

Towards Tractable Hardware Model Validation Against Real Hardware

Rishika Varma Kalidindi

University of Texas at Austin



TEXAS

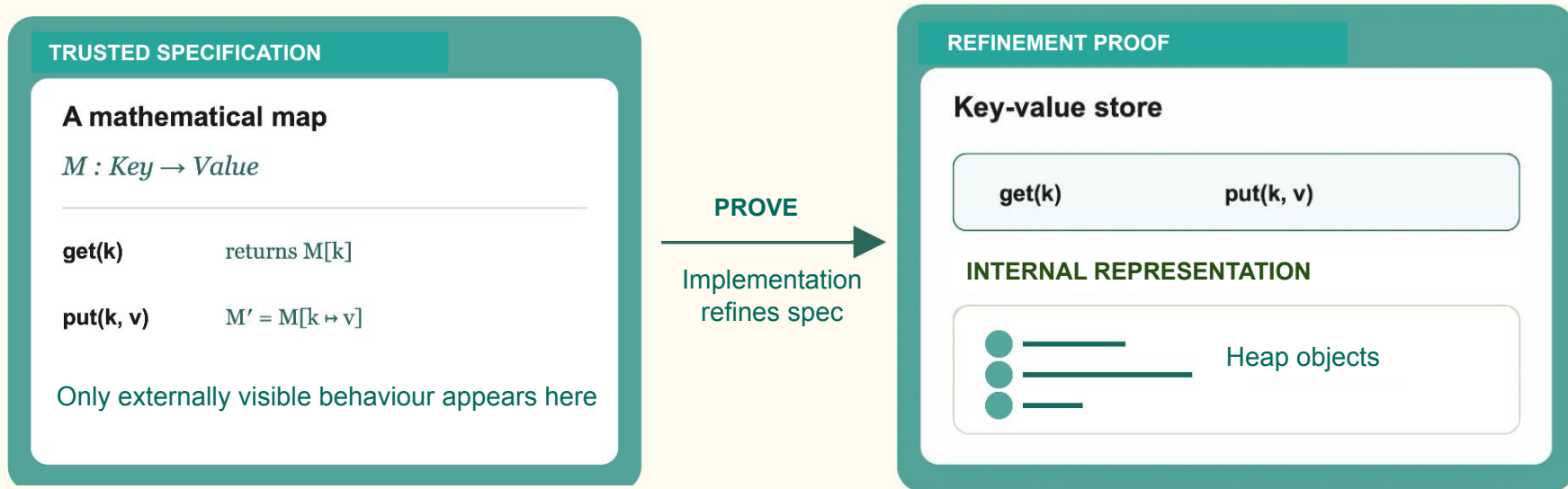
The University of Texas at Austin

Reto Achermann

Technical University of Munich

