RealityCheck: Bringing Modularity, Hierarchy, and Abstraction to Automated Microarchitectural Memory Consistency Verification

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

Abstract—Modern SoCs are heterogeneous parallel systems comprised of components developed by distinct teams and possibly even different vendors. The memory consistency model (MCM) of processors in such SoCs specifies the ordering rules which constrain the values that can be read by load instructions in parallel programs running on such systems. The implementation of required MCM orderings can span components which may be designed and implemented by many different teams. Ideally, each team would be able to specify the orderings enforced by their components independently and then connect them together when conducting MCM verification. However, no prior automated approach for formal hardware MCM verification provided this.

To bring automated hardware MCM verification in line with the realities of the design process, we present RealityCheck, a methodology and tool for automated formal MCM verification of modular microarchitectural ordering specifications. RealityCheck allows users to specify their designs as a hierarchy of distinct modules connected to each other rather than a single flat specifications. It can then automatically verify litmus test programs against these modular specifications. RealityCheck also provides support for abstraction, which enables scalable verification by breaking up the verification of the entire design into smaller verification problems. We present results for verifying litmus tests on 7 different designs using RealityCheck. These include in-order and out-of-order pipelines, a non-blocking cache, and a heterogeneous processor. Our case studies cover the TSO and RISC-V (RVWMO) weak memory models. RealityCheck is capable of verifying 98 RVWMO litmus tests in under 4 minutes each, and its capability for abstraction enables up to a 32.1% reduction in litmus test verification time for RVWMO.

I. Introduction

Today's systems-on-chip (SoCs) are complex heterogeneous systems developed by many individuals. Hardware development involves chip agents which develop the processors used in these SoCs is divided up among different teams, with each team responsible for one or a few components. For instance, one team may be responsible for the pipeline, another for the store buffer, a third for the L1 caches, and a fourth for an accelerator. Each team will have detailed knowledge about the components they design, but may know little about other components. Nevertheless, a processor or SoC created by the interconnection of various components developed by different teams must function correctly.

One measure of the correctness of a processor is its conformance to its memory consistency model (MCM). MCMs consist of rules which constrain the values that can be read by load instructions in parallel programs [1]. An ISA-level MCM like that of x86 [14] serves as both a target for compilers as well as a specification that microarchitectures must implement. If a microarchitecture does not obey the MCM of its ISA, then parallel code compiled to target that ISA will not run correctly on the hardware. MCM bugs have been discovered in hardware in recent years [8, 11, 17, 22, 34], and will continue to increase as designs become more complex and parallelism becomes ubiquitous. This makes microarchitectural MCM verification critical to parallel system correctness. There has been much work in recent years on formal MCM specifications of hardware ISAs [25, 26, 28, 32], as well as work on automatically formally verifying that hardware implementations correctly implement those MCMs [17, 18, 21–23, 32]. However, all of these prior automated approaches suffer from deficiencies which put them at odds with the realities of the processor design cycle. RealityCheck shows a graphical depiction of some of these deficiencies. Prior automated formal approaches provide flat verification (Figure 1). There is no way for users to encapsulate component functionality into a unit whose properties only apply to that unit, or to verify a component independently of the rest of the system. In other words, there is no support for modularity as shown in Figure 1. Similarly, users could not build larger modules from smaller ones as there was no support for hierarchy. Figure 1 shows an example of hierarchy where the L1 and Main Memory modules reside within the MemoryHierarchy module.

Prior work also has no support for abstraction. In other words, there is no way for users to decouple the specification of their component’s external behaviour from the specification of its implementation. As an example of abstraction, Figure 1 represents the MemoryHierarchy using an abstract interface AtomicMemory. AtomicMemory specifies the external-facing behaviour of the memory hierarchy, but says nothing about the internal implementation, like how many caches there are. Abstraction facilitates scalable verification by breaking up verification into smaller problems, as this paper shows.

Due to their lack of modularity, prior automated approaches also suffer from a prevalence of omniscient or global properties in their design specifications. For example, a property guaranteed by most shared-memory systems today is that of coherence [29]. At an instruction level, coherence requires that there exists a total order on all stores to the same address that is respected by all cores in the system. However, hardware
implementations of coherence use distributed protocols where each cache (and often a bus/directory) is responsible for enforcing part of the orderings required. None of the hardware components in such an implementation has omniscient visibility of the entire processor, and none of them can make statements about the global behaviour of the system. Thus, global properties such as the coherence definition above reflect a designer’s abstraction of the hardware rather than what the hardware is actually doing. If this abstraction is inaccurate, verification using such a specification will be unsound.

To address these deficiencies of prior automated formal hardware MCM verification approaches, we present RealityCheck, a methodology and tool for automated formal MCM verification which supports modularity, hierarchy, and abstraction. RealityCheck allows users to specify their design as a hierarchy of distinct modules connected to each other, closely matching the structure of real hardware implementations. RealityCheck can then automatically verify whether the composition of the various modules exhibits behaviours forbidden by the ISA-level MCM of the processor through bounded verification for suites of litmus test programs. RealityCheck also lets users write interface specifications of the external behaviour of components; it can then verify component implementation specifications against these interfaces. The use of such abstraction allows a design to be verified piece-by-piece, enabling scalable automated microarchitectural MCM verification.

The contributions of this paper are as follows:

- **A Modular Microarchitectural Ordering Specification Language**: RealityCheck develops μspec++, the first domain-specific language for hierarchical modular specifications of microarchitectural orderings. μspec++ extends the μspec domain-specific language from prior work [18] to incorporate modularity, hierarchy, and abstraction.

- **Automated MCM Verification of Modular Design Specifications**: RealityCheck is the first framework for hierarchical modular specification of hardware orderings that enables automated verification of these specifications against ISA-level MCMs for litmus test suites.

- **Scalable Automated Microarchitectural MCM Verification**: RealityCheck is the first automated approach to enable hierarchical MCM verification of a hardware design piece-by-piece. It breaks down verification of the entire design into smaller verification problems.

---

1. A litmus test is a small program (usually 4-8 instructions long) used to test MCM specifications and implementations.
Axiom "Read_Initial":
forall microop i, forall microop j,
IsAnyRead i /\ DataFromInitialState i /\ IsAnyWrite j /\ SameAddress i j =>
AddEdge((i,L1ViCL_E),(j,L1ViCL_C),""").

