

Towards Tractable Hardware Model Validation Against Real Hardware

Rishika Varma Kalidindi*

University of Texas at Austin
Austin, TX, USA

Reto Achermann

Technical University of Munich
Munich, Germany

Abstract

Verifying low-level system code requires reasoning about how software interacts with the hardware environment on which it runs. Typically, this means specifying an abstract model of the hardware and verifying the code against it. However, the validity of the verification result depends on the accuracy of the formalized hardware model. Unfortunately, validating the accuracy of the formalized hardware model is hard. It not only requires relating the model's specification to the observable behavior of real hardware, but also involves many invisible internal microarchitectural steps that influence the observable execution of a modern processor.

We present a methodology that lets us validate an abstract formal model against real hardware by generating test cases, executing them on real hardware to capture traces of observable behavior, and finally employing a synthesis approach to identify the internal microarchitectural steps that explain the observable behavior. We discuss this methodology for an x86 MMU model of TLB and page-table behavior, using model-specific state reductions to keep test generation and synthesis tractable and to illustrate how similar reduction arguments can apply in other hardware-model contexts.

CCS Concepts: • Computing methodologies → Model verification and validation; • Software and its engineering → Formal software verification.

Keywords: Hardware Model Validation, Software Verification, Complexity Reduction.

ACM Reference Format:

Rishika Varma Kalidindi and Reto Achermann. 2026. Towards Tractable Hardware Model Validation Against Real Hardware. In *14th Workshop on Programming Languages and Operating Systems (PLOS '26)*, September 29–October 02, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3831586.3838143>

*Work done while at the University of Michigan and MPI-SWS.



This work is licensed under a Creative Commons Attribution 4.0 International License.

PLOS '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2877-8/2026/09

<https://doi.org/10.1145/3831586.3838143>

1 Introduction

System software is a high-value target for formal verification due to its critical role in enforcing isolation between processes, containers, or virtual machines. There has been a steady stream of verified kernels [8, 15, 18], hypervisors [27], or security monitors [30]. All system software must directly interact with hardware to establish the desired isolation properties, e.g., by setting up a virtual machine or a process's address space.

Verifying system software means reasoning about the effects of its interactions with hardware. This means that we must write an abstract specification of *relevant* parts of the hardware platform and its behavioral semantics, e.g., control registers, the translation lookaside buffer (TLB), the mechanics of a page table walk, or the semantics of the virtual machine control structure fields. Then use this specification to reason about the effects of a write to a register or a page table entry on the overall system, e.g., whether the process now has access to a given memory page. Consequently, the verification result depends directly on the accuracy of the hardware specification against which it is verified.

Unfortunately, writing accurate hardware specifications is hard in itself, as documentation is often incomplete, imprecise, or even contradictory. There are three main problems: First, the formal specifications of the environment contain bugs [12] that manifest in the verified implementation [14, 23]. Second, there is a reality gap between the formal model and the actual hardware [7, 13], i.e., the formal model does not contain bugs, but is either too permissive or too restrictive. The verification effort becomes much harder than necessary, or overapproximates relevant low-level hardware behavior. Third, not all the steps the hardware takes are directly visible to the software. However, they influence observable outcomes. For example, a speculative TLB fill walks the page table and inserts an entry in the TLB that might be used in a future memory access.

It is paramount that we do not take hardware specification solely on faith, but validate the formal specification against real hardware implementations [5, 23]. The benefit is twofold: a mismatch can indicate a bug in the specification or in the hardware itself. Existing validation approaches cover the ARMv8 architecture specification [24], TSO memory model [21] or the Arm weak memory models [3, 4]. However, none of them explicitly include the invisible, internal microarchitectural steps of the virtual memory system.

We describe a methodology for validating hardware models against real hardware. On a high level, this includes 1) test case generation based on an operational model, 2) execute these test cases on real hardware, and 3) validation of the model against a trace of observed hardware behavior. During validation, we need to identify the *missing* internal microarchitectural steps that justify the observed trace. To ease the analysis of a large number of long, complex traces, we propose automating validation by framing it as a software synthesis problem, thereby preventing manual intervention. For this, instead of synthesizing instructions, we synthesize the invisible, internal microarchitectural steps and use the observable trace as the verification condition.

