Hopscotch: A Scalable Flow-Graph Based Approach to Formally Specify and Verify Memory-Tagged Store Execution in Arm CPUs

Abhinav Sethi*
9408 Muskberry Cove, Austin TX 78717
phone: 979-985-0995 email: abhinav.sethi@arm.com

Madhu Iyer*
7021 Vicenza Dr, Austin TX 78739
phone: 832-660-3994 email: madhu.iyer@arm.com

Sai Komaravelli**
phone: +91-7337-09107 email: sai.komaravelli@arm.com

Vikram Khosa*
909 West Johanna Street, Unit B, Austin TX 78704
phone: 512-431-0302 email: vikram.khosa@arm.com

* Arm Ltd.
Bldg 1, 5707 Southwest Pkwy #100
Austin, TX 78735
Phone: 512-327-9249

**Arm India
Bagmane World Technology Center-SEZ, Citrine Block 5th & 6th Floor
Marathahalli Outer Ring Road, Doddanakundi Village, Mahadevpura
Bangalore-560048
Karanataka, India

***Abstract* - This paper describes Hopscotch, a novel flow-graph based scalable framework used to formally specify and verify a complex RTL implementation of Arm v8.5 Memory Tagging Extension (MTE) functionality on the Load-Store unit of a high-performance Arm CPU.**

GLOSSARY

CPU: Central Processing Unit
DUT: Design-Under-Test
E2E: End-to-End
GO: Globally Observed; the point at which the effects of a read or write operation become visible to all observers
IVA: Initial-Value Abstraction
LSU: Load-Store Unit in a CPU that manages all load and store operations
MTE: Memory Tagging Extension (Arm v8-A Architecture)
OC: Over-Constraint
Store-Exclusive: A type of Store instruction in the Arm architecture used to obtain Exclusive access to a location in memory, paired with a Load-Exclusive instruction (and guarded by an **Exclusive-Monitor**)
TB: Testbench

I. Introduction And Overview

A. Load-Store Unit

The **Load-Store Unit (LSU)** is typically the most complex (by an order-of-magnitude) unit in a high-performance superscalar, out-of-order, cache-coherent CPU and presents unique challenges to verification. Formal verification is no exception.

B. Design and Formal Complexity

In theory, formal techniques are well-suited to exploration of extreme corners that arise from the asynchronous nature of out-of-order issue of memory access of multiple types, attributes etc. as well as cache-coherence traffic.

However, the sheer amount of logic and state dedicated to implementing this functionality in RTL as well as state-tracking require to correctly constrain related interfaces also creates state-space explosion for formal tools (reflected by first-order measures of sequential complexity such as design and testbench flop-counts).

For any high-level assertions written naively to verify end-to-end behavior, this complexity inevitably severely limits both stimulus coverage and the adequacy of proof bounds achievable.

C. Complexity Mitigation

Well-known mitigation techniques e.g., case-splitting, deep abstractions, design-mutations, helper-invariants etc. can be quite effective in reducing this complexity even as they come with their own set of costs and tradeoffs.

Nevertheless, achieving required proof-depth can still be a challenge for high-level checkers for key architectural and microarchitectural behaviors using state-of-the-art tools, even after applying these techniques.

D. A Fresh Approach

To address this challenge, we have developed a unique framework called "*Hopscotch*", which enables us to (A) formalize and abstract the specification of distinct execution-flows in a design in terms of execution events and their ordering relationships and (B) use this formalism to decompose the formal verification problem and increase its tractability.