Fig. 4. Example µspec axiom.

before relationships between nodes. Figure 3 shows a µhb graph depicting the execution of sb on exampleProc for the outcome where both loads return 0. Each column in the graph represents the path of one litmus test instruction through the microarchitecture, and each row represents the nodes corresponding to one type of event being modelled. So for instance, the node in the second row and first column of the graph represents instruction i1 at its Execute stage, while the node in the second row and second column represents instruction i2 at its Execute stage.

Edges in µhb graphs represent happens-before relationships between nodes, and correspond to the various orderings the microarchitecture enforces. e.g. the blue edge between the Execute stages of i1 and i2 indicates that i2 goes through Execute before i2 does, as required of an in-order pipeline.

The last two rows in the µhb graph (L1ViCL_C and L1ViCL_E) use the ViCL (Value in Cache Lifetime) abstraction to model cache occupancy and coherence protocol events relevant to MCM verification. Briefly speaking, a ViCL represents the period of time (relative to a single cache or main memory) over which a given cache/memory line provides a specific value for a specific address. The time period referenced by a ViCL begins at a ViCL Create event, and ends at a ViCL Expire event. Figure 3 uses L1ViCL_C and L1ViCL_E to refer to ViCL Create and ViCL Expire events respectively, for the L1 caches in exampleProc. We refer the reader to CCICheck [23] for further details.

µhb graphs can be constructed based on microarchitectural specifications that dictate when and where nodes and edges should be added for a given program. Prior work specified microarchitectural orderings as a set of axioms in the domain-specific µspec language [18], which is similar to first-order logic. Figure 2 shows an example µspec axiom. This axiom enforces that for every microop i (a microop is a single load or store), if it is a load (enforced through the IsAnyRead predicate) that reads from the initial state of memory (DataFromInitialState i), then its L1 ViCL must expire before the creation of L1 ViCLs of any write j to that address, as the write would have caused the invalidation of all other cache lines for that address. The loads i2 and i4 in sb both read from the initial state, so this axiom adds the red edges in Figure 3 to enforce the expiration of their ViCLs before the creation of ViCLs for stores i3 and i1 respectively. µspec supports both forall and exists quantifiers.forall quantifiers are translated to ANDs over the litmus test instructions, while exists quantifiers are translated to ORs over the litmus test instructions.

Other axioms for exampleProc would enforce that a given read or write goes through the pipeline stages in order, and that instructions on the same core (like i1 and i2) proceed through a given pipeline stage in program order (blue edges in Figure 3). Another axiom would enforce that a write must create its L1 ViCL (i.e. reach the memory hierarchy) before subsequent loads execute (in order to maintain program order as required by SC), shown by the yellow edges in Figure 3.

A cycle in a µhb graph implies that an event must happen before itself, which is impossible. Thus, cyclic µhb graphs correspond to executions that are unobservable on the target microarchitecture. Likewise, acyclic µhb graphs correspond to executions that are observable on the target microarchitecture. Figure 3’s µhb graph is cyclic (the edges that comprise the cycle are bolded), and thus this execution is unobservable on exampleProc, as required of an SC microarchitecture.

Typically, there are multiple µhb graphs for a given litmus test and microarchitecture. These correspond to different possible event orderings. To automatically verify a given litmus test on a microarchitecture, prior work uses SMT [4] solvers to search for an acyclic µhb graph that satisfies all µspec axioms. If such a graph is found, the litmus test is observable on the microarchitecture, while if no such graph can be found, the litmus test is guaranteed to be unobservable on the microarchitecture. The microarchitectural observability of the litmus test is then compared to ISA-level MCM requirements. In particular, if the observable litmus test is forbidden by the ISA-level MCM, then the microarchitecture is buggy.

B. Deficiencies of Prior Automated Approaches

1) No Scoping of Axioms: The flat verification conducted by prior automated microarchitectural MCM verification approaches encourages the use of axioms which exercise a global view of the entire processor. For instance, Figure 4’s axiom adds µhb edges between ViCLs of Core 0’s L1 and Core 1’s L1 in Figure 3 to reflect that the loads must read memory before the stores invalidate their cache lines through the coherence protocol. While this axiom is straightforward to write and would be valid in a coherent memory hierarchy, it does not correspond to what the hardware is actually doing, because no single hardware component has global visibility of the entire processor. For example, in Figure 4’s processor, Core 0’s L1 can observe the values in its own cache, but it cannot see the values in Core 1’s L1. Each component in a hardware design can only enforce orderings on the events that it sees. The ordering in Figure 4’s axiom is actually enforced in a distributed manner by a combination of modules, specifically the L1s and (likely) a bus or directory. An architect may surmise that the combination of the orderings enforced by the L1s and bus/directory is sufficient to enforce Figure 4’s axiom, but such an assumption must be validated before using the axiom for microarchitectural MCM verification. Otherwise it is possible that the axiom does not hold in the actual hardware design, leading to unsound verification. A better approach is to allow the teams working on each module to specify the axioms that actually hold in their modules, and then verify that the composition of the modules correctly maintains required MCM orderings. However, all axioms in prior work have global
visibility. There is no way for users to write axioms that are scoped to one portion of the design.

2) No Per-Module or Scalable Verification: Prior automated MCM verification approaches also provided no way for users to verify a component independently of the rest of the system. For instance, all verification in Figure 3’s μbb graph is conducted in terms of instructions, relative to an ISA-level litmus test. There is no way to verify say, the L1 caches individually. Since prior work can only verify a design all at once, it cannot scale to large detailed designs due to the exponential complexity of SMT solving [4]. Ideally, each component would have an interface specification that it could be verified against, and it would be possible to verify a design piece-by-piece, allowing verification to scale to large detailed designs.

RealityCheck solves the above problems through its approach to modularity, hierarchy, and abstraction. The next section provides a high-level overview of how it does so.

### III. RealityCheck Overview

Figure 5 shows the high-level block diagram of RealityCheck. RealityCheck can be run in two ways: (i) for litmus test verification to verify a modular microarchitectural ordering specification against an ISA-level litmus test, or (ii) for interface verification to verify the microarchitectural ordering specification of design components against the ordering specification of their abstract interfaces (Section IV). The latter use case enables a module to be verified independently of the rest of the design. The five steps in RealityCheck operation (Microarchitecture Tree Generation, Operation Assignment, Formula Generation, Translation to Z3, and Graph Generation) are common to both litmus test verification and interface verification. The difference between the two cases lies in which modules are checked and which operations they are checked on. Operations represent the instructions or instruction-like quantities that a module operates on. For example, the operations of a core module would be instructions, but the operations of a memory module would be memory transactions. Figure 6 provides a high-level graphical depiction of RealityCheck’s basic terms (including operations) for Figure 1c’s processor.