Exhaustively exploring all possible hardware behaviors is prohibitively expensive. Hence, we must reduce the number of choices for test case generation and synthesis-based validation. To reduce the number of choices in test case generation and validation, we employ techniques that leverage symmetry and abstraction exploitable in the hardware model. Intuitively, we can replace a trace with a simpler one if it still exposes the same hardware behavior. For example, consider an execution trace of memory accesses in an eight-core system; we might be able to find an equivalent trace on a two-core system by combining execution steps or breaking the trace into smaller test traces.

We validate our approach using a model of the x86 memory management unit that models the translation lookaside buffer, page walk caches, page-table walks, speculative TLB fills, and TSO-memory semantics. We cover test-case generation, trace collection, and trace validation against this model. Finally, we present our choice-reduction techniques applicable to our model, which reduce the number of cores, virtual pages, physical pages, and page-table entries.

The contributions of this paper are:

- A synthesis-based validation method that reconstructs internal microarchitectural steps (§4)
- A reduction strategy enabling tractable validation of the x86 MMU model adaptable to other models (§5)
- A prototype implementation evaluated on x86 paging behavior (§9)

2 Background: x86 Processor Architecture

Memory Address Translation The x86 MMU translates virtual addresses used by programs into physical addresses understood by the hardware at the granularity of pages: continuous, naturally aligned regions of memory 4KiB, 2MiB, or 1GiB in size. The page table, a memory-resident, multi-level radix tree, stores this address mapping.

Translation Caching. The x86 MMU has two kinds of per-core caches that speed up address translation: 1) the translation lookaside buffer (TLB) stores the result of successful page table walks, and 2) page-walk cache (PWC) stores partial page table walks. TLB hits bypass the page walk entirely,

while a hit in the PWC skips some levels of the page table. Moreover, the page table walk itself uses the processor's data caches. The TLB and PWC are non-coherent. The OS must explicitly invalidate stale entries after a page table update.

Store Buffer and Memory Model. The processor implements a store buffer that records memory writes before they are written to the cache, thereby making them globally visible through the cache coherence protocol. The processor can retire store instructions without waiting for the memory subsystem. However, writes sitting in the store buffer are not visible to other cores, while the store buffer is snooped and returns the last-written value for subsequent same-core reads. The x86 architecture follows a *Total Store Order (TSO)* memory model [21, 25], meaning that the hardware guarantees that memory writes become visible in the same order as they are emitted to the store buffer. Serializing instructions (e.g., fences) force store buffer writeback.

Processor Internal Events. A processor effectively executes a sequence of events that we can classify as externally observable or invisible internal microarchitectural events. Instruction execution is an observable event with a well-defined outcome, e.g., a memory read returns the value 42, which gets copied into a register. Internal events such as cache fill/eviction, store buffer writebacks, and speculative execution are not directly observable, but they might influence the outcome of an observable event. For example, the processor speculatively executes a TLB fill, resulting in a TLB hit on the next memory access.

3 Why Accurate Hardware Models Matter

Verifying operating systems is particularly challenging because they directly interact with hardware to configure security-relevant features, such as virtual memory. These interactions can change which memory regions are accessible and which are not. To reason about isolation properties and system integrity, one must formally specify the semantics of the underlying hardware environment and analyze the operating system's interaction with it.

3.1 A Simple Model, and its Problems.

Recall, to unmap a memory page, the operating system must invalidate the corresponding page table entry by setting its present bit to 0. However, this alone is not sufficient, as the TLB might still cache the now-stale translation. We could model the system as two flat mathematical maps: one representing the TLB and one storing a flattened representation of the page table's virtual-to-physical address translation. Then define operations to update a page table entry, fill and invalidate the TLB, and perform a memory access that uses the translation information in the TLB. Unfortunately, this model has several problems in accurately depicting certain important behaviors.

Problem 1: Over-Aproximation. The page table is a flat mathematical map. Updates to intermediate levels in the page table structure are abstracted away. TLB fills are atomic.

Problem 2: No Memory Model. Store buffers and data caches are omitted. Updates to the page table are atomic and become globally visible to all cores simultaneously.

Problem 3: Missing Internal Microarchitectural Steps. The hardware may speculatively evict and/or fill the TLB, or write back a store buffer entry, at any time, even while updates to the page table are in progress. As a result, the MMU may not see updates to the page table immediately. Thus, it may map a virtual address to the wrong physical address or apply the stale (wrong) permissions during access control, breaking isolation and violating system security.