E. Overview

The rest of this paper is divided into 7 key sections:

1. First, we introduce key architectural principles and components of the Load-Store formal testbench environment.
2. Second, we provide more details of existing complexity-mitigation strategies deployed to achieve better formal coverage.
3. Third, we present an overview of the *Memory-Tagging Extension (MTE)* of the *Arm* architecture, its microarchitectural implementation and its verification complexity, all in the context of our effort to formally verify this feature.
4. Fourth, we describe the rationale behind (including the choice of graph-based implementation for) the *Hopscotch* framework used to rapidly create and easily maintain abstracted event-based specifications of design functionality as well as build and test a set of interlocking component checkers to verify the former.
5. Fifth, we present a detailed description of our key contributions: *Hopscotch* as a structured methodology, which also addresses the scalability problem by allowing decomposition of scope for both specifications and checker along two additional axes: the set of distinct *execution flows* supported by the design and the set of required (or allowable) *execution events* along each *execution flow.*

6. Sixth, we present a case-study of the application of Hopscotch methodology to verify the implementation of *MTE* functionality for *Store-Execution* in the *Load-Store Unit* of an Arm CPU. We also discuss results from this implementation, including a sample of representative design bugs found.
7. Finally, we discuss the current limitations of and potential future improvements and extensions to the Hopscotch methodology.

II. Existing Approach to End-to-End Checking

A. End-to-end Verification Principles, Primitives and Components

The key design principles of the Load-Store E2E formal environment (see **Figure 1** below) are **(A)** *Separation of Concern*: Each component implements a single clearly defined function. Interfaces across components are clean and well-defined. and **(B)** *Hierarchical Information Flow*: For a given formal trace, the set of choices/values non-deterministically generated by *oracles* uniquely determine a specific ***Trace-Scope*** of Checker/*DUT* activity. The lower layers of the testbench pass an abstracted view within the *Trace-Scope* into the upper layers and filter out observations of *DUT* activity outside of it.

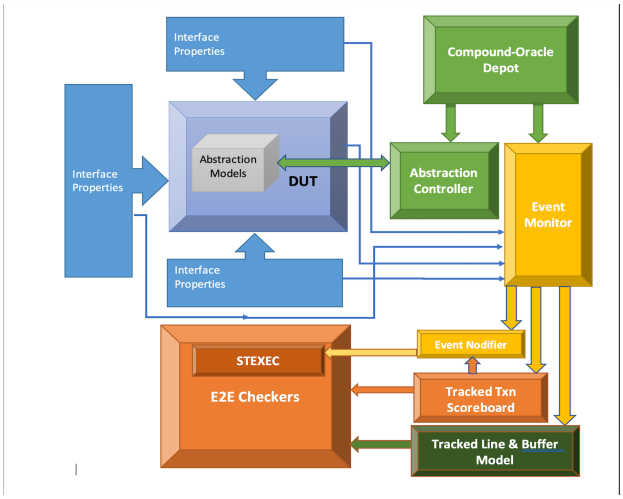

Fig. 1. E2E Formal Testbench Architecture

We provide formal definitions of the non-deterministic building-blocks of the Load-Store E2E environment below:

- **Choice-Variables** represents one of A) the value of a *DUT* signal or bus directly visible to (controllable and/or observable by) the formal environment, B) active *E2E* formal checker selection (including case-split attributes) and C) abstraction behavior or policy selection.
- An **Oracle** or a **Free Variable** can be used to generate arbitrary but legal (constrained) values for a *Choice-Variable.* These provide a way to generalize the formal analysis, considering cases involving each possible value of the variable all at once. When the formal environment contains some level of symmetry to be exploited, these variables can significantly increase the efficiency of formal analysis **[4].** A legal value delivered by an *Oracle* per invocation is called a **Value-Binding**.
- A coherent grouping of such *Oracles* that generate mutually consistent *Value-Bindings* under a specified context via a network of constraints is known as a **Compound Oracle**. Its function is to *filter* design activity

visible to, tracked by or checked by formal abstractions and/or checkers respectively. It is typically held stable across a given trace e.g., restricted to specific addresses/translations, ops, flows etc.

F. Oracle-based Event Filtering Approach

Figure 2 below lists the key Compound Oracles that collectively determine the specific *Trace Scope* visible to a given check in the E2E environment.