Two inputs that are provided to RealityCheck in both litmus test verification and interface verification are the implementation axiom files and the module definition files. These files are specified in the μspec++ language (Section V) developed as part of RealityCheck. The μspec++ language is based on the μspec language [18], but adds support to the language for modularity, hierarchy, and abstraction, much like C++ does to C. The module definition files (Section V-B) specify μspec++ modules in a manner similar to a C++ .h file. Meanwhile, each implementation axiom file (Section V-A) specifies the events relevant to a given module as well as orderings on these events, in a manner similar to a C++ .cpp file.

If verifying a litmus test, the test is provided as input in the .test format from prior work [18]. Meanwhile, if running interface verification, RealityCheck takes in a list of implementation-interface pairs. Each pair specifies an implementation module to verify against an interface specification, and their corresponding node mappings (Section V-D).

The final input to RealityCheck (which is always provided to the tool) is the bound up to which to conduct verification. Similar to most prior automated hardware MCM verification work [17, 18, 22, 23, 32] but unlike PipeProof [21], RealityCheck conducts bounded verification; i.e., it explores all possible executions that use up to the specified number of operations (per module). Thus, RealityCheck is excellent for bug-finding, as we show in our case studies in Section VIII-D.

### IV. Abstraction and Its Benefits

In RealityCheck, interfaces can be used to separate the specification of a component’s functional behaviour from the details of its implementation. For example, users may want to abstract the behaviour of their memory hierarchy as a single atomic memory, as shown in Figure 7.

The use of interfaces has several benefits. First, any implementation of the interface can be verified against the interface specification independently of the rest of the system. This gives users a method to verify the correctness of a design component without needing to link it to a top-level litmus test. For example, Figure 6 shows the verification of the MemoryHierarchy module and its submodules (instances of other modules that exist within MemoryHierarchy) against the AtomicMemory interface. Interface verification enables easy localisation of bugs to a given module based on whether it satisfies its interface. Second, interfaces enable scalable verification. Instead of verifying the entire design at once (Figure 1d), which will likely result in an SMT formula too large for solvers to handle, interfaces enable verification to be split into multiple steps. Specifically, the design is first verified using the (likely) smaller and simpler interface specification of the component (Figure 1c) rather than its implementation. Then, the component implementation is separately verified against the interface specification up to a user-provided bound (Figure 7). These two verification queries can be run in parallel, and will likely be smaller SMT formulae than verifying the design all at once. This process can be repeated further down the hierarchy. For instance, if the L1 caches in Figure 7 had interfaces, those interfaces could be used when conducting interface verification in Figure 7. The L1 implementation could then be separately verified against its interface specification. This splitting of verification queries using interfaces can be done again and again to split verification into smaller problems, thus allowing it to scale.

Third, interfaces allow implementations to be switched in and out easily. For example, if the user wants to introduce a new memory hierarchy (say, one with an L2) to a previously verified version of Figure 1c, then all they need to do to ensure correctness is to verify the new memory hierarchy against the AtomicMemory interface, independently of the rest of the design. Finally, the use of interfaces facilitates sharing of IP between vendors at the SoC level. A vendor can internally verify their implementation against its interface for correctness,
and then share their interface with other vendors without having to share their internal implementation specification.

V. **μspec++ MODULAR DESIGN SPECIFICATIONS**

This section explains the μspec++ domain-specific language using a pedagogical microarchitecture simpleProc, comprised of 3-stage pipelines connected to a single main memory.

A. **Implementation Axiom Files**

Each implementation axiom file specifies the events relevant to a given module. Figure 8 shows part of the implementation axiom file for a module of type Core. The file begins with the module's type, and is followed by a list of the types of events that this module can observe and/or enforce orderings on, denoted using DefineEvent. The rest of the file details the axioms which enforce orderings on these events. Figure 8 shows two such axioms, PO_Fetch and Req_Resp_PO. These axioms are identical to μspec axioms, except that their scope is restricted to the operations of the module in which they are declared. For example, if evaluating Figure 8's litmus test on a μspec++ design, an instance of the Core module representing Core 0 would only be able to see instructions i1 and i2 rather than all the instructions of the test. In such a case, the forall quantifiers in the PO_Fetch axiom would evaluate to an AND over instructions i1 and i2.

By default, events can only be viewed by the module instance in which they are declared, similar to private member variables in C++. For example, the IF and WB events in an instance of the Core module cannot be seen outside that Core instance. This capability allows designers to hide events internal to their design component from other modules in the system, just like in a real Verilog design.

Meanwhile, a module’s external events can be viewed by itself as well as by its parent module and any modules it may connect to (see Section V-B for further details). Such events are labelled with the External keyword when they are declared in the implementation axiom file. For example, the MemReq and MemResp events in Figure 8’s Core module are both external events. Thus, if the Req_Resp_PO axiom adds an edge between the MemResp and MemReq nodes of two instructions, that edge will be visible outside the instance of the Core module in which the instructions reside. Figure 6 shows a graphical depiction of internal and external events/nodes.

B. **Module Definition Files**

Figure 9 shows the module definition of simpleProc's Core module. A module definition file specifies the module’s operation type, properties, submodules, and connection axioms.
Module Processor () {
  OperationType none
  Properties { IsCore no }
  Submodules {
    Core c0 (c : 0)
    Core c1 (c : 1)
    Core c2 (c : 2)
    Core c3 (c : 3)
    Mem mem ()
  }
  ConnectionAxioms {
    Axiom "inst_has_tran":
      forall microop "i" in "c0;c1;c2;c3",
      NodeExists (i, MemReq) =>
        exists transaction "j" in "mem",
        Mapped i j.
    ...
    Axiom "mapped_effects":
      forall microop "i" in "c0;c1;c2;c3",
      forall transaction "j" in "mem",
      Mapped i j =>
        (SameAddress i j /
         IsAnyData i j)
      /
        (IsAnyRead i <=> IsAnyRead j) /
        (IsAnyWrite i <=> IsAnyWrite j) /
        SameNode (i, MemReq) (j, Req) /
        SameNode (i, MemResp) (j, Resp))
  }
}

Fig. 9. SimpleProc's Processor module definition.
Axiom "mem_writes_path":
forall transaction "i",
NodeExists (i, Req) /
AddEdges [ [(i, Req), (i, ViCL_C), ""],
            [(i, ViCL_C), (i, ViCL_E), ""],
            [(i, ViCL_C), (i, Resp), "]" ]

Fig. 10. An implementation axiom of SimpleProc's Mem.