3.2 The Core Problem: Model-Reality Mismatch

There are two kinds of mismatches:

1) Hardware steps outside of the modeled behavior. The formal model abstracts away relevant underlying details (e.g., page table structure, store buffers, non-atomic updates) or overly constrains its operations (e.g., by preventing certain memory reorderings that are possible on weak-memory hardware). Consequently, hardware may exhibit behavior for which there exists no corresponding step in the specification, and verification against such an overapproximated specification yields an implementation that does not correctly handle behavior present on real hardware. This results in undefined behaviors, potentially causing security violations.

2) Model Specification is too Broad. The formal model is more permissive than the real hardware. For example, it allows arbitrary reorderings of memory writes while the hardware enforces a total-store order. While this covers all behavior of the underlying hardware, it also forces reasoning within a less permissive environment specification, which may make the proof effort more substantial or impractical.

3.3 The Environment Specification must be Tight

The environment specification should express the hardware’s possible behavior, nothing more. A too-strict model yields meaningless verification results because the verified program runs in an environment it cannot handle, and its subsequent behavior may diverge from expected, safe patterns [14, 23]. While an overly permissive environment specification might lead to a more complicated verification effort or even less performant code due to unnecessary synchronization.

4 Design

Figure 1 shows the overview of our model validation pipeline. The goal is to systematically validate the hardware model against real hardware. The validation pipeline consists of a test-case generator, a test execution harness, and a trace validator. The pipeline’s input is a reduced hardware model that serves as the basis for generating test programs and

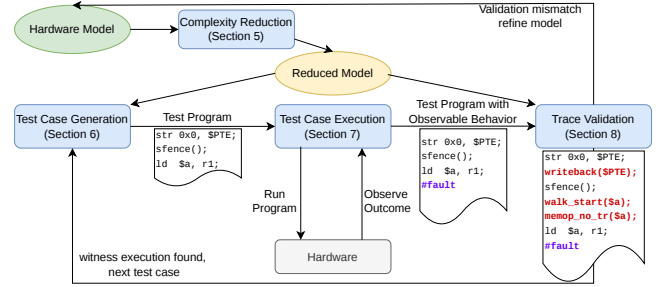


Figure 1. Overview of the validation methodology. Bolded red steps denote synthesized internal transitions required to explain the observed trace.

validating traces. We first present the problem description, then explain each component in the validation pipeline.

Problem Definition. Given a hardware model, generate a set of test cases that cover hardware behaviors that we intend to observe, execute the test cases, and for each test, identify whether the output is allowed by the model. Refine the model if there is a mismatch; repeat.

Hardware Model. The hardware model is the part of the system we want to validate. While our high-level methodology extends to other hardware models, for our discussion we specifically consider an x86 multi-core processor with its MMU modeled as a TLA+-style state machine [19] with transitions for *observable* program steps (e.g., memory accesses, barriers, TLB invalidations) and for *internal* hardware events (e.g., a TLB eviction, a page table walk step, a store buffer writeback). Each transition may affect the state (e.g., a memory access reads or updates the value in a memory cell) and may have enabling conditions (e.g., for a memory access to be permitted, there must be a corresponding entry in the TLB). We base our initial model on documentation and existing OS code.

Model Reduction (Section 5) The hardware model must accurately represent the underlying hardware platform. For an x86 multi-core machine, this means all its cores, 64-bit memory, 5-level paging, etc. The resulting state space is enormous. Fortunately, there are many symmetries we can exploit to reduce the state space. For example, while multi-core machines can have several hundred cores, we might need only a handful to observe unique hardware behavior.

We reduce our model using the small-scope hypotheses [17]. The small scope hypothesis anticipates that most behaviors observable in a given system are likely to be observed at a smaller scope as well. In our context, we make the reasonable assumption that a bounded testing environment, e.g., with a small number of cores or pages, would suffice to expose the relevant behaviors. While not formally proven, this assumption is standard in testing and bounded model checking and can guide induction-based proofs. While many of the simplifications we make are closely tied to the x86

hardware, similar reduction arguments could be adapted appropriately to other hardware models.

Test Case Generation (Section 6) Based on the reduced hardware model, we generate diverse test programs, i.e., per-core sequences of instructions that define the trace of externally visible program steps, intended to expose interesting hardware behavior. This is similar to a fuzzer generating inputs to a program under test.