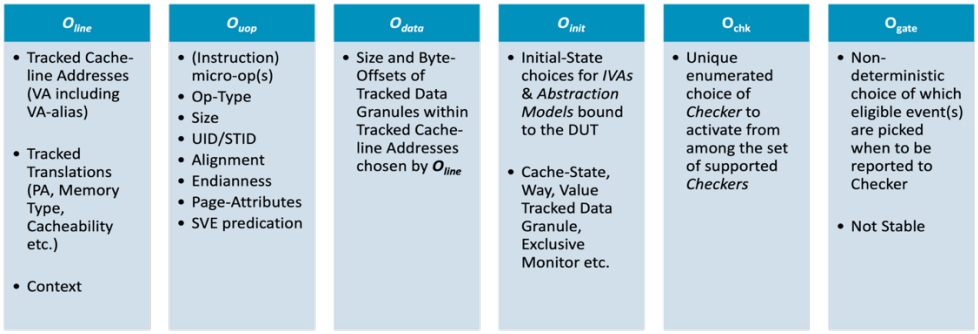

Fig. 2. Key Oracles

G. E2E Testbench Components

The E2E **Event Monitor** is a low-level, stateless, filtering component and has three functions: **(A)** ***Listener*** monitors various interfaces for events of interest **(B)** ***Filter*** selects only events that match the *Value-Binding*s selected to by the respective *Compound Oracle* and **(C)** ***Broadcaster*** bundles up filtered events into transactions and passes them to other components.

The key E2E state-tracking components are (A) ***Scoreboard***, which tracks filtered requests and their state e.g., loads, stores, snoops, barriers etc. **(B)** ***Model,*** which models *Tracked Cache-line*'s *Coherence-State* and *Data* (bit) and allows for valid *State/Data* IVA (bound to O_{init}), It also models an *Abstract Store-Buffer Model* and *Abstract Merge-Buffer Model* for the *Tracked Line & Data*. **(C)** ***Abstraction Controller***: enforces consistency required across *Value Bindings* in *E2E Checkers, Abstraction Models* and *IVAs*. It is implemented via an assume network, supported by a lightweight tracker.

III. PAST COMPLEXITY MITIGATION MEASURES

These include the following interventions:

- *Formal Abstractions* include abstraction models, initial-value abstractions as well as abstractions for prediction structures, counters and configuration registers.
- *Design Mutations* include those for ID widths *(*shrinks both design size and interface trackers), buffer sizes, counter widths (watchdog timers, arbitration saturation counters). Multiple design configurations resulted in state-element count reduced by an order-of-magnitude (variable), including a 4-5X reduction in build time for smallest config.
- *Structured Case-Splitting is* implemented for key, hard E2E checkers. Full proofs were achieved for extreme case-splits, further accelerated by a handcrafted suite of helper assertions. Each case-split variable enables sensitivity analysis by measuring its effect on proof-convergence and indirectly, its contribution to formal complexity.

- *Over-Constraint based Complexity Management* is implemented using both static and dynamic approaches) ***(A)*** *Static Transaction-Limiting Profiles*: Create a reference set of *OC*s from a combination of stimulus at interfaces, checker-selection oracles, abstraction-policy oracles etc. Limiting include disabling specific types of stimuli, limiting number of transactions of each type of enabled stimulus, narrowed/unique choice of oracle values etc. Map each *OC* into corresponding *`define*, concatenate *`defines* into a set of named, unique "profiles", which are allowed to be specified at build time ***(B)*** *Dynamic OC-Recombining*: Create a static pool of weighted, abstracted *OC*s with attributes, value distributions, and cross-dependencies (e.g., affinity, mutual-exclusion). On each invocation, the *Loc-K-Picker* tool picks a random set of a minimum number ("K") of mutually consistent, concretized **L**ocal **O**ver-**C**onstraints, based on a solution to the *Knapsack Problem.* This can be dynamically (including interactively) applied on a per-task basis such that each task/proof-thread gets a unique set of selected OCs.
- *Complexity Hotspot Detection and Mitigation:* Several tool-specific techniques have been deployed to diagnose and remedy complexity bottlenecks (including dead-ends). Discussion of these techniques is outside the scope of this paper.