1) Operation Types and Properties: The operations in each module have some type, specified using the OperationType keyword. For example, SimpleProc's Core module has an operation type of microop, since it deals with instructions, while the Mem module has an operation type of transaction. Users can add additional operation types. The Processor module in Figure 9 has an operation type of none, a special identifier indicating that it has no operations. This is because the Processor module serves only to encapsulate the other modules in the system.

When quantifying over operations, RealityCheck verifies that the operation type used in the quantifier matches that of the operations over which the quantifier is being evaluated. So for instance, if the first forall quantifier in Figure 8's PO_Fetch axiom was replaced with forall transaction "i1", RealityCheck would flag a type error.

A module's properties are certain fields that are shared across all instances of the module, similar to static variables of classes in C++. An example of a property is the IsCore property, which can be set to yes or no.

2) Submodules: The submodules of a module are instances of other modules that exist within it. Submodules enable hierarchy in RealityCheck, allowing larger modules to be built using smaller ones. A module can evaluate µspec++ predicates on the operations of its submodules and observe their external events, but it cannot observe their internal events.

When instantiating a submodule, parameters may be passed to the instance to populate some instance-specific fields (similar to C++ constructors). For example, when the Processor module instantiates Core modules as its submodules, it passes each of them an integer parameter c denoting its core ID. This parameter can then be used in the axioms of that module.

3) Connection Axioms: Submodules are connected to each other and to their parent module through connection axioms. The bottom of Figure 9 shows two example connection axioms. Connection axioms are similar to implementation axioms (Section V-A), but have differences in their scope. A module's connection axioms can observe the operations of the module itself and those of its submodules. They can observe all events of their module (both internal and external), but only the external events of any submodules.

Since connection axioms can observe the operations of multiple modules, their quantifiers must specify the domain over which they operate. Each quantifier provides a list of modules whose operations it applies to (this refers to the operations of the module itself). For example, the forall quantifier in the inst_has_tran axiom is evaluated on the operations from modules c0, c1, c2, and c3. Thus, the quantifier evaluates to an AND over all these operations, but
does not apply to the operations in the mem module.

Connection axioms are responsible for translating one module’s operations to those of another, and for linking them together. For example, if a core is connected to memory, the core’s instructions need to be translated and mapped to their corresponding memory transactions. Furthermore, there may be multiple possible translations for a given litmus test, e.g. a load may read from the store buffer in one execution (and thus generate no memory transactions), while in another execution it may read from memory, thus generating a memory transaction.

By checking all the different ways the connection axioms could be satisfied, RealityCheck examines all possible translations of operations between modules. For example, the instr_has_tran axiom in Figure 2 maps each instruction on the four cores which requests data from memory (signified by the existence of its external MemReq node) to some transaction in mem, denoted by the Mapped predicate. This reflects how in a real design, an instruction in the pipeline that accesses memory will generate a corresponding memory transaction. Other axioms not shown ensure that the mapping between instructions and memory transactions is 1-1. Mapping schemes other than 1-1 can be used where necessary.

If operations are mapped to each other 1-1, mapped pairs must agree on which addresses, values, etc. are being accessed. They must also agree on the timing of their events. For example, the second connection axiom (mapped_effects) in Figure 9 enforces some of these orderings. It enforces that if an instruction is mapped to a memory transaction, then they must have the same address and read/write the same data value. It also enforces that load instructions map to read transactions and stores map to write transactions (through the IsAnyRead and IsAnyWrite predicates). In addition, it uses the SameNode predicate to link the MemReq event of the instruction i to the Req event of the transaction j. Likewise, the MemResp event of i is linked to the Resp event of j.

Linking two nodes with SameNode essentially merges the two nodes together, ensuring that they are exactly the same event. In this case, the processor’s request to and response from memory are viewed from the memory side as an arriving request to which it sends a response. Further details about the semantics of SameNode are discussed in Section VII-D.

4) A Graphical Example: Figure 11 graphically depicts the effect of connection axioms in simpleProc. Black outlines denote internal nodes. External nodes are outlined in blue (in Core) or red (in Mem). In Figure 11a, we have two stores i1 and i2 in program order from an instance of Core. Note that the MemResp nodes of i1 and i2 are dotted to indicate that these nodes are not guaranteed to exist. This reflects the fact that a Core cannot just assume that its memory requests will be responded to. The existence of the MemResp nodes must be enforced by axioms in the Mem module and communicated to the Core through connection axioms. The implementation axioms of the Core module in Figure 8 obey this convention; for instance, the Req_Resp_PO axiom does not add an edge between the MemResp event of i1 and the MemReq event of i2 unless the MemResp node of i1 exists.

Meanwhile, in Figure 11b we have two memory transactions t1 and t2, governed by the axioms of the Mem module – specifically, the axiom shown in Figure 10. This axiom enforces that if a write request is provided to the Mem module, then it is responded to. Note that all the nodes of t1 and t2 are dotted, indicating that none of them are guaranteed to exist. This reflects the fact that in simpleProc, memory will remain idle unless data is provided to or requested from it. Without connection axioms, no instructions must interact with memory, and so no nodes or edges in Mem are guaranteed to exist.

When the connection axioms in Figure 2 are enforced, the result is Figure 11, where instruction i1 is mapped to transaction t1 and instruction i2 is mapped to transaction t2. The Req node of transaction t1 is now guaranteed to exist, because it is the same node as i1’s MemReq node (denoted by the concentric blue and red circles), which is guaranteed to exist by Core. Transaction t2’s Req node is similarly guaranteed to exist by t2 being mapped to i2. Figure 10’s axiom now enforces that both transactions are responded to, causing the Resp nodes of t1 and t2 to exist, and thus also causing the MemResp nodes of i1 and i2 to exist (since they are now merged with the Resp nodes). Finally, since the MemResp node of i1 now exists, the Req_Resp_PO axiom of the Core module (Figure 8) now enforces (through the orange edge in Figure 11) that the Core must receive i1’s response from memory before sending i2’s request to memory.