Test Execution Harness (Section 7) We execute the test programs on real hardware. This produces a set of per-core execution traces. Each trace is a sequence of instructions and their observable outcome, i.e., the value of the memory cell or the occurrence of a fault. The challenge here is how to synchronize cores and store the instruction's result without inadvertently constraining the test's outcome. Ultimately, the observed values should reflect hardware translation and caching effects rather than interference from the OS kernel.

Trace Validation (Section 8) To validate the set of collected execution traces, we must extend it to include the internal microarchitectural steps that justify the outcomes of the observable steps. For example, for a memory access to complete, there must be a corresponding TLB entry and a successful page walk. To model concurrency and asynchrony accurately, we search for a single sequence of model transitions that interleave the execution traces of each core, as well as hardware-triggered unobservable events, as allowed by the model constraints. A failed search indicates a model bug. We cast this as a software synthesis problem: complete a program sketch by interleaving a set of execution traces and inserting the necessary internal steps.

5 State Reduction

To make test generation and validation tractable, we reduce the hardware model by constraining the ranges of values for various aspects of it. We present several model state reductions that we applied, along with informal arguments justifying each. While all aspects are specific to the x86 multi-core model, we expect similar arguments to apply to other hardware models. These justifications are not rigorous proofs, but rather a principled application of a small-scope hypothesis [17] that states that most behaviors observable in a given system are likely to be observed at a smaller scope as well. Specifically, by constraining the domain of values in a principled way, we can still observe the relevant hardware behavior.

Core Fidelity. In a multi-core system, any given core does not directly observe the internal steps of the other cores. Instead, each core only sees memory writebacks from other cores through cache-coherency protocol events. Hence, we do not need to model all cores together with their full fidelity. Instead, we can pick each core separately as the “core under test” and reduce the other cores to execute a sequence of memory writebacks that can happen at any time in the trace.

This breakdown preserves the overall fidelity of all cores while simplifying analysis.

Two cores. By reducing the core fidelity, we can make a second observation: From the “core-under-test's” point of view, it does not matter which other core executed the memory writeback. What matters is the order in which the core observes them. This enables us to merge writebacks from other cores into a single writeback sequence, where the non-deterministic ordering results in a permutation of the order. Assuming a multicopy atomic update architecture [10] such as x86, we can reduce the total arbitrary core count to two: the core under test and a single “remote” core that executes all writebacks of the other cores in the system.

Two Levels of Paging. The x86 architecture has up to five levels of page tables that store the virtual-to-physical address translation. We note that for a page walk to resolve to the wrong address, any wrong step while traversing the page table, including using the page walk caches, is sufficient. The critical aspect here is that we need to model the non-atomicity of a page table walk. Specifically, we must 1) capture the effect of the page walk caches that skip the upper levels of the page table on the page walk, 2) include potential behaviors resulting from interleavings between page table writes and page walks, and 3) expose the effect of super pages in the mapping.

Selecting only the lowest two levels (PMD and PT in Linux terms) for modification, and keeping the remaining levels fixed, lets us expose the necessary behavior: the page walk is non-atomic because it reads the PMD and PT entries in two steps. Further, a page table walk may use the old value of the PMD entry directly or indirectly: either by skipping it entirely via the page walk cache, or by reading the PMD entry before it was updated (cases 1 and 2). Similarly, on x86, super pages are established at the PMD level, and thus the two levels are sufficient to expose changing of page sizes (case 3). Other behaviors at upper-level page table nodes or other superpage sizes can be mapped to a similar sequence of steps on the two-level setup.

Three Virtual Pages (or PTEs). Each virtual page maps to a specific leaf-level page table entry. Each virtual page has a distinct path that the page table walker takes during address translation. The important factor here is the shared prefix. While we can consider paths that diverge at the root largely independent, paths that share a prefix are of interest as they will populate the page walk caches. Hence, the common prefix of the path might be skipped due to those caches. We select three virtual pages: two on the same PT page and one on a different PT page, covering the earlier prefix divergence.

Four Virtual Addresses. The x86 MMU does not translate at the byte level; instead, it operates at the page granularity. Given that we reduced the number of virtual pages to three, we should select one representative address of each page. In addition, we add one additional address on one of the pages, exposing a TLB-cached virtual-to-physical mapping.

Four Physical Frames. These are the corresponding physical memory frames that the virtual pages map to. Here, we must be able to remap an existing mapping of the virtual pages without creating an alias. Hence, we need to use one more physical frame than we use virtual frames.