IV. Arm® Memory-Tagging Extension, Design Implementation and Verification Complexity

A. Arm® Memory-Tagging Extension (MTE) Overview

The *Arm® Memory-Tagging Extension (MTE)* **[2,3]** aims to increase the memory safety of code written in unsafe languages without requiring source changes or, in some cases, recompilation. Easily deployable detections of and mitigations against memory safety violations may prevent a large class of security vulnerabilities from being exploitable.

Memory unsafe languages allow unintended data corruption or unauthorized access to sensitive data. Violations of memory safety fall into two main categories: *spatial safety* and *temporal safety*. *MTE* provides a mechanism to detect memory safety violations of both *spatial safety* (e.g., bounds overflow) and *temporal safety* (e.g., out-of-scope access to reallocated memory.

MTE implements lock and key access to memory. Locks can be set on memory and keys provided during memory access. If the key matches the lock, the access is permitted. If it does not match, an error is reported. Memory locations are tagged by adding 4 bits of metadata to each 16 bytes of physical memory, which forms a *Tag Granule*. Tagging memory implements the *Lock*. Pointers, and therefore virtual addresses, are modified to contain the *Key*. *MTE* relies on the *Lock* and the *Key* being different to detect memory safety violations.

Algorithm: **(A***)* Assign a color *("tag"*) to each memory allocation **(B***)* Store color in unused high bits of address used to reference that location (pointer) **(C)** Match color for each reference against stored color prior to access **(D)** Reassign color when freeing allocation.

B. MTE Nomenclature

To disambiguate ***"D**ata" **A**ddress **T**ags* in the L1 cache
from ***L**ogical **A**llocation **T**ags* and ***P**hysical **A**llocation **T**ags*, we abbreviate the former to *DAT* and the latter to *LAT / PAT.* Checked and Unchecked accesses are abbreviated as *CHK* and *UCHK* respectively. We Abbreviate *Precise* and *Imprecise LAT vs. PAT* Checking Modes to *PRC* and *IMP* respectively.

C. Architectural and Micro-architectural Rules

All *CHK* Loads/Stores to memory carry the *Lock* as a *Logical Address Tag* (*LAT*). 4 bits of Key as *Physical Address Tag* (*PAT)* are stored per 16B granule of memory (*Ke*y). MTE enables are configurable at each *Exception Level* as

well as at page granularity. *LAT* for *Checked* Loads/Stores are compared against *PATs* for all overlapping granules in memory (cache). There also exist separate instructions (*LDG** and *STG**) to read and write *PATs*.

There are two supported *MTE Tag-Check* modes. In *Precise Mode* (enabled during software testing*)*, each *CHK* load or store that fails *Tag-Chec*k requires a *Synchronous Abort*. A *checked* store requires a Tag-Check to succeed before it merges, incurring a high performance overhead. In *Imprecise Mode* (enabled in production), store execution is not gated by *Tag-Check* success. Stores that fail *Tag-Check* allow *Asynchronous Aborts* and incur low performance overhead.

IV. Evolution to a more scalable approach

A. MTE verification planning

MTE was a novel, high-complexity feature tacked onto an existing Load-Store micro-architecture. Its predominant impact lay on the Store Path in the Load-Store unit.

The formal verification of MTE was scheduled, planned, and executed quite late in the project cycle, a few months before production release. At the time of planning, formal bring-up of the Load-Store formal environment for this CPU design only covered support for basic stimulus (loads, stores, and snoops). A Load-Value Checker was the only E2E checker to cover any store-side checks. However, this checker had historically suffered from high formal complexity (predominantly via the store-to-load forwarding logic) and did not promise sufficient formal coverage, even with intensive application of complexity mitigation strategies described above.

B. The path to Hopscotch

Given that the Store-Path dominated the scope of design changes for the *MTE* implementation, the key target execution flow for verification was that for a *CHK* Store. The latency from issue of a Store *μOp* to the point of *GO* would already stretch the limits of a monolithic E2E checker.