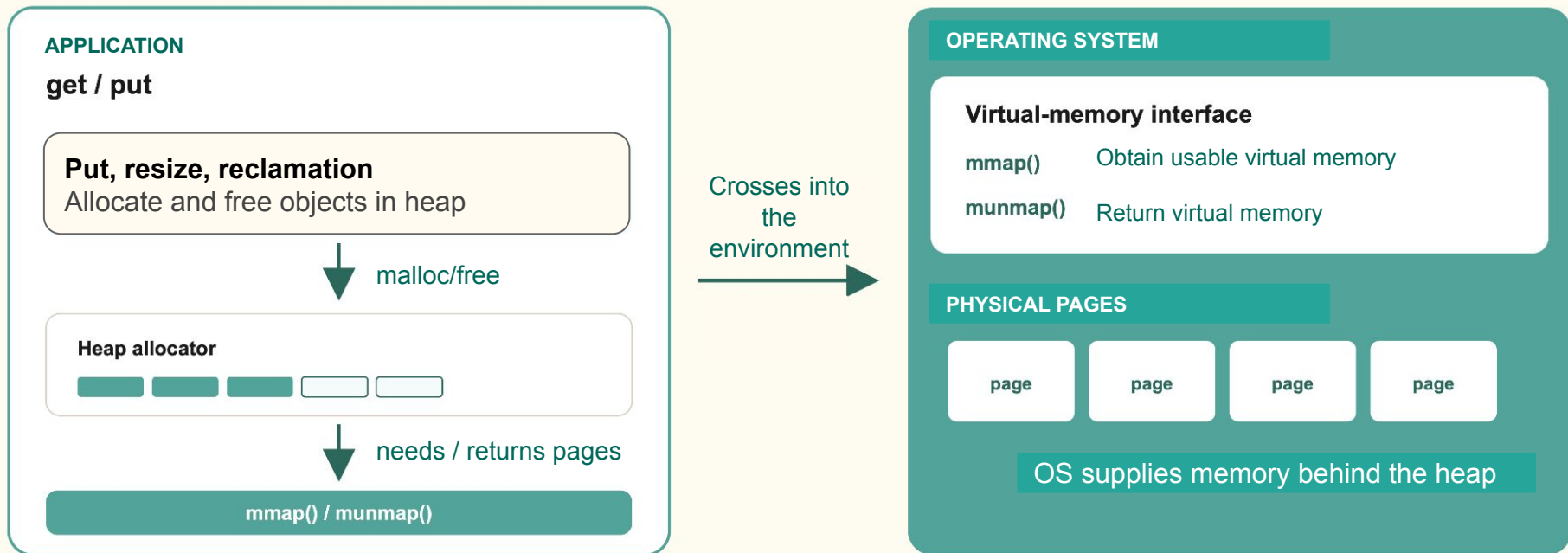


Verifying a Key-Value Store

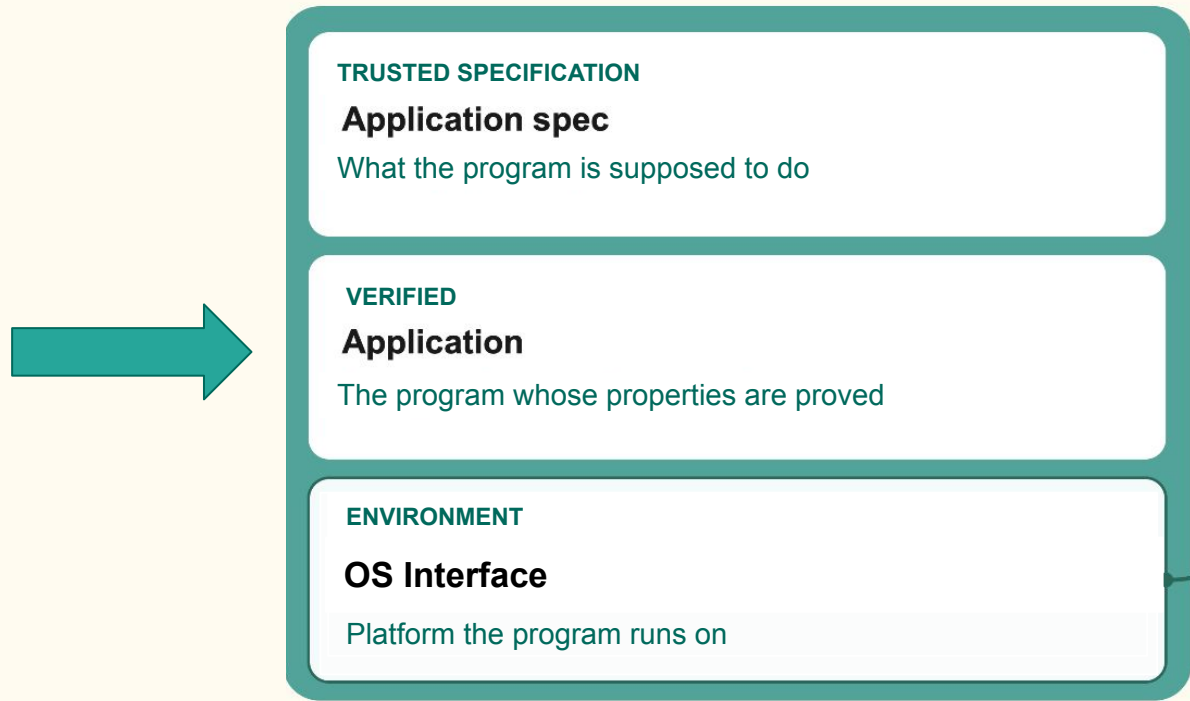


Proof: The implementation correctly realizes the specification.

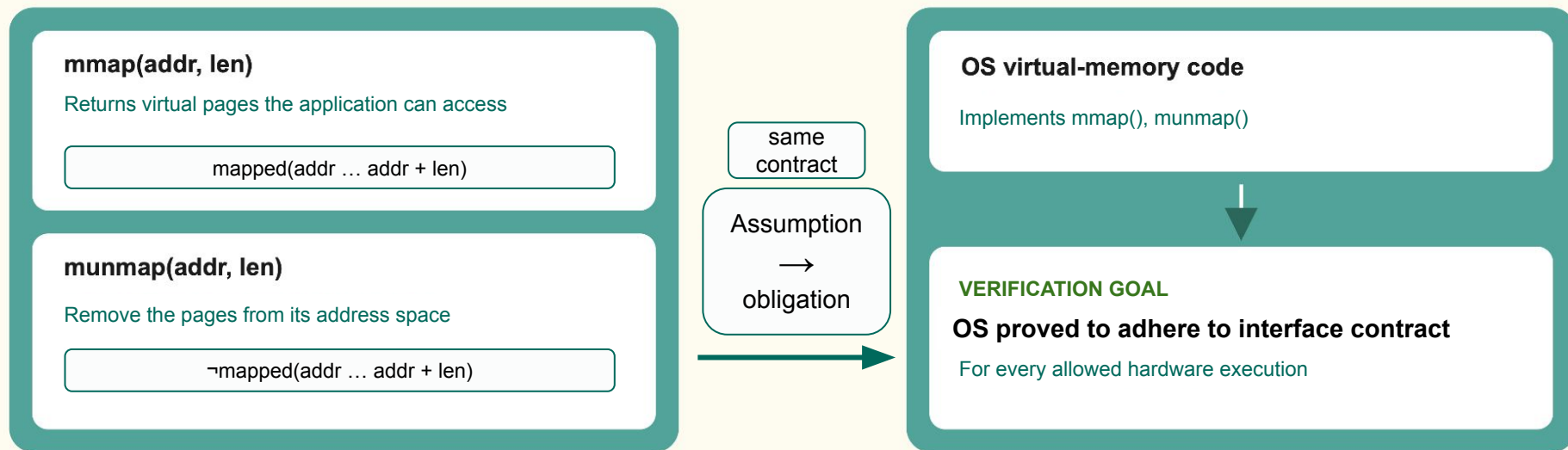
Applications do not run in a vacuum.