6 Test-Case Generation

A test case is a program, i.e., a set of per-core instruction sequences, that interacts with the hardware. The program includes writing a page table entry, reading or writing a memory address, invalidating a TLB entry, and emitting a barrier. The instructions are parameterized. For example, a memory store takes an address and a value. We use the constraints from the model reduction to select arguments for reading and writing memory addresses and page table entries. For example, we select one of the four virtual memory addresses for a memory read. We also ensure that we pick values so that no test pattern type is repeated, using different addresses that don't introduce new scenarios.

Graph-based Generation. Instead of randomly generating programs, we leverage a graph representation of the system. Specifically, we express nodes as instances of the page table allowed by the constrained model. Edges represent writes to page table entries that update the page table.

When generating a test case, we pick a node in the graph as the starting point. Then we start picking instructions for the test program. We use the model to check the outcome of a memory access (i.e., whether it causes a page fault) and ensure that both cases are covered in the test case. Finally, we split each test case into a per-core sequence of instructions and link them into the test harness.

Limiting Test Case Lengths. Test case length can be limited by tracking which graph nodes each test case visits and ensuring each node is visited at most once. We prioritize test cases with fewer page table modifications first, then increase the number of modifications until no reachable nodes remain. This strategy can be applied to the other instructions, i.e., normal memory accesses, barriers, and TLB invalidations. This avoids repeated memory accesses to the same page without requiring page table modifications, thereby bounding the length of our test programs.

Generator Implementation. We implemented the test generator in Rust, totaling 772 LoC, using an executable model of the x86 MMU. This includes graph-based test case generation, insertion of appropriate memory operations, page invalidations, barriers, and per-core test sequence output. Limiting test case length is a work in progress.

7 Test Harness

Executing tests on real hardware is not a trivial endeavor, as we must run the test case in supervisor mode while limiting potential interference from other programs and the rest of the operating system. Further, we must carefully eliminate

interference from the test case itself; e.g., storing the result of a memory access adds additional memory accesses that might interfere with test execution.

Instead of writing our own test harness from scratch, we leverage Barrelfish's multikernel architecture [6] for our prototype. Specifically, it has isolated per-core kernels and runs system calls with interrupts disabled, limiting interference. That lets us leverage Barrelfish's infrastructure for booting cores and memory management. We integrate the test cases into the kernel. We write a small user-space driver that performs setup, coordinates among the cores, and then invokes the test case in the kernel via a system call. At the end, we collect the results: the observed outcome of the memory accesses (i.e., whether they caused a fault and the value read).

We cannot use shared-memory barriers for synchronization among cores, as they introduce memory accesses that add undesired interference with the tests. Instead, we intend to leverage synchronized timestamp counters for synchronization. The test drivers agree on the timestamp counter value at which each core starts its execution.

8 Trace Validation with Software Synthesis

To validate whether the execution traces match the model, we need to find a sequence of model transitions that explain the traces' behavior. Recall that the model's transitions fall into observable and internal steps, where the observable steps correspond to the behavior of the execution traces. While for short traces, this is easily achieved by manual analysis, we expect situations with long traces with many internal steps or a large number of traces to be analyzed.

We observe that we can treat the model's state transitions as the possible steps a program can take. In our case, this lets us cast the problem of trace validation into a program completion problem: we need to find the missing internal steps and combine the observed traces from the execution into a single sequence. In other words, the problem we are facing is similar to "repairing" the traces by patching missing steps, a process similar to sketching [26] or program repair [20] in the context of software synthesis. Thus, we cast trace validation as a software synthesis problem.

The Synthesis Problem. *Given the model and an observable trace set, we ask a solver to find a combined sequence of internal actions interleaved with the observed external events, that is admissible by the model.*

A satisfying solution is a witness execution containing the internal actions that explain the observed outcomes of the test program. An unsatisfiable query indicates that no model-admissible execution can explain the traces and is evidence that the model is incorrect or incomplete.