Initial planning indicated multiple independent architectural and microarchitectural checkers to verify Store Path execution across different tag-check modes, MTE attributes, op-types, alignments etc. This pointed to significant overlap in tracking required for each such checker and therefore, duplication of both solution space and effort. Given the late stage of formal deployment and the narrow window till release, this strategy did not meet schedule, resource, and quality requirements.

Searching for a more cost-effective approach compelled us to envision a more systematic, integrative, preferably automatable solution, leading to the idea behind *Hopscotch*. The key motivation was to develop a conceptually unified framework, spanning the entire lifecycle of an arbitrary Store instruction.

The methodology would **(A)** identify key events (architectural & micro-architectural) across multiple execution flows, **(B)** specify legal (or illegal) event orderings for each flow (mode, instruction-type, other attributes) and **(C)** at each event occurrence, trigger checks for both safety and liveness relative to predecessor or successor events respectively, as well as key off other function checks (tag-check correctness, data-consistency etc.) on a subset of eligible events. The resulting framework promised the advantages of being flexible, modular, iterative, scalable, and reusable.

C. Design Goals

The design-goals driving the architecture of *Hopscotch* were that it allows for

- a clean separation between *(A)* a user-defined layer of functional specification expressed in terms of ordering requirements for abstracted design events using simple temporal operators, and *(B)* a static code-

substrate operating on domain-agnostic, configurable, regular structures, which translates, stores and executes the specification (both checking and coverage)

- ease of maintenance: specification updates can be clearly captured and interactively tested (counterexample-guided) and have little impact on underlying executable codebase

- ease of decomposition supports both structured case-splitting and path-decomposition, which are both key to mitigating formal complexity.

D. Choice of Graph-based Implementation

One of the more obvious solutions to tracking the lifetime of a Store is to design a conventional tracking structure akin to either a single FSM or a scoreboard which allows for each execution event to transition tracked state among a finite set of predefined states. A scoreboard may be considered a special case of FSM where the state-bits are typically stored in a decoded form across sets of Boolean flags, enumerations and/or counters and any updates to or checks/covers based on these state-bits are triggered inside closely coupled procedural code, usually written in an ad-hoc (irregular) style.

Any changes to design or the choice of events tracked will invariably perturb procedural code that performs next-state computations of the tracked state and/or its correctness checking. Given a complex enough design/feature with enough inter-related events of interest, that complexity is invariably transferred to such an ad-hoc implementation of the tracker/checker, undermining its interpretability, maintainability, as well as flexibility, besides leaving it more vulnerable to human error.

A possible improvement on this solution might be to *distribute* the aggregate state *tracking* of the design independently across each of the participating events in simple (often binary) terms of (A) whether a given event *E* has been reported or not (or in some cases, how often) and (B) its relative ordering vs. other events *E'* reported (or not) across time (cycles). In a similar manner, *checking* of the tracked state could be performed against a specification of whether a given event *E* should have been reported (or not) in relation to other events *E'* previously or concurrently reported (or not) at a given cycle. On closer inspection, this has the advantage of decomposing both the tracking and specification across a set of events for a given execution.

On the other hand, both tracking-updates and checking-triggers in traditional FSM-based or scoreboard-based checkers operate under the influence of various qualifiers (related to either current state or inputs or other static/predetermined attributes of the event or the operands of the execution in question).

This begs the question of how these qualifiers could be integrated into the distributed tracking solution suggested above. There are three possible sources (and points of integration) for such qualifiers that could be used to satisfy these requirements: A) maintain both simple global state as well as a per-event state (booleans and/or counters) which could be used to qualify any tracker updates to or any checks against the specification B) use event or execution flow attributes to cleave the specification space into separate per-flow/attribute checkers while leaving the tracking mechanism unqualified and C) embed the tracking qualifiers into primitives used to express ordering relation tracking updates or correctness requirements among the set of possible events.