Verification against Application + Environment Spec

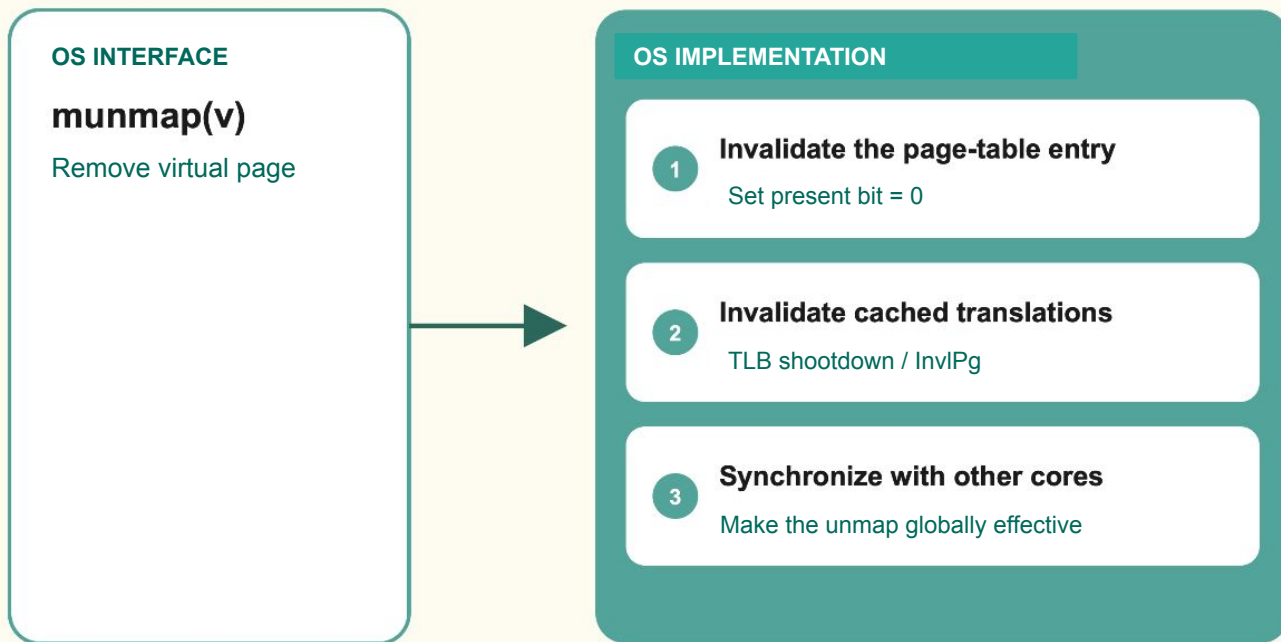


Application Environment == Operating System Spec



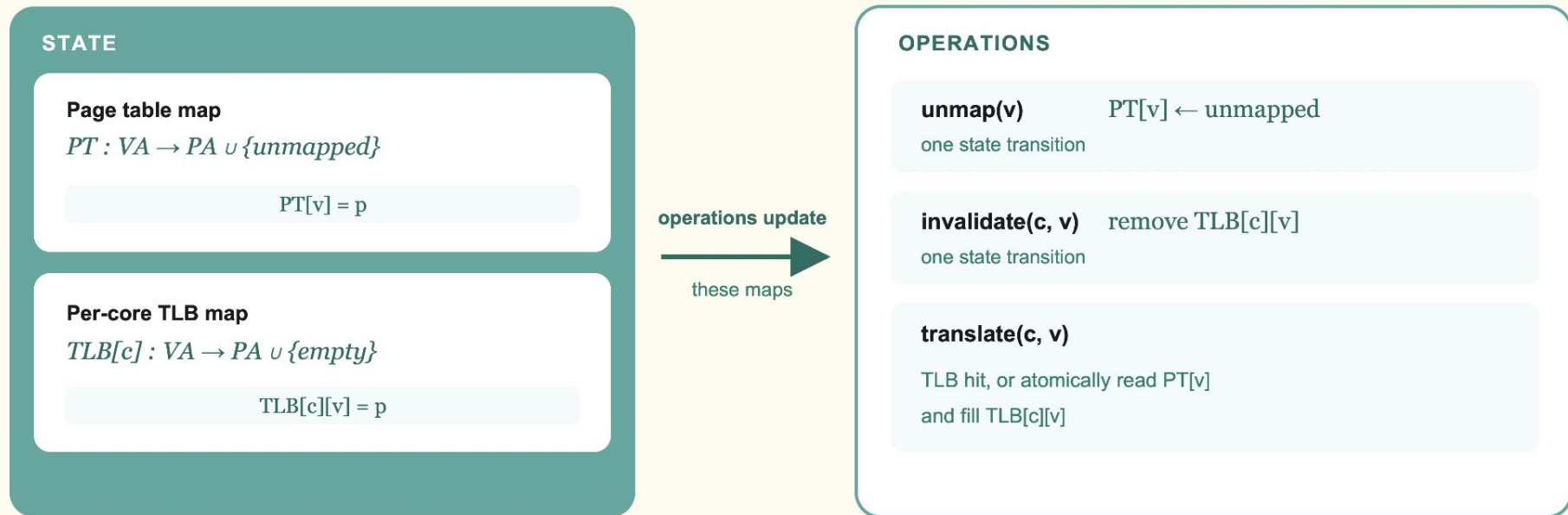
To verify the OS, we need a specification of its environment - **The hardware**

Implementing the OS Interface requires MMU Interaction



How should the MMU be modeled?

A (too) Simple MMU Model



Issues with this?