Synthesis Implementation. We encode the synthesis problem in Rosette [28] as a set of constraints over a single linear trace that interleaves the core's actions. Listing 1 shows the simplified encoding consisting of the following elements:

```

1 ;; State: MMU components.
2 (struct State (mem root pt tlb walks cache sbuf))
3
4 ;; Model transitions modifying state
5 (define (step st op) ...) ; returns (state, valid?)
6
7 ;; Grammar: candidate linear traces.
8 (define-grammar (trace-grammar)
9   [tr (choose (Return) (Seq (op) (tr)))]
10  [op (choose (OpMemOp c va r)
11             (OpWritePTE c pa pte)
12             (OpWalkNext c w)
13             (OpTLBFill c w) ...)]
14  [c (choose 0 1 2)]
15  [va (choose VA0 VA1 ...)]
16  [pa (choose PA0 PA1 ...)]
17  [pte (choose #f (cons pa valid?))]
18  [w (choose walk0 walk1 ...)]
19
20 ;; Interpretation: Semantic definition of steps
21 (define (interp tr st)
22   (match tr
23     [(Return) (values st #t)]
24     [(Seq op rest)
25      (define-values (st1 ok1) (step st op))
26      (define-values (st2 ok2) (interp rest st1))
27      (values st2 (and ok1 ok2)))]
28
29 ;; Program Correctness Predicate.
30 (define (check observed)
31   (define tr (trace-grammar))
32   (assert-required-ops tr observed)
33   (define-values (_ valid) (interp tr (make-check-state)))
34   (assert valid))
35
36 (synthesize #:guarantee (check observed-trace))

```

Listing 1. Simplified Rosette encoding for trace validation. The grammar expresses a linear sequence of model actions; the correctness condition mandates the ordering of observed events and asserts that the model accepts the interpretation.

1. *System State.* One-to-one encoding of the model’s state in Rosette, i.e., memory, per-core TLBs, caches, etc.
2. *Grammar.* Defines a grammar from which the trace is completed. It contains a rule for each model transition.
3. *Interpretation.* Defines the grammar semantics equivalent to the model transitions. If a step would not be permissible, the interpretation yields the non-updated state with an indication of failure; otherwise, it updates the system state according to its semantics.
4. *Program Correctness Predicate.* A candidate program is correct if it contains the observed external actions in the same order and with the same outcomes as in the execution trace.

Rosette then uses the grammar to construct candidate programs, initializes the system state, starts interpreting the program, and checks whether the program satisfies the trace using the correctness condition.

Challenges. The synthesis problem grows exponentially in trace length and grammar size. While our current prototype can handle small scenarios (i.e., finding a handful of missing internal actions), scaling it to larger ones requires additional work. Our model reduction technique has proven useful in this regard. However, there are still some opportunities to

further reduce the choices. For example, we do not currently impose structure or order on the internal steps. Instead, we let Rosette determine them during synthesis. While this includes all possible interleavings, it also produces traces that would not occur in hardware, e.g., filling the TLB before a corresponding page walk completes. The core challenge here is to over-constrain the grammar.

A multi-step synthesis approach offers a promising path. Specifically, we plan to:

- *Abstract non-test cores.* Currently, we model all cores with the same fidelity, as we have not implemented the core-fidelity reduction yet (§5). Consequently, Rosette must handle internal steps for all cores, which increases complexity. We plan to abstract away all internal steps for the other cores.
- *Use Big-Step Walks.* The model expresses page walks as multiple small steps. Instead, we can introduce a complete walk as a grammar element, interleaved by possible page-table updates from the local or other cores.
- *Leverage Domain Knowledge.* We observe that a memory access can only succeed if there is a corresponding TLB entry, which in turn can only exist if a page walk succeeded – a dependency we can leverage.

9 Evaluation

We evaluate our approach using a qualitative evaluation showing that 1) we are capable of generating useful test cases for validation, and 2) the synthesis approach successfully finds the missing internal steps. While a complete validation pipeline automatically performs all the steps outlined in Figure 1, our current prototype requires manual intervention.

9.1 Test Case Generation

We evaluate whether using the reduced model to generate test cases yields useful programs that expose the desired hardware behavior.

Methodology. We analyze our graph-based test case generator’s reduction strategy to identify allowed test cases similar to existing hand-crafted Litmus tests designed to expose specific behavior of the memory subsystem [3, 4, 21, 25], particularly those involving the store buffer, since our model also includes it. The classic store-buffer test [11] has two threads, each writing one location and then reading the other. The outcome in which both reads see the initial value is allowed by x86-TSO but forbidden under sequential consistency.