While considering candidate structures to implement this distributed solution in, it does not take a leap of imagination to arrive at a correspondence of such a specification to a set of graphs, where each graph could correspond to a unique combination of execution flow and/or attributes, each node of which could represent an execution event and the edges or paths between nodes represent ordering relations. Using additional state (stored for each such graph and/or its constituent nodes), we could also implement a tracker to implement updates based on event occurrences as node visits.

More specifically for formal verification, this implementation has the inherent advantage of modularity and decomposition along the axes of both precondition case-splits and sequential depth. This will be discussed in more detail in the *Contributions* section below.

Such was the chain of reasoning that led us to an execution-flow graph-based solution.

V. Description of Contributions

A. Hopscotch Methodology Overview

A summary of the *Hopscotch* methodology and framework follows:

1. First step is to enumerate various *execution flows* within the desired scope along unique sets of attributes, e.g., instruction-type, memory type, alignment etc. Each execution flow typically corresponds to the lifetime of a single transaction e.g., a specific memory access tracked by the end-to-end checker.

2. Next capture and categorize a set of key intermediate architectural/micro-architectural *execution events* (spatially or temporally separated across one or more pipelines), typically common across the set of related *execution flows*

3. Specify a set of legal (or alternately illegal) ordering requirements among the set of *execution events* that describe each *execution flow* via a set of simple temporal primitives. This specification builds a unique static **flow-multigraph,** which represents an abstraction of all legal (non-deterministic) traces for a given *execution flow*. Each node in this graph corresponds to a unique *execution event* and various hop, concurrence and path, and concurrence visit-relations among a subset of these nodes constitute edges or paths in the respective directed flow-graphs.

4. Model actual (non-deterministic) dynamic execution of each flow in a separate checkable **trace-graph** that overlays additional state on the *flow-multigraph*, updated both globally and for each node visited along the execution path.

5. Each node-visit on the *trace-graph* triggers the appropriate set of correctness and forward-progress checks determined by the underlying *flow-multigraph*, the former being retrospective and the latter being prospective in nature.

6. Forward-progress checks are implemented as both **liveness checkers** from a given node to a subset of visitable nodes for a given *execution flow,* as well as *assume-guarantee* based bounded **safety checkers** for each eligible node : *(A)* **progress-checks** guard against saturation of finite-width progress-counters, qualified by a combination of external wait dependencies and internal stalls and *(B)* independent **guarantee-checks** for internal-stall conditions (that the *progress checker*s are contingent upon) to eventually clear

7. Almost all component checkers are autogenerated by traversal of the underlying graph representation. Any counterexample traces can be logged or visualized in terms of a more intuitive sequence of node/event visits.

The more readily apparent benefit (in terms of complexity reduction and proof-depth) of the decomposition inherent in this approach is that the distributed node-based checks are invoked with significantly narrowed preconditions at progressively deeper pipeline stages. In addition, this approach is flexible, modular and enables incremental development.

B. Execution Events

An *Execution Event* is one of the following types of architectural or micro-architectural events directly or indirectly either (A) triggered on behalf of the *Tracked-Transaction* being executed (or other transactions relative to it) in the *DUT* or (B) affecting the *Tracked-Transaction*'s future execution.

Some examples include Issue, Acceptance, Speculation Updates, Associative Lookups (including memory), Buffer Allocations or Cross-Linkages, Replays, Data Reads or Writes, Resolutions, Faults and Aborts, Secondary (Miss)

Requests, Silent Escapes (e.g., a non-faulting check fail causes conversion to NOP), Ambiguation / Aliasing Events (once it is no longer possible to distinguish the future side-effects of the tracked transaction from another from a checking point-of-view)

D. Execution Flows and Flow-Attributes

Each *Execution Flow* abstracts the complete set of non-deterministic legal executions of a specific type of transaction through the DUT, expressed as a set of *Event-Ordering Rules*, each operating on a tuple of *Execution Events* as its arguments. To avoid a proliferation of distinct *Execution Flow* specifications based only on minor variations in their respective attributes, we also allow a limited set of *Flow-Attributes* to be passed as arguments to each *Execution Flow* to qualify the *Event-Ordering Rules* with (in the form of external control-flow or as arguments passed directly to the primitives).