This model misses some important observed behaviours

1 Page walks are not atomic

Simple model

$PT[v] \rightarrow p$

Real walk



PTE write can interleave

Flattening erases partial walks
and their interleavings

2 Store buffers and data caches are not represented

Simple model: Updates are globally visible immediately

$PT[v] \leftarrow \text{unmapped}$

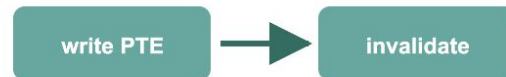
Hardware



A memory model is required

3 Hardware takes hidden steps

Visible program events

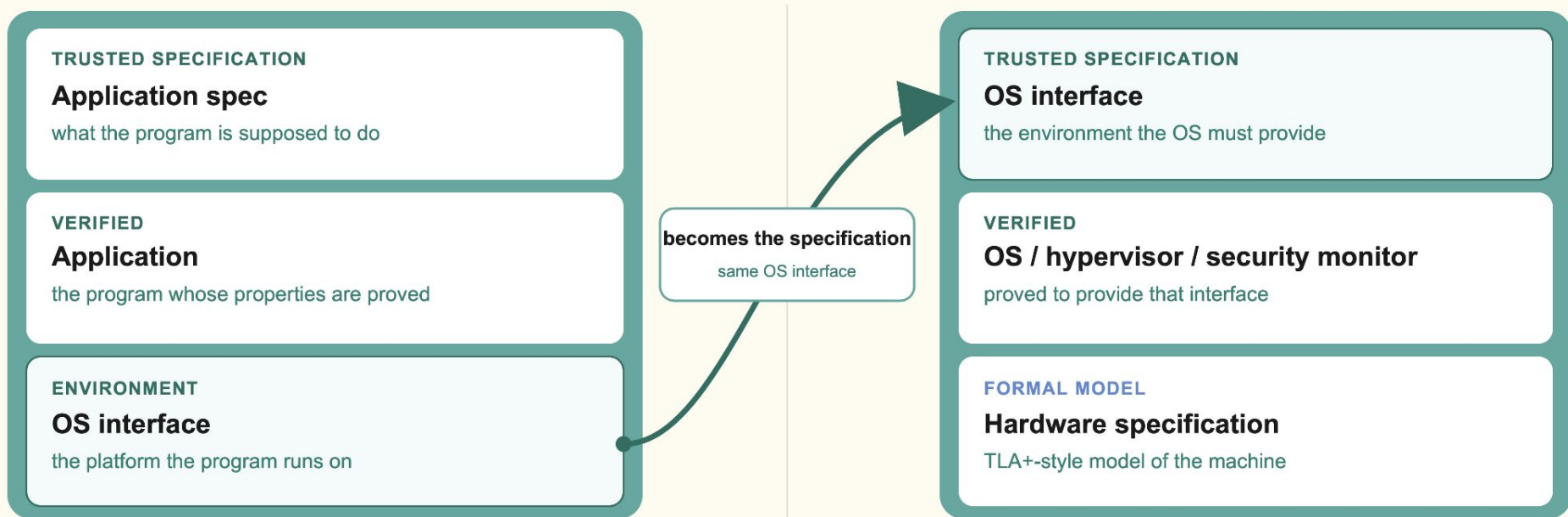


Possible hidden execution

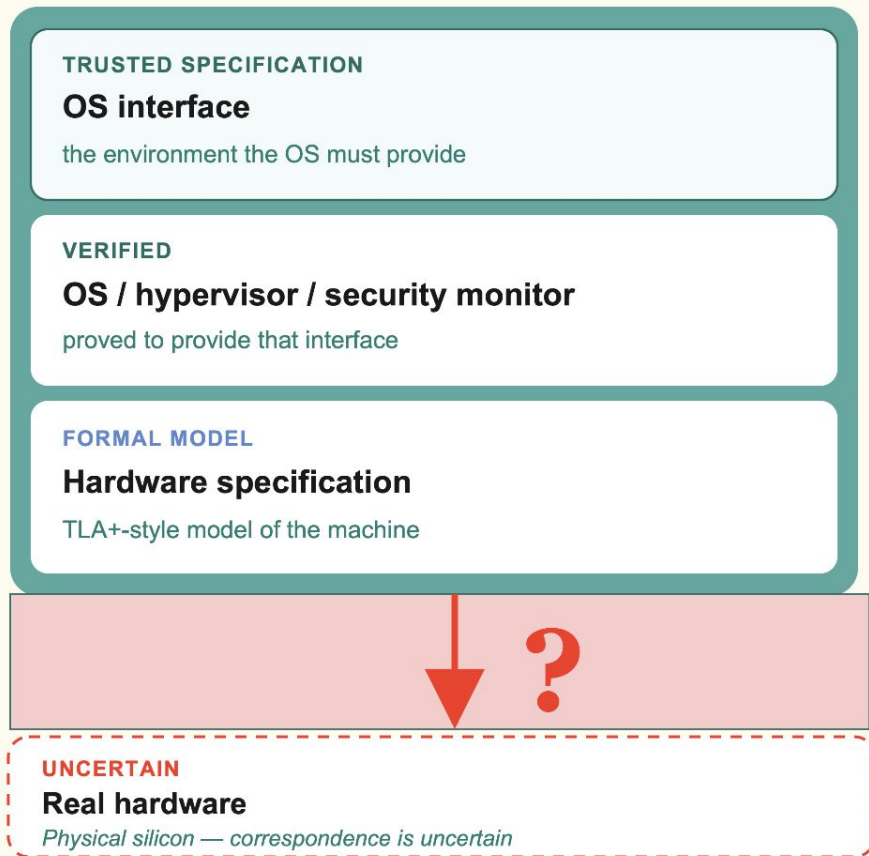


A stale translation may be created
by steps software never observes

Verification is Rooted in the Hardware Specification



Hardware Specification matches the Real Hardware?



Writing an accurate hardware spec is hard



Formal specifications can contain bugs [12]

Those bugs can propagate into verified implementations. [14, 23]



Model-reality gap [7, 13]

The formal model and real hardware admit different behaviors.

That mismatch makes proofs incomplete or more difficult than necessary.



Invisible microarchitectural steps affect observable behavior

Page walks, TLB events, cache activity, and writebacks can change what software observes.

We need a systematic way to validate formal models against real hardware. [5, 23]

[5] Armstrong et al., POPL '19 / [7] Bognar et al., IEEE S&P '22 / [12] Fonseca et al., EuroSys '17

[13] Gleirscher et al., CSUR '19 / [14] Goldweber et al., OSDI '24 / [23] Reid, FMCAD '16

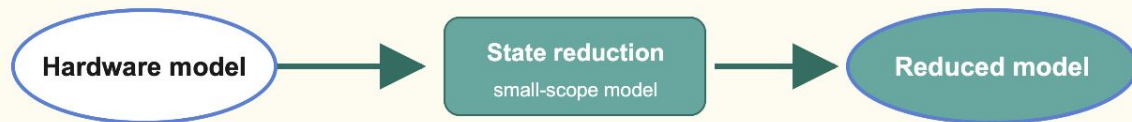
How can we build an automated hardware model validation pipeline?