Results. The generator would select two virtual memory addresses and generate a store and a load for each thread. The test generator based on our reduced model would be capable of emitting the following program, which is identical to the classic store-buffer test [11].

```

P0: Write(va1) = 1 | P1: Write(va2) = 1
    Read(va2)      |   Read(va1)

```

```
// Case 1: Successful memory read
memop_read core=0, vaddr=0x00000000, observed=42
// Case 2: Memory read that should not succeed
memop_read core=0, vaddr=0x00008000, observed=42
```

Listing 2. Observable program Traces for the satisfiable and unsatisfiable evaluation cases.

```
// internal step, filling the TLB.
tlb_fill core=0, walk={
    virt=0x00000000, stage=complete,
    pte0=(0xfff00000 -> 0xfff01000, valid),
    pte1=(0xfff01000 -> 0xfff10000, valid)}
// TLB now contains the entry for virt=0x00000000
memop_read core=0, vaddr=0x00000000, observed=42
```

Listing 3. Completed Program Trace for Case 1.

Discussion. While this is a simple example, it shows that our reduction allows the generation of test programs similar to existing Litmus tests.

9.2 Trace Synthesis

We evaluate synthesis-based trace validation using two small case studies, one of which we designed so that no satisfiable solution is permissible under the model.

Methodology. We set the initial state as follows: The TLB is empty. The two-level page table has one valid page mapping, with the value 42 in the first byte. All other pages are not present. The page-walker contains an ongoing walk of the only present page in the page table. We assume this walk was started speculatively before test execution.

We then use the two observable program traces shown in Listing 2 as correctness constraints for the synthesizer. Case 1 accesses the page for which there is a present mapping, whereas case 2 accesses a page that is not present, but the test case reports an observed value.

Results. For the first case, the synthesizer successfully finds the missing internal step: filling the TLB with the value from the page walk (tlb_fill in Listing 3). This step inserts the required entry into the TLB. Recall, a memory access is only possible if there is an entry in the TLB for the corresponding virtual page. The memory access can now read the memory cell and obtain the value 42, which matches the value observed in the trace. In contrast, in the second case, which reads from a non-present page, there is no corresponding page walk to complete. Synthesis reports *no solution*. Since the internal steps cannot update the page table, this could never have resulted in a successful memory access.

Discussion. Despite this being a small example, it demonstrates the approach’s feasibility as a proof of concept. The synthesizer successfully identified the missing internal steps.

10 Related Work

Validation of formal hardware models is important, as the validity of verification results depends on their accuracy [5,

22, 23]; bugs in this environment specification are not isolated occurrences [12]. Prior work focused on the semantics of the instruction set architecture by validating a formal description against real hardware [5, 22, 23]. This enables the construction of verified interpreters of machine code and the reasoning about instructions.

Another line of work focuses on understanding the ARM’s weak-memory model [3, 4] or the TSO model of x86 [21, 25]. Their work uses Litmus tests to validate their memory model against real hardware, testing the hardware-permitted orderings of memory accesses. However, none of these works explicitly include the internal microarchitectural steps and focus only on the directly visible outcome of the memory accesses. Our work also includes the interactions with the memory management unit and page table updates.

IronSpec [14] explored debugging specifications to determine whether developer intent matches the specification, using mutation testing and other techniques. This work focuses on software specifications while we explore the validation of a hardware model. TraceLink [16] checks whether an execution trace of a program matches its state machine specification. However, it requires all events to be present, while we need to handle hidden events. Similarly, approaches such as PAC-style guarantees for automata learning [29] and conformance-testing techniques such as the W-method [9] do not directly apply to our setting due to hidden events, making it difficult to establish the assumptions required for the corresponding correctness guarantees. Velosiraptor [1] applies software synthesis to generate programs that configure the page table, but does not validate whether the specification itself is correct. Neither do static page table generation approaches [2] which further do not change the configuration at runtime.

11 Conclusion and Future Work

We presented a proof-of-concept validation strategy for a formal hardware model against its real-world counterpart. We proposed an adaptable reduction argument to reduce the state space and a trace-validation technique based on software synthesis. Our evaluation shows that this can be a viable approach in validating formal hardware specifications.

While our work shows promising early results, this is just one step towards a full fidelity model, including more complex micro-architectural interactions. While we provided informal arguments for our model-reduction approach targeted at the x86 MMU, a formal proof would strengthen the case. This aspect of our methodology would vary most across different hardware models and could pose scalability challenges if the model cannot be reduced efficiently. We leave these aspects, along with improving our prototype, for future work.