E. Execution Flow as a Multigraph

For a given *Execution Flow* ***F*** across ***N*** events based on a set ***R*** of ***m*** *Event-Ordering Relations* for *Safety* or *Progress*

- Each *Execution Event* in ***F*** corresponds to an *Atomic Event-Node* on a set of *(Directed) Execution-Flow Graphs* $\boldsymbol{FG_{R0}}$, $\boldsymbol{FG_{R1}}$…$\boldsymbol{FG_{Rm}}$, where $\{\mathbf{R_0}, \mathbf{R_1}\ldots\mathbf{R_m}\} \in \mathbf{R}$
- Each *Directed Edge (Path)* from *Event-Node(s)* $\boldsymbol{E_a}$ to *Event-Node(s)* $\boldsymbol{E_b}$ on each *Directed Execution-Flow Graph* $\boldsymbol{FG_{Ri}}$ captures a unique ***Direct (Transitive)*** *Event-Ordering-Relation* $\boldsymbol{R_i}$ between $\boldsymbol{E_a}$ and $\boldsymbol{E_b}$
- Each *Atomic Event-Node* $\boldsymbol{E_i}$ is mapped to a *Node-Type* and a set of *Node-Attributes*
- Multiple *Atomic Event-Nodes* may be combined to form a (*Composite) Rendezvous Event-Node* of a specified *Rendezvous-Type*

Figure 3 below illustrates an example execution-flow graph.

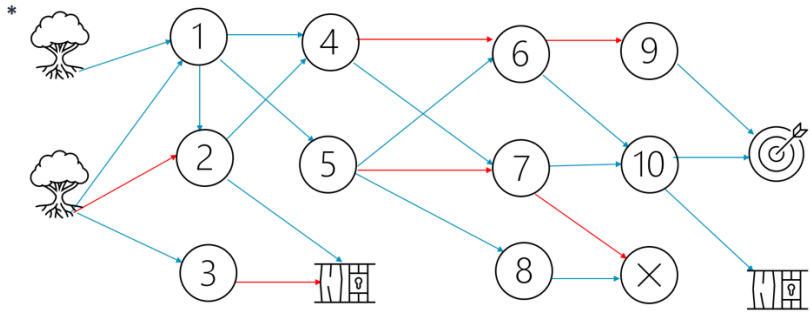

Fig. 3. Conceptual Illustration of a Flow-Graph for a Binary Ordering-Relation (Blue and Red edges indicating Legal and Illegal Relations respectively); A Tree icon represents a Root Node, a Number icon represents an Intermediate-Hop Node, a Cross icon represents a Failure Node, an Open-Bars icon represents an Escape Node, and the Dartboard icon represents an Endpoint Node.

E. Atomic Event-Node Types

Code	Name	Description
NONE	Initialized	Default
ASNC	Asynchronous	No ordering relation w.r.t. any other *Event-Node*
ROOT	Root	*Atomic Event-Node* visited first (allow multiple); no inbound edges
FORK	Boolean Test	Check Invocation (Pass/Fail)
IHOP	Intermediate Hop	Neither Root nor Leaf
FAIL	Failure	*Tracked-Transaction* aborted (Leaf)
ESCP	Escape	Cannot disambiguate *Tracked Transaction* for future hops (Leaf) due to an aliasing event
ENDP	Endpoint	*Tracked-Transaction* success; No outbound edges (Leaf)

Table 1. Node Types

F. Event-Ordering Semantics (Safety)

Event-Ordering semantics for *safety* are based on looking back at *past* or *concurrent* visits to same or different *Event-Nodes,* given a *current* visit to an *Event-Node.* These can be expressed in terms of *Concurrence-Relations, Hop-Relations and Path-Relations* (dual-polarity) below.

