[3] Jade Alglave, Mark Batty, Alastair Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak behaviours and programming assumptions. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[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 (July 2014). Issue 2.

[5] ARM. 2011. Cortex-A9 MPCore, programmer advice notice, read-after-read hazards. ARM Reference 761319. (2011). http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf.

[6] ARM. 2017. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. (2017). https://static.docs.arm.com/ddi0487/b/DDI0487B_a_armv8_arm.pdf

[7] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *39th Annual Symposium on Principles of Programming Languages (POPL)*.

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, , and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *38th Annual Symposium on Principles of Programming Languages, (POPL)*.

[9] Colin Blundell, Milo Martin, and Thomas Wenisch. 2009. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *36th International Symposium on Computer Architecture (ISCA)*.

[10] Hans-J. Boehm and Sarita Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *29th Conference on Programming Language Design and Implementation (PLDI)*.

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

[12] Cadence Design Systems, Inc. 2015. JasperGold Apps User's Guide. (2015).

[13] Cadence Design Systems, Inc. 2016. JasperGold Engine Selection Guide. (2016).

[14] Eduard Cerny, Surrendra Dudani, John Havlicek, Dmitry Korchemny, et al. 2015. *SVA: The Power of Assertions in SystemVerilog*. Springer.

[15] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk Enforcement of Sequential Consistency. In *34th International Symposium on Computer Architecture (ISCA)*.

[16] comododragon. 2016. Stores are not working. (2016). https://github.com/ucb-bar/vscale/issues/13.

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

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

[19] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *43rd Annual Symposium on Principles of Programming Languages, (POPL)*. 608–621.

[20] Harry D. Foster. 2015. Trends in Functional Verification: A 2014 Industry Study. In *52nd Design Automation Conference (DAC)*.

[21] 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 *48th International Symposium on Microarchitecture (MICRO)*. 635–646.

[22] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*. 38–55. https://doi.org/10.1109/SP.2016.11

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

[24] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. 2004. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *31st International Symposium on Computer Architecture (ISCA)*.

[25] IBM. 2013. Power ISA Version 2.07. (2013).

[26] IEEE. 2010. IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std1850-2005)* (April 2010).

[27] IEEE. 2013. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)* (Feb 2013), 1–1315. https://doi.org/10.1109/IEEESTD.2013.6469140

[28] Intel. 2013. Intel 64 and IA-32 Architectures Software Developer's Manual. (2013). http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[29] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computing* 28, 9 (1979).

[30] Yunsup Lee, Albert Ou, and Albert Magyar. 2015. Z-Scale: Tiny 32-bit RISC-V Systems. (2015). https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf.

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

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

[33] Daniel Lustig, Caroline Trippel, Yatin A. Manerkar, Margaret Martonosi, and Michael Pellauer. 2017. Check Research Tools and Papers. (2017). http://check.cs.princeton.edu/.

[34] Albert Magyar. 2016. Verilog version of Z-scale. (2016). https://github.com/ucb-bar/vscale.

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

[36] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. *CoRR* abs/1611.01507 (2016). http://arxiv.org/abs/1611.01507

[37] Jeremy Manson, William Pugh, and Sarita Adve. 2005. The Java Memory Model. In *32nd Annual Symposium on Principles of Programming Languages, (POPL)*.

[38] A. Meixner and D.J. Sorin. 2009. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2009).

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

[40] Michael Pellauer, Mieszko Lis, Don Baltus, and Rishiyur S. Nikhil. 2005. Synthesis of synchronous assertions with guarded atomic actions. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*.

[41] 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 Processors with ISA-Formal. In *28th International Conference on Computer Aided Verification (CAV)*.

[42] RISC-V Foundation. 2015. RISC-V in Verilog. (2015). https://riscv.org/2015/09/risc-v-in-verilog/.

[43] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *33rd Conference on Programming Language Design and Implementation (PLDI)*. 311–322. https://doi.org/10.1145/2254064.2254102

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

[45] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *37th Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[46] SPARC. 1994. SPARC Architecture Manual, version 9. (1994).

[47] Daryl Stewart, David Gilday, Daniel Nevill, and Thomas Roberts. 2014. Processor memory system verification using DOGReL: a language for specifying end-to-end properties. In *International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS)*.

[48] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. 2015. Template-based Synthesis of Instruction-level Abstractions for SoC Verification. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design (FMCAD '15)*. 8. http://dl.acm.org/citation.cfm?id=2893529.2893557

[49] 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. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[50] Babu Turumella and Mukesh Sharma. 2008. Assertion-based Verification of a 32 Thread SPARC CMT Microprocessor. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*. https://doi.org/10.1145/1391469.1391535

[51] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *27th International Conference on Computer Aided Verification (CAV)*.

[52] Mark Walton. 2016. Intel Skylake bug causes PCs to freeze during complex workloads. (2016). https://arstechnica.com/gadgets/2016/01/intel-skylake-bug-causes-pcs-to-freeze-during-complex-workloads/.

[53] Ping Yeung and K. Larsen. 2005. Practical Assertion-based Formal Verification for SoC Designs. In *2005 International Symposium on System-on-Chip*. 58–61.