C. Preventing Globally Scoped Axioms

Consider trying to include the globally scoped axiom from Figure 3 in a RealityCheck specification for Figure 1’s processor. Figure 4’s axiom references L1 ViCL nodes, which are internal to their corresponding L1 instance in the processor’s module hierarchy. Thus, an axiom that references ViCL nodes (like Figure 4) must be an internal axiom in the L1 module. However, any internal axiom in an L1 module instance can only observe its own transactions. For example, if evaluating the litmus test sb (Figure 2), the internal axioms of Core 0’s L1 module can only see the memory transactions mapped to i1 and i2. \$\mu\text{spec++}\$ provides no way for Core 0’s L1’s internal axioms to refer to the memory transactions of i3 and i4. Similarly, Core 1’s L1 instance can observe the memory transactions of i3 and i4, but not those of i1 and i2. Thus, as an internal axiom in RealityCheck, Figure 4 cannot enforce Figure 5’s red edges between i2 and i3 or between i4 and i1, because it cannot refer to either of these pairs at the same time. Likewise, the internal or connection axioms of the MemoryHierarchy module can refer to the memory transactions of all litmus test instructions, but they cannot refer to the internal ViCL nodes of the L1 modules (RealityCheck will flag an error if they do so). Either way, RealityCheck makes it impossible to write a single axiom that accomplishes Figure 4’s functionality. The ordering must be enforced by a combination of axioms: one or more connection axioms to communicate coherence invalidations from one L1 to another, and another internal axiom in the L1 module to process these invalidations and ensure that the ViCLs
expire appropriately. Thus, if users organise their \texttt{spec++}
specifications into modules that reflect the structure of their
design, RealityCheck will automatically prevent them from
writing globally scoped axioms that violate this design structure.

D. Interface Specification and Node Mappings

Interfaces are specified in RealityCheck in a manner simi-
lar to other modules, but with some additional constraints.
Interfaces cannot have submodules or connection axioms, as
their goal is to provide a simple specification of component
behaviour that does not delve into implementation details.

When verifying an implementation against an interface, the
event types of the implementation must be mapped to those of
the interface, so that the interface’s properties can be checked
on the implementation. Otherwise the interface and implemen-
tation would be referring to different events. For example, if
verifying MemoryHierarchy against AtomicMemory as
per Figure 7, one might map the request and response events of
the memory hierarchy to the corresponding request and reponse
events of the atomic memory. This list of node mappings must
be provided along with a interface-implementation pair when
it is input to RealityCheck for interface verification.

VI. REALITYCHECK OPERATION

A. Step 1: Microarchitecture Tree Generation

Microarchitecture Tree Generation creates a tree of
\texttt{spec++} module instances (i.e. copies) according to the module
definition files and interface/implementation axiom files.

B. Step 2: Operation Assignment

Operation Assignment generates and assigns operations to
each module. The design’s axioms are subsequently evaluated
over these operations, with the visibility restrictions enforced
by \texttt{spec++} detailed earlier.

RealityCheck assigns a number of operations to each module
equal to the bound specified by the user as input. For instance,
if assigning operations to Figure 1b’s processor for a bound
of 4, there would be 4 operations assigned to each of the 4
cores, each of the L1s, and to main memory, for a total of 16
+ 16 + 4 = 36 operations for the entire design. The bound is
the maximum number of operations per module that can exist
in any verified execution, so an execution may use only some
of the operations per module. RealityCheck checks this by
associating every operation with an implicit IsNotNull
predicate, and enforcing that axioms only apply to non-null
operations. This is the approach used by tools like Alloy [15].

In litmus test verification, the design’s Core modules
(identified by the IsCore property) are assigned the litmus
test instructions corresponding to their core. These litmus
test instructions are \textit{concrete}; their type, address, and value
are dictated by the litmus test and cannot change. However,
as Section V.B3 covers, verification must cover all possible
translations (up to a bound) of these litmus test instructions to
lower-level modules. RealityCheck enables such translation by
having operations in modules other than cores be \textit{symbolic}, and
having connection axioms enforce requirements on them based
on the instructions they are (directly or indirectly) mapped to.
Symbolic operations are abstract operations which can have
any type (e.g.: read, write, etc.), address, or value, as long as
the design’s axioms are maintained.

Meanwhile, in interface verification, all operations of in-
volved modules are symbolic. Thus, in that case, RealityCheck
verifies that an implementation satisfies its interface for all
possible combinations of operations up to the bound.

C. Step 3: Formula Generation

In Formula Generation, RealityCheck takes the conjunction
of every module’s implementation axioms and connection
axioms, eliminating quantifiers by translating \texttt{forall}s into
ANDs and \texttt{exists} into ORs over each quantifier’s domain
(the operations being quantified over). RealityCheck conducts
some preliminary simplification on the resultant representation,
and then converts it (as described below) into an SMT formula
checkable by the Z3 SMT solver [7].

D. Steps 4 & 5: Translate to Z3 and Graph Generation

RealityCheck translates AND, OR, and NOT operators
to their Z3 equivalents. Each predicate is mapped to a Z3
Boolean variable, except for \texttt{SameNode} (explained below).
RealityCheck uses Z3’s Linear Integer Arithmetic (LIA) theory
to enforce happens-before orders. Each \texttt{hb} node has two
variables in Z3. The first is a Boolean variable dictating whether
or not the node exists. The second is an integer variable
denoting the timestamp of the node in the microarchitectural
execution. An edge from a node \(s\) to a node \(d\) is translated to
a constraint \(e_s < e_d\) where \(e_s\) and \(e_d\) are the integer
variables denoting the timestamps of \(s\) and \(d\) respectively.

The \texttt{SameNode} predicate requires special handling, as it
is not just a Boolean predicate, but also enforces that two
nodes be merged together. If the user declares two nodes
to be the same node in their \texttt{spec++}, RealityCheck first
creates a bi-implication between their Boolean variables to
ensure that if one exists, so does the other (and vice versa).
Then, RealityCheck adds a constraint that the integer variables
denoting the timestamps of the nodes must have the same value.
Together, these two constraints ensure that the two nodes in
the \texttt{SameNode} predicate are treated as the same node.

If Z3 finds a satisfying assignment to the generated formula,
the assignment represents an acyclic graph where nodes with
their Boolean variables set to true exist, and where edges exist
between nodes \(s\) and \(d\) if the integer variable for \(s\) is less
than the integer variable for \(d\). In such a case, RealityCheck
parses the satisfying assignment to generate a \texttt{hb} graph that
the user can view and use for debugging. If Z3 cannot find
a satisfying assignment, then no acyclic \texttt{hb} graph satisfying
the constraints exists, and the outcome under consideration is
unobservable (up to the user-specified bound).