Validating a model against real hardware

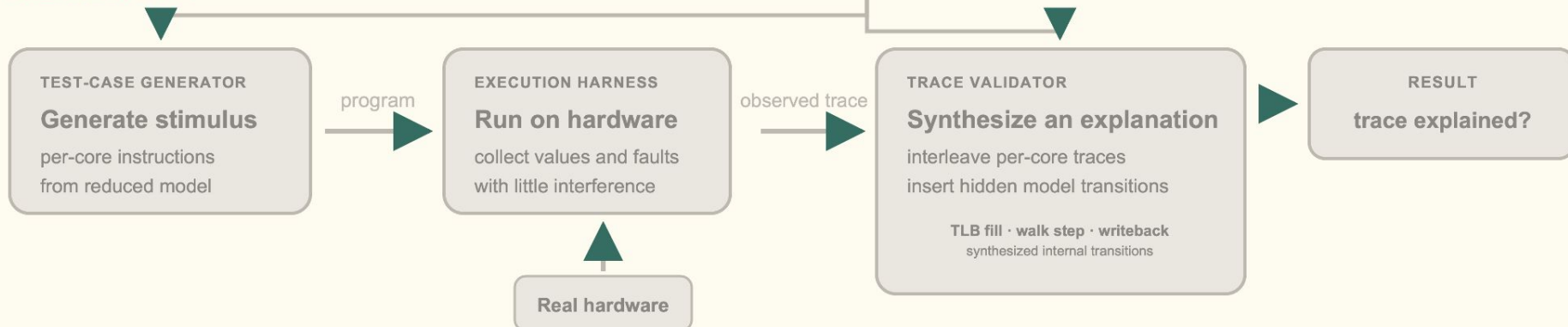


Towards an automated validation pipeline

MODEL PREPARATION



VALIDATION LOOP

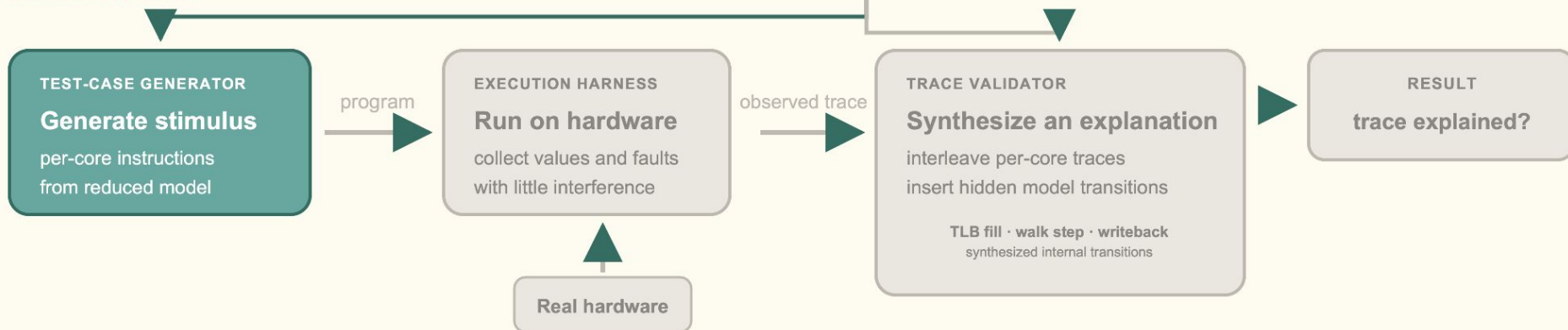


Towards an automated validation pipeline

MODEL PREPARATION



VALIDATION LOOP

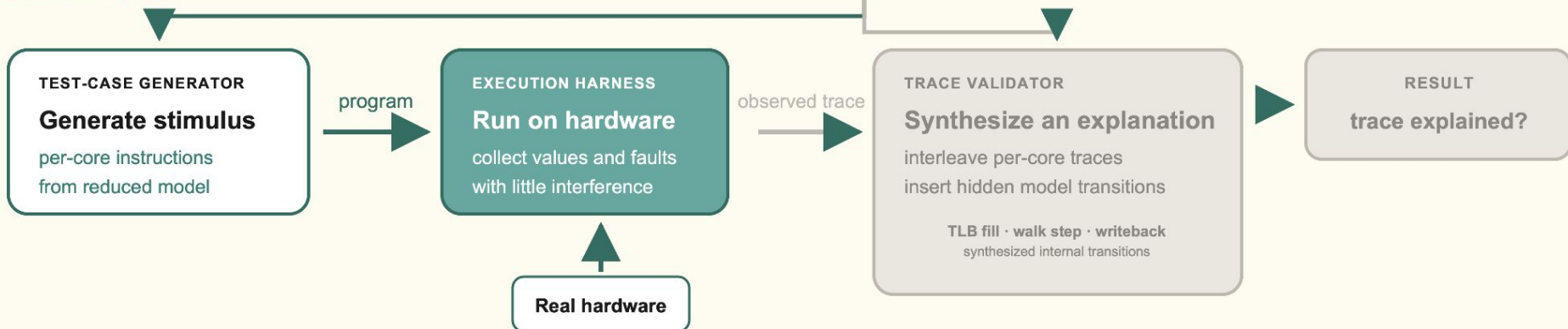


Towards an automated validation pipeline

MODEL PREPARATION



VALIDATION LOOP

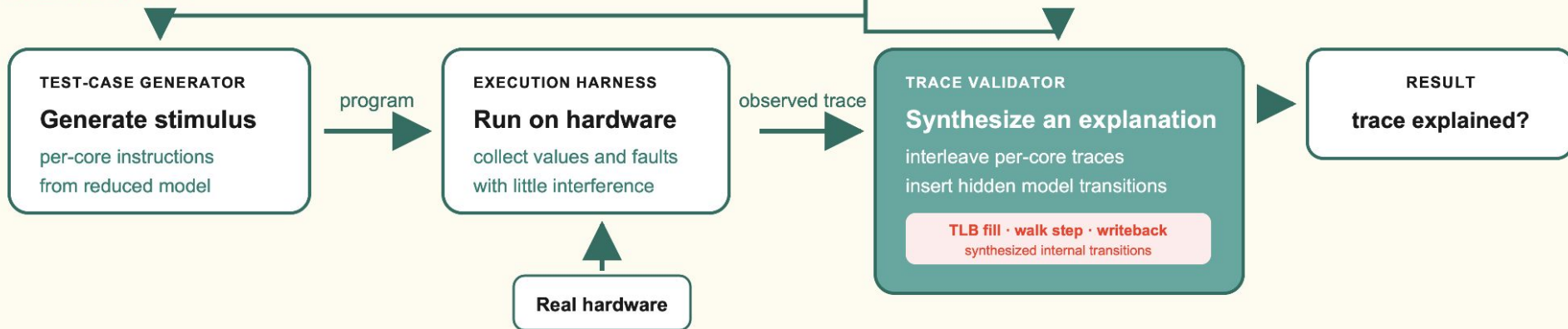


Towards an automated validation pipeline

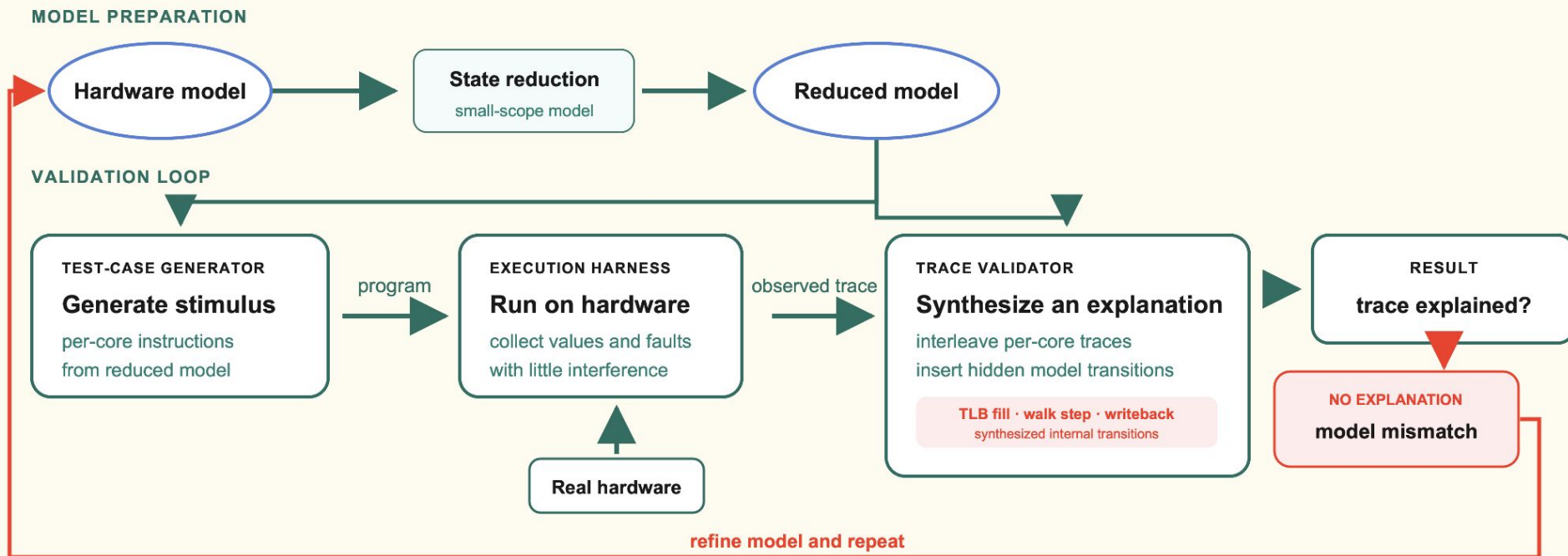
MODEL PREPARATION



VALIDATION LOOP



Towards an automated validation pipeline



Mismatch can indicate bugs in model, existing OS implementations or hardware

Challenges

1 Generate useful stimulus programs

Choose per-core instruction sequences and parameters that exercise interesting translation and caching behavior.

2 Run with little interference

Kernel activity, synchronization, and recording outcomes can perturb the translation and caching behavior being measured.

3 Match traces with hidden events

Observed traces omit page walks, TLB events, cache activity, and store-buffer writebacks needed to explain each outcome.

4 Control state-space explosion

Choices grow with cores, pages, paging levels, trace length, and the grammar of possible internal transitions.

Exhaustive testing is infeasible

1 Cores

a few → 100 cores

each core adds another concurrent instruction stream

2 Virtual pages

$$2^{(48-12)} = 2^{36}$$

4 KiB pages in a 48-bit virtual-address space

≈ 68.7 billion virtual pages

3 Physical frames

$$2^{(52-12)} = 2^{40}$$

4 KiB frames with the architectural 52-bit physical-address limit

≈ 1.1 trillion physical frames

4 Page-table paths

$$512^4 = 2^{36}$$

512 entries at each of four page-table levels

before varying contents, permissions, and page sizes

We must reduce the state space to make this tractable

Model reduction via Small State Hypothesis

Small state hypothesis:

“Most behaviors observable at full scale are also observable at small scale.”