References

- [1] Reto Achermann, Em Chu, Ryan Mehri, Ilias Karimalis, and Margo Seltzer. 2025. Velosiraptor: Code Synthesis for Memory Translation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 1365–1381. doi:10.1145/3676641.3711998
- [2] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. 2021. Generating correct initial page tables from formal hardware descriptions. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) (PLOS '21). Association for Computing Machinery, New York, NY, USA, 69–75. doi:10.1145/3477113.3487270
- [3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *Proceedings of TACAS 2011*. doi:10.1007/978-3-642-19835-9_5
- [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* 36, 2 (2014). doi:10.1145/2627752
- [5] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, and others. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of POPL 2019*. doi:10.1145/3290384
- [6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) (SOSP '09). Association for Computing Machinery, New York, NY, USA, 29–44. doi:10.1145/1629575.1629579
- [7] Márton Bognár, Jo Van Bulck, and Frank Piessens. 2022. Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. doi:10.1109/SP46214.2022.9833735
- [8] Xiangdong Chen, Zhaofeng Li, Jerry Zhang, Vikram Narayanan, and Anton Burtsev. 2025. Atmosphere: Practical Verified Kernels with Rust and Verus. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25). Association for Computing Machinery, New York, NY, USA, 752–767. doi:10.1145/3731569.3764821
- [9] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* 4, 3 (May 1978), 178–187. doi:10.1109/TSE.1978.231496
- [10] Nicholas Coughlin, Kirsten Winter, and Graeme Smith. 2023. Compositional Reasoning for Non-multicopy Atomic Architectures. *Form. Asp. Comput.* 35, 2, Article 8 (June 2023), 30 pages. doi:10.1145/3574137
- [11] diy toolsuite. [n. d.]. X86 SB Litmus Test. <http://diy.inria.fr/tst/doc/SB.litmus>. Accessed 2026-05-21.
- [12] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) (EuroSys '17). Association for Computing Machinery, New York, NY, USA, 328–343. doi:10.1145/3064176.3064183
- [13] Mario Gleirscher, Simon Foster, and Jim Woodcock. 2019. New Opportunities for Integrated Formal Methods. *ACM Comput. Surv.* 52, 6, Article 117 (Oct. 2019), 36 pages. doi:10.1145/3357231
- [14] Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. 2024. IronSpec: Increasing the Reliability of Formal Specifications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 875–891. <https://www.usenix.org/conference/osdi24/presentation/goldweber>
- [15] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) (OSDI'16). USENIX Association, USA, 653–669.
- [16] Finn Hackett and Ivan Beschastnikh. 2025. TraceLinking Implementations with Their Verified Designs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 350 (Oct. 2025), 28 pages. doi:10.1145/3763128
- [17] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290. doi:10.1145/505145.505149
- [18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) (SOSP '09). Association for Computing Machinery, New York, NY, USA, 207–220. doi:10.1145/1629575.1629596
- [19] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. 2002. Specifying and verifying systems with TLA+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (Saint-Emilion, France) (EW '10). Association for Computing Machinery, New York, NY, USA, 45–48. doi:10.1145/1133373.1133382
- [20] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) (ICSE '13). IEEE Press, 772–781.
- [21] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) (TPHOLs '09). Springer-Verlag, Berlin, Heidelberg, 391–407. doi:10.1007/978-3-642-03359-9_27
- [22] Alastair Reid. 2016. End-to-End Verification of ARM Processors with ISA-Formal. In *Proceedings of CAV 2016*.
- [23] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) (FMCAD '16). FMCAD Inc, Austin, Texas, 161–168.
- [24] Alastair Reid. 2017. Who guards the guards? formal validation of the Arm v8-m architecture specification. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 88 (Oct. 2017), 24 pages. doi:10.1145/3133912
- [25] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. doi:10.1145/1785414.1785443
- [26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) (ASPLOS XII). Association for Computing Machinery, New York, NY, USA, 404–415. doi:10.1145/1168857.1168907
- [27] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) (SOSP '21). Association for Computing Machinery, New York, NY, USA, 866–881. doi:10.1145/3477132.3483560
- [28] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) (Onward! 2013). Association for Computing Machinery, New York, NY, USA, 135–152. doi:10.1145/2509578.2509586
- [29] L. G. Valiant. 1984. A theory of the learnable. *Commun. ACM* 27, 11 (Nov. 1984), 1134–1142. doi:10.1145/1968.1972

- [30] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VERISMO: a verified security module for confidential VMs. In *Proceedings of the 18th USENIX Conference on*

Operating Systems Design and Implementation (Santa Clara, CA, USA) (*OSDI'24*). USENIX Association, USA, Article 32, 16 pages.