VII. REALITYCHECK USAGE FLOWS

RealityCheck may be used to verify a component against its
interface in isolation (Section IV), independent of the rest of
the design. This capability of per-module verification enables
TABLE I
MICROARCHITECTURES EVALUATED USING REALITYCHECK

| Name            | Pipelines  | Mem. Hierarchy | MCM  |
|-----------------|------------|----------------|------|
| simpleProc      | inOrderCore| unifiedMem     | SC   |
| cacheProc       | inOrderCore| L1Hier         | SC   |
| simpleProcTSO   | sbCore     | unifiedMem     | TSO  |
| cacheProcTSO    | sbCore     | L1Hier         | TSO  |
| simpleProcRISCV | rvwmoCore  | unifiedMem     | RVWMO|
| cacheProcRISCV  | rvwmoCore  | L1Hier         | RVWMO|
| heteroProcRISCV | 2 sbCore-RISCV, 2 rvwmoCore | L1Hier | RVWMO|

RealityCheck to adapt to multiple design flows. If used at early-stage design time, users may first come up with a shallow design specification where all modules are represented by their interfaces, and then progressively replace modules with their submodules and implementations to create a more detailed design over time. Interface verification can be used to check implementations for correctness as the design becomes more detailed in this “outside-in” approach. If interface verification finds bugs, then additional axioms should be added to the implementation until interface verification succeeds.

On the other hand, if an implementation (or RTL) already exists, users may favour an “inside-out” approach, where they first model one or a few modules deep in the system, and progressively add more modules and hierarchy to the specification until the entire design is modelled. In the case where RTL exists, users may use a tool like RTLCheck [22] to check that the axioms they are writing for modules are sound.

When decomposing a module into submodules, users should try and minimise communication between the submodules. If such decomposition proves difficult, or would result in submodules with minimal internal functionality and large amounts of communication between them, it may be better to leave the module as a single unit (i.e. no submodules). The more internal events one can hide using abstraction, the faster verification will be (generally speaking). So for instance, RealityCheck will be fast for a microarchitecture that has a simple request-response interface to memory where all other memory functionality is hidden. Likewise, it will be slower for a speculative microarchitecture where coherence invalidations are propagated up to the pipeline. Nevertheless, RealityCheck is still capable of verifying such microarchitectures.

VIII. METHODOLOGY AND RESULTS

A. Methodology

RealityCheck is written primarily in Gallina, the functional programming language of Coq [30]. It also includes some OCaml, specifically the code to translate µspec++ formulae into Z3 using the Z3 OCaml API. RealityCheck builds on the Check suite’s µspec parsing and axiom simplification [18], adding support for the various µspec++ features discussed in Section [V]. In contrast to prior work [18] that used a basic SMT solver written in Gallina that supported µspec, RealityCheck utilises the state-of-the-art Z3 SMT solver [7] to check its SMT formulae. We extract our Gallina code to OCaml using Coq’s built-in extraction functionality, and compile it and our OCaml code along with Z3’s OCaml API into a standalone binary that can call into Z3 to solve SMT formulae. Experiments were run on an Ubuntu 18.04 machine with 4 Intel Xeon Gold 6230 processors (80 total cores) and 1 TB of RAM. Each run of RealityCheck only utilises one core at time, but multiple instances of it can be run in parallel.

B. Verifying Litmus Tests

Table [I] lists the 7 microarchitectures (each with 4 cores) on which we ran RealityCheck. The first six microarchitectures are comprised of the possible combinations of three pipelines and two memory hierarchies. The three pipelines are: (i) inOrderCore, an in-order 5-stage pipeline that performs memory operations in program order, (ii) sbCore, an in-order 5-stage pipeline with a store buffer from which it can forward values, and (iii) rvwmoCore, an out-of-order RISC-V pipeline which implements RISC-V’s RVWMO weak memory model [27]. sbCore is capable of reordering writes with subsequent reads, as allowed by TSO (Total Store Order), the consistency model of Intel [14] and AMD x86 processors. rvwmoCore is more relaxed; it allows any loads and stores to different addresses to be reordered (in the absence of fences), while preserving address, data, and control dependencies as
required by RVWMO. rvwmoCore supports coalescing of stores in its store buffer, and also supports all 8 of the RISC-V base ISA’s fences for ordering memory loads and stores. The two memory hierarchies we model are unifiedMem, a single unified memory, and L1Hier, which consists of an L1 cache module backed by a module for a unified memory. The L1 cache in L1Hier is a non-blocking cache. It models requests for data from main memory, cache occupancy, and coalescing of stores before writing back to memory.

The final microarchitecture we evaluate RealityCheck on is heteroProcRISC, a heterogeneous RISC-V processor. heteroProcRISC has 2 out-of-order rvwmoCore pipelines and 2 in-order sbCore pipelines modified for the RISC-V ISA. To modify sbCore for RISC-V, we changed it to only enforce RISC-V fences that order writes with respect to subsequent reads, and to treat all other fences as nops. This is because the orderings enforced by other RISC-V fence types are already maintained by sbCore by default.

We also created four interfaces: one for each pipeline (inOrderInt, sbInt, and rvwmoInt) and one for the L1Hier memory hierarchy (AtomicMemory). Pipeline interfaces like inOrderInt reduce each pipeline module to its requests to and responses from memory (and its dependency orderings, in the case of rvwmoInt). Meanwhile, AtomicMemory abstracts L1Hier as a unified memory. These interfaces help verification scale as discussed below.

1) SC and TSO Results: Figure [12] shows RealityCheck’s runtimes (as a box-and-whisker plot) for a suite of 95 SC/TSO litmus tests on the four SC and TSO microarchitectures, both with and without interfaces. These results use a bound of up to 11 operations per module. The SC/TSO suite comes from a variety of sources, including existing x86- TSO suites [25] and automatically generated tests from the diy tool [31]. As Figure [12] shows, RealityCheck’s verification of litmus tests is quite fast, despite the increased detail of its microarchitectural specifications when compared to prior automated formal MCM verification. 92 of the 95 tests are verified by all 8 configurations in under 4 minutes each. The three remaining tests (co-ir1w, n3, and iwp27) take longer as they have a large number of instructions (e.g. n3 has 9 instructions) and/or possibilities to consider. However, RealityCheck still verifies them under all configurations in less than 14 minutes each.

Figure [12] also shows how the use of interfaces provides significant reductions in overall litmus test verification runtime. Pipeline interfaces and AtomicMemory can be used to abstract away portions of each design for litmus test verification. The use of abstraction reduces the total time to verify all litmus tests by 24.2% for cacheProc, 0.6% for simpleProcTSO, and 29.7% for cacheProcTSO. Meanwhile, the use of abstraction increases runtime for simpleProc by 3.7%, illustrating that interfaces may not reduce verification time for very simple designs. The runtime savings are much higher for cacheProc and cacheProcTSO because they use the AtomicMemory interface to abstract L1Hier. L1Hier is relatively detailed when compared to AtomicMemory, so verification using AtomicMemory takes much less time.