Not a proof — a principled assumption standard in bounded model checking.

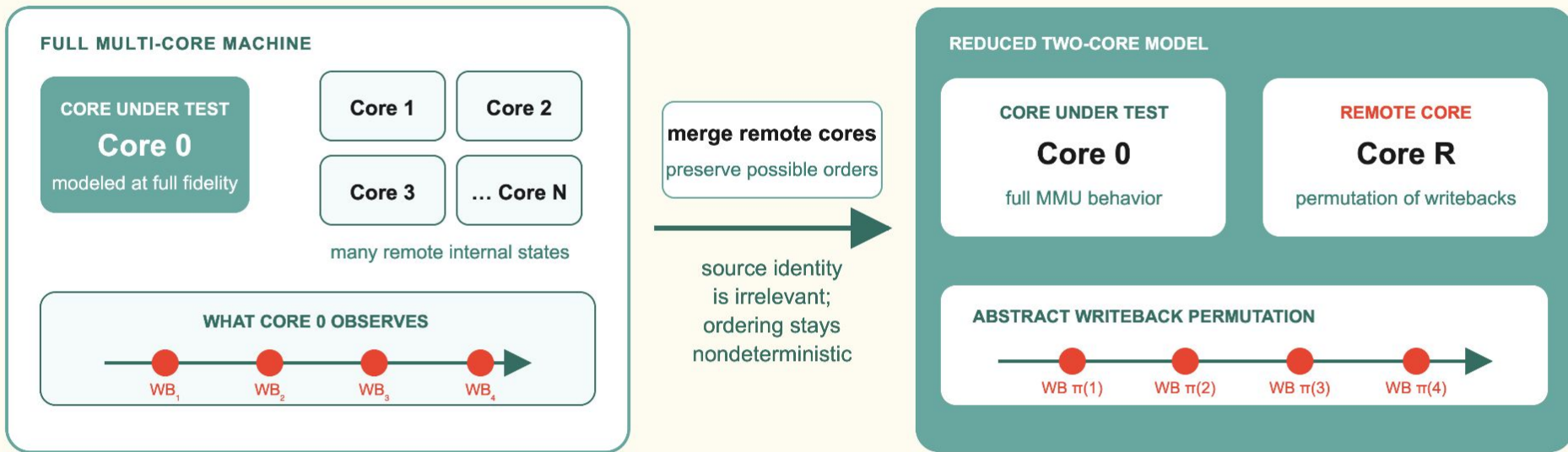
State Reductions applied to x86 MMU

Dimension	Reduced to	Why
Cores	2	"Core under test" + one merged "remote" core for writebacks
Paging levels	2 (PMD, PT)	Lowest two levels expose walk non-atomicity and superpages
Virtual pages	3	Two share a PT page (exercise the PWC); one diverges early
Virtual addresses	4	One per page, plus one extra to exercise a TLB-cached mapping
Physical frames	4	One spare frame lets a page remap without creating an alias

Informal justifications, not formal proofs — principled application of the small-scope hypothesis.

Example Reduction: Many cores - two roles

Preserve what the core under test can observe; abstract everything else.

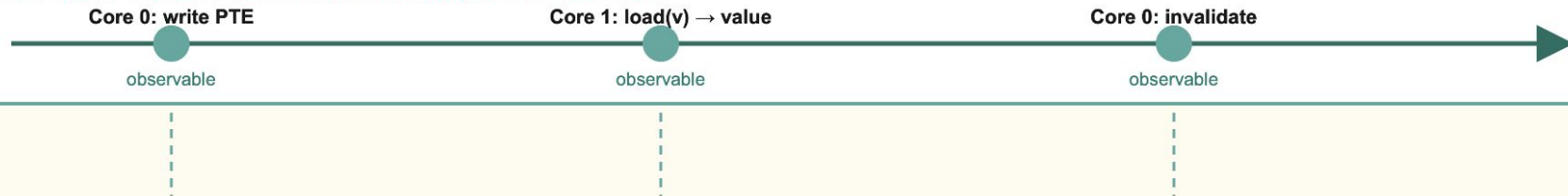


Divide-and-Conquer: Split by selecting each core as the core under test with one abstract remote core. Then merge the results

Trace Validation: Observed events vs full trace

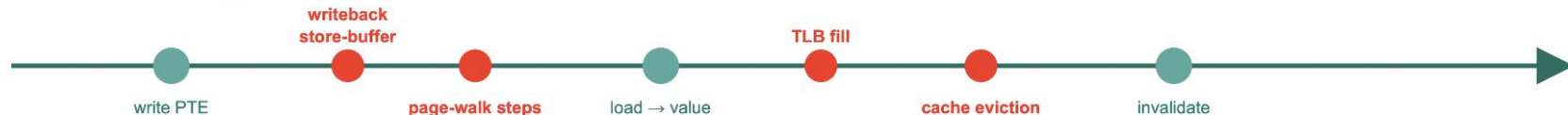
OBSERVED TRACE

Software-visible instructions and outcomes collected from each core



A COMPLETE EXECUTION

The same observable steps, interleaved with internal microarchitectural transitions