2) RVWMO Results: Figure [13] shows RealityCheck’s runtimes for 98 RVWMO litmus tests on our 3 RVWMO microarchitectures, both with and without interfaces. These results use a bound of up to 11 operations per module. The RVWMO litmus tests used are generated using an automated litmus test synthesis tool [19]. The litmus tests we use for RVWMO are the set of litmus tests up to 6 instructions long generated by this tool for the RVWMO MCM.

As Figure [13] shows, RealityCheck verifies each litmus test in all 6 configurations in under 4 minutes per test. The maximum time per test is lower under RVWMO than SC or TSO because our largest RVWMO litmus test is 6 instructions long (compared to e.g. the n3 TSO litmus test which has 9 instructions). Similar to the SC and TSO microarchitectures, interfaces reduce verification time by 32.1% for cacheProcRISC and 30.0% for heteroProcRISC, while increasing runtime by 1.0% for simpleProcRISC.

The use of interfaces for abstraction depends on the implementations of those interfaces being verified against the interfaces. We present results on those next.

C. Interface Verification

We conducted interface verification of three pipelines (inOrderCore, sbCore, and rvwmoCore) against their respective interfaces. Table [1] shows interface verification times for these pipelines (bound of 15). Interface verification of rvwmoCore takes longer than interface verification for the other two pipelines because the RISC-V pipeline is substantially more complicated. Nevertheless, its interface verification completes in 42 minutes.

We also verified the L1Hier memory hierarchy against AtomicMemory. Figure [14] shows interface verification runtime for L1Hier against AtomicMemory with varying bounds. MemHier interface verification runtimes at higher bounds are significantly larger than litmus test runtimes. For instance, interface verification of L1Hier at a bound of 10
takes over 14 hours. This is not surprising, as interface verification checks the implementation for all possible combinations of operations up to the user-specified bound. So for example, if verifying L1Hier against AtomicMemory with a bound of 10, one is checking all possible combinations of up to 10 transactions. This is essentially verifying all possible “programs” (from the perspective of memory) of up to 10 operations, which is far more than the possible memory transaction combinations that could result from a single litmus test.

The high runtimes for interface verification of MemHier are not as big an issue as they may initially seem. As Section VIII-D below shows, bugs in implementations which cause them to not match their interfaces are detectable at lower bounds, and are found quickly even at higher bounds. Thus, even if interface verification has not terminated, if it does not find a bug quickly, the design is likely to be correct. Furthermore, interface verification can be run in parallel with both litmus test verification and with interface verification of other modules, making it well placed to take advantage of large compute clusters. Finally, interface verification of a module only needs to be run once, not once per litmus test. As the number of litmus tests run increases, the time saved from using interfaces for abstraction will draw closer to the time for interface verification.

D. Bug Finding

To test how quickly interface verification can find bugs, we performed three case studies where we added a bug to the implementation of a component and then verified it against its interface. The first bug we added was to remove the axiom in L1Hier’s L1 cache which ensured that it could only have one value for a given address at any time. RealityCheck discovered this bug when verifying L1Hier against AtomicMemory at a bound of 3 in less than a second. Even when the bound was increased to 15, RealityCheck still found the bug in under 2 minutes. The second bug we added was to remove the axiom in L1Hier’s L1 cache which prevented it from dropping dirty values without writing them back. RealityCheck discovered this bug during interface verification at a bound of 4 in less than a second. Once again, even when the bound was increased to 15 operations, RealityCheck still caught the bug in less than 2 minutes. The final bug we added was to try and verify sbCore against inOrderInt. This should fail because inOrderInt requires program order to be preserved, while sbCore relaxes write-read ordering. RealityCheck duly discovered the bug at a bound of 15 in under a second.

IX. RELATED WORK

There has been much work on formal MCM specifications of hardware ISAs in recent years [2, 3, 5, 9, 10, 13, 20, 25, 26, 28, 32, 35, 38]. There has also been much work on MCM verification, using a variety of approaches. Fractal coherence [36] uses a hierarchical design for coherence protocols to enable verifiability, but does not address MCM verification as a whole. Dynamic approaches like TSOTool [12] and DVMC [24] are incomplete in that they do not check all possible executions of the programs they are verifying. DVMC does modularise the system into the pipeline and coherence protocol, but no further. Furthermore, dynamic approaches require an implementation to exist. This is in contrast to RealityCheck, which can be used for early-stage design-time verification before RTL is written.

Automated formal approaches for hardware MCM verification consist of the Check suite [17, 18, 22, 23, 32] and PipeProof [21]. The Check suite can automatically examine all possible executions of a litmus test on a microarchitecture to formally verify whether the test is observable on the microarchitectural specification. Meanwhile, PipeProof can automatically prove microarchitectural MCM correctness across all possible programs using an approach based on abstraction refinement. Despite their successes, the Check suite and PipeProof suffer from a lack of modularity that inhibits their ability to be fully usable in a real-world design setting. Their lack of modularity also encourages the writing of global axioms that do not directly match the underlying hardware. In contrast, RealityCheck supports modularity, hierarchy, and abstraction, which enable independent verification of each component, scalable verification of the entire design, and make it impossible to write global axioms. However, RealityCheck cannot guarantee correctness across all possible programs. Nevertheless, we believe that RealityCheck provides the basis for unbounded MCM correctness proofs of modular specifications in future work, just as litmus test-based verification approaches enabled the development of PipeProof for unbounded proofs of flat microarchitectural specifications.

The only prior hardware MCM verification work that supports modularity, hierarchy, and abstraction is Kami [6, 33]. While Kami can prove correctness across all possible programs, these proofs must be written manually in a Coq framework, a setup which is not suitable for typical computer architects. In contrast, RealityCheck is an automated tool that is easy to use while still providing modularity, hierarchy, and abstraction, though it cannot prove correctness across all possible programs.

Coppelia [37] searches for hardware security exploits by generating a C++ class hierarchy from processor Verilog and conducting backward symbolic execution on it. However, unlike RealityCheck, it evaluates the entire processor at once, and it does not conduct MCM verification.

X. CONCLUSION

Modern processors are complex parallel systems which incorporate components built by many different teams, and they require stringent MCM verification to ensure their correctness. In order to work with the distributed nature of the hardware design process, specification and verification of orderings enforced by hardware components would ideally be modular. However, all prior automated formal hardware MCM verification approaches provide no support for modularity, making it hard for them to be used in real-world design settings.