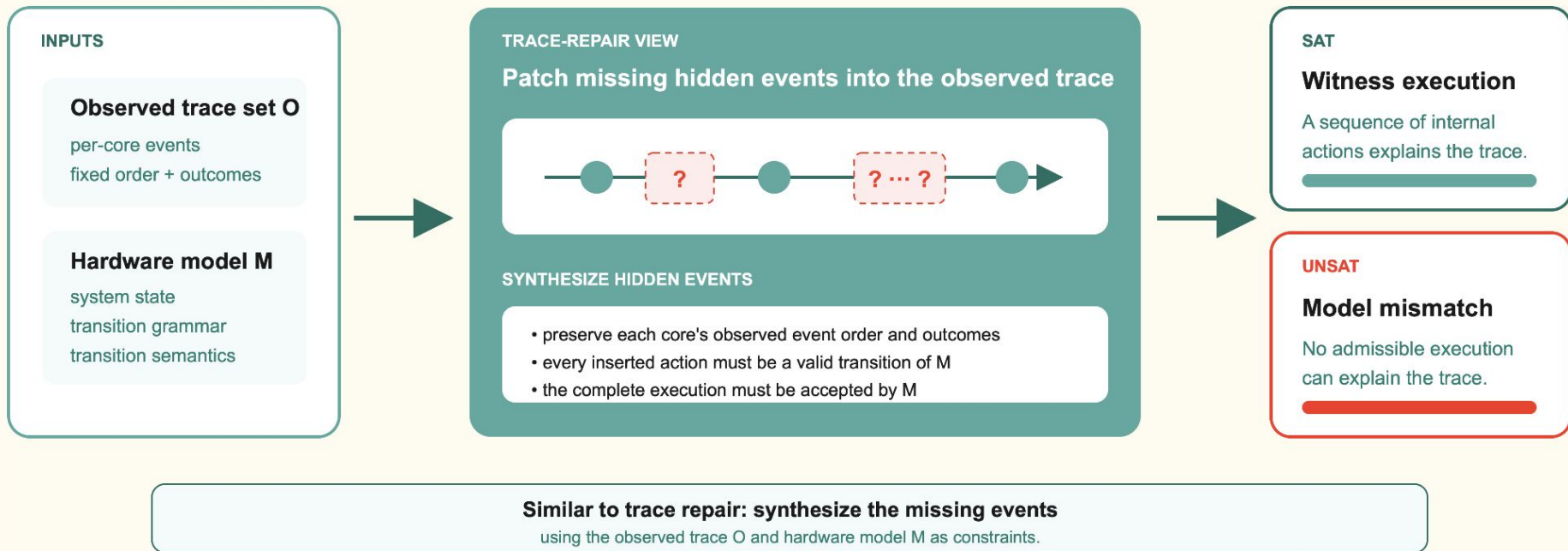
● observable: fixed by the trace

● internal: not recorded, but constrained by the model

From the test harness we only get the observable hardware events.

Trace Validation through Software Synthesis

Find at least one set of hidden events, allowed by the model, that explains the observed trace.



Evaluation

Prototype Evaluation 1: Model Reduction

CAPABILITY ARGUMENT

The reduced model retains:

- 1 **two cores**
enough for both threads
- 2 **two distinct virtual addresses**
one location per thread
- 3 **stores, loads, and x86-TSO**
including store-buffer behavior
- 4 **parameterized operations**
can choose va_1 and va_2

high-level strategy analysis



LITMUS TEST EXPRESSIBLE BY THE STRATEGY

Core 0

Write(va_1) = 1



Read(va_2) → initial value zero

Core 1

Write(va_2) = 1



Read(va_1) → initial value zero

Both reads observe the initial value

allowed by x86-TSO · forbidden by sequential consistency

Result: the strategy is capable of expressing the classic store-buffer litmus test

Future validation: repeat this analysis across a broader suite of litmus tests.

Prototype Evaluation 2: Trace Validation

INITIAL MODEL STATE

- 1 TLB is empty**
No cached translation initially enables either read.
- 2 Two-level page table has one valid mapping**
 $v_0 \rightarrow p_0$ · first byte at p_0 contains 42
- 3 Every other virtual page is not present**
 $v_1 \notin$ page table
- 4 One speculative walk for v_0 is already in progress**
The walk started before execution and can supply a TLB entry.

OBSERVED CONSTRAINT · CASE 1

Successful read of the mapped page

```
memop_read(core 0, v0) → 42
```

OBSERVED CONSTRAINT · CASE 2

Successful read of a non-present page

```
memop_read(core 0, v1) → 42
```

For each case, ask the synthesizer to complete the trace with model-allowed internal steps.

The initial state is identical; only the observed virtual address changes.

Prototype Evaluation 2: Trace Validation

CASE 1 · GENERATED COMPLETION

The existing walk can justify the read

INITIAL STATE

empty TLB + completed page walk for v_0



SYNTHESIZED INTERNAL STEP

`tlb_fill(core 0, walk( $v_0$ ))`



OBSERVED STEP NOW ENABLED

`memop_read(core 0,  $v_0$ ) → 42`

CASE 2 · REJECTED

No hidden execution can justify the read

INITIAL STATE

v_1 is not present + no page walk for v_1



SEARCH OVER HIDDEN STEPS

No TLB fill is possible; internal steps cannot update the page table to create the missing mapping.



UNSAT · NO TRACE GENERATED

`memop_read(core 0,  $v_1$ ) → 42` is rejected

Proof of concept: generate a missing internal action when one exists; reject the trace otherwise.

Future Works

1

Scale synthesis

• Abstract non-test cores • use big-step page walks • add domain-knowledge constraints

2

Formalize the reduction arguments

Move from informal small-scope justifications toward proof obligations and inductive proofs

3

Automate and broaden validation

Connect the full pipeline end to end · evaluate more tests, traces, and hardware configurations

Summary

Problem: Verified software is only as sound as its hardware specification — hidden microarchitectural behavior can make the model diverge from real hardware

Tractable Hardware Model Validation: Check whether the model can explain observable hardware traces

- **Synthesis-based validation** — reconstructs missing internal microarchitectural steps
- **Model reduction** — makes x86 MMU validation tractable and adapts to other models
- **Prototype implementation** — evaluated on x86 paging behavior as an early proof of concept

Please reach out to me at:

Rishika Varma · rishikavarma@utexas.edu