In this paper we present RealityCheck, the first methodology and tool for automated formal MCM verification of modular hardware design specifications. RealityCheck provides support
in its novel specification language µspec++ for modularity, hierarchy, and abstraction. This allows users to encapsulate orderings enforced by a component into modules which can be composed with each other to create larger modules. Each component can be verified against its interface specification independently of the rest of the system, and regardless of whether the rest of the design specification exists. Verification of the entire design can be split into multiple smaller verification problems, allowing verification to scale. We implement RealityCheck as an automated tool and show that it is capable of verifying hardware designs across a range of litmus tests, and that it can detect bugs extremely quickly. In summary, RealityCheck is a significant step forward on the road to fully verifying Memory Ordering at the Hardware-OS Interface,” in Proceedings of the 47th International Symposium on Microarchitecture (MICRO), 2014.

[16] L. Lamport, “How to make a multiprocessor computer that correctly simulates memory ordering at the hardware-software interface,” IEEE Transactions on Computers, vol. 19, no. 9, pp. 690–691, 1979.

[17] D. Lustig, M. Pellauer, and M. Martinos, “PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models,” in 47th International Symposium on Microarchitecture (MICRO), 2014.

[18] D. Lustig, S. Furl, G. Kharachorloo, and O. Giroux, “Automated synthesis of comprehensive memory model litmus test suites,” in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

[19] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, “A compositional model for POWER multiprocessors,” in ASPLOS ’17. New York, NY, USA: ACM, 2017, pp. 661–675. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037723

[20] S. Mador, Huan, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, “A compositional model for POWER multiprocessors,” in ASPLOS ’17. New York, NY, USA: ACM, 2017, pp. 661–675. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037723

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

[22] Y. A. Maneker, D. Lustig, M. Martinos, and A. Gupta, “PipeprooF: Automated memory consistency proofs for microarchitectural specifications,” in Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 788–801. [Online]. Available: http://doi.acm.org/10.1145/3298063.3298154 [23] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in Handbook of Satisfiability, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2009, vol. 185, ch. 26, pp. 825–885. [Online]. Available: http://www.cs.stanford.edu/~barrett/pubs/BSS10.pdf

[24] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in Handbook of Satisfiability, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2009, vol. 185, ch. 26, pp. 825–885. [Online]. Available: http://www.cs.stanford.edu/~barrett/pubs/BSS10.pdf

[25] B. M. Beckmann, B. R. Gaster, M. D. Hill, and R. A. Rutenbar, “Aspin: A platform for high-level parametric hardware specification and its verification,” in Proceedings of the 4th Annual ACM/IEEE Design Automation Conference, ser. DAC ’97. New York, NY, USA: ACM, 1997, p. 246. [Online]. Available: http://doi.acm.org/10.1145/262279.262375

[26] Y. A. Maneker, D. Lustig, M. Pellauer, and M. Martinos, “CCICheck: Using µbs graphs to verify the coherence-consistency interface,” in 48th International Symposium on Microarchitecture (MICRO), 2015.

[27] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, “Modular verification,” in Reactive Modules for Heterogeneous Multithreaded Computer Architectures,” IEEE Transactions on Dependable and Secure Computing (TDSC), 2009.

[28] S. Owens, S. Sarkar, and P. Sewell, “A better x86 memory model: x86-TSO,” Conference on Theorem Proving in Higher Order Logics (TPHOLs), 2009.

[29] C. Pulte, S. Furl, W. Deacon, J. French, S. Sarkar, and P. Sewell, “Simplifying ARM concurrency: Multi-copy atomic and operational models for ARMv8,” in 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2018.

[30] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, “Understanding POWER microprocessors,” PLDI ’11.

[31] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, “Modular specifications,” in 39th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013. [Online]. Available: http://hsafoundation.com/hot-chips-2013-hsa-foundation-presented-deeper-detail-hsa-hsai/ [32] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in Handbook of Satisfiability, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2009, vol. 185, ch. 26, pp. 825–885. [Online]. Available: http://www.cs.stanford.edu/~barrett/pubs/BSS10.pdf

[33] M. Walton, “Intel Skylake bug causes PCs to freeze during complex workloads,” 2016, https://arstechnica.com/gadgets/2016/01/intel-skylake-6".

[34] S. Flur, K. E. Gray, G. Kharachorloo, and O. Giroux, “Remote-scope promotion: Clarified, rectified, and verified,” in Proceedings of the 20th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-20. Piscataway, NJ, USA: IEEE Press, 2017.

[35] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, “Understanding POWER microprocessors,” in Proceedings of the 43rd Annual IEEE/ACM International Conference on Computer Aided Verification (CAV), 2011.

[36] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, “Understanding POWER microprocessors,” in Proceedings of the 43rd Annual IEEE/ACM International Conference on Computer Aided Verification (CAV), 2011.

[37] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, “Modular verification,” in Reactive Modules for Heterogeneous Multithreaded Computer Architectures,” IEEE Transactions on Dependable and Secure Computing (TDSC), 2009.

[38] S. Owens, S. Sarkar, and P. Sewell, “A better x86 memory model: x86-TSO,” Conference on Theorem Proving in Higher Order Logics (TPHOLs), 2009.

[39] C. Pulte, S. Furl, W. Deacon, J. French, S. Sarkar, and P. Sewell, “Simplifying ARM concurrency: Multi-copy atomic and operational models for ARMv8,” in 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2018.

[40] RISC-V Foundation, “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20190608-base-ratified,” June 2019.

[41] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, “Understanding POWER microprocessors,” PLDI ’11.

[42] D. Sorin, M. Hill, and D. Wood, A Primer on Memory Consistency and Cache Coherence, ser. Synthesis Lectures on Computer Architecture, M. Hill, Ed. Morgan & Claypool Publishers, 2011.

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

[44] The Coq development team, A don’t (diy) tutorial, version 5.07, 2012. [Online]. Available: http://dip.dip.inria.fr/doc/index.html

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

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

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

[48] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, “Remote-scope promotion: Clarified, rectified, and verified,” SIGPLAN Not., vol. 50, no. 10, pp. 731–747, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858965.2814283

12
[36] M. Zhang, A. R. Lebeck, and D. J. Sorin, “Fractal coherence: Scalably verifiable cache coherence,” in MICRO, 2010.

[37] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, “End-to-end automated exploit generation for validating the security of processor designs,” in Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-51. IEEE Press, 2018, p. 815827. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00071

[38] S. Zhang, M. Vijayaraghavan, A. Wright, M. Alipour, and Arvind, “Constructing a weak memory model,” in Proceedings of the 45th Annual International Symposium on Computer Architecture, ser. ISCA ’18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 124–137. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00021