Primitive Concurrence Relation	Description	Formula
POS	Positive Implication	*A* |-> *B*
NEG	Negative Implication	*A* |-> *~B*

Table 2. Primitive Concurrence Relation Types between Event-Nodes A and B

Composite Concurrence Relation	Description
MUTEX	A and B never concurrent
CONDITIONAL (One-Way)	A implies B concurrent but not vice-versa
COUPLED	A and B always concurrent

Table 3. Composite Concurrence Relation Types between Event-Nodes A and B

Hop-Relation	Description
ILLEGAL	*A* can never be immediately followed by *B*
OPEN	legal if *A* visited strictly before *B*
CLOSED	legal if *A* visited before or concurrently with *B*
NONE	don't care

Table 4. Hop-Relation Types from Node-Event A to Node-Event B

Path Relation Polarity	Description
INCLUSION	*A* visited on every path to *B*
EXCLUSION	*A* never visited on a path to B

Table 5. Path-Relation Polarities from Event A to Event B

Path Relation	Description
NONE	No path requirement between *A* and *B* (Don't-care)
CLOSED	*A* is (never) either visited before (including direct hop to) or concurrently with *B*
OPEN	*A* is (never) visited before (including direct hop to) *B*
NONHOP	*A* is (never) visited strictly before any direct hop to *B*

Table 6. Path-Relation Types from Node-Event A to Node-Event B (Negative Polarity in parentheses)

Figure 4 above graphically illustrates a single Execution-Flow Graph based on a binary *Ordering- Relation* (*Legal* or *Illegal*) with the various nodes representing types from **Table 1**.

G. Composite Event-Node Types (Progress)

Rendezvous-Nodes are composite *Event-Nodes* with various "Barrier" semantics across a set of *Atomic Event-Nodes*

Rendezvous-Node	Description
*ANY**	Reached once any member node(s) visited
*ONE**	Reached once exactly one member node visited (one-hot)
ALL	Reached once all member nodes visited
NONE	Don't-Care

Table 7. Rendezvous Node Types for a set of Atomic Event-Nodes
**: may not differ in implementation (except for subsequent safety checks)*

H. Event-Ordering Semantics (Safety)

Event-Ordering semantics for progress is expressed in terms of *Progress-Relations* below. These are akin to *Hop-Relations* & *Path Relations* across *Event-Nodes*, the key difference being that they are *prospective* (forward-looking) instead of *retrospective* (backward-looking).

Progress-Relations	Description
Cross-Atomic	Define progress from a visited *Atomic Event-Node* to a specified *hop-related Atomic Event-Node* when the latter is visited
Atomic-to-Rendezvous	Define progress from a visited *Atomic Event-Node* once all *path-related Rendezvous Event-Nodes* are subsequently visited
Cross-Rendezvous	Define progress across visits to two *Rendezvous Event-Nodes*

Table 8. Progress-Relation Types across Atomic/Composite Nodes

I. Other Event-Node Attributes (Threads/Strands and Revisitability)

To simplify the specification process further, *Hopscotch* allows the user to identify subsets of *Event-Nodes* that represent execution of different parts of the overall *Tracked Transaction*. Within the member Event-Nodes of each such subset (assumed to have a stronger mutual affinity), *Event-Ordering Relations* are required to be defined in explicit terms, which in graph-theoretic terms translates to a strongly-connected directed subgraph. Conversely, this subset may be (relatively) weakly-connected with the rest of the *Execution Flow-Graph*.

Threads and *Strands* together provide a 2-level subgraph identifier that allow the *Event-Ordering Relations* between any two *Event-Nodes* in an *Execution Flow-Graph* to be initialized with a *"Don't-Care"* (or alternately, no connecting edge) if they don't belong to the same *Thread* and *Strand* pair.

This allows for a considerably more concise specification, where the user is expected to always explicitly specify (non-default) *Event-Ordering Relations* only between *Event-Nodes* belonging to each *Thread/Strand ID* pair, while still allowing for the defaults to be overridden across edges straddling these subgraphs (i.e., *Event-Nodes* belonging to different *Thread/Strand IDs*).

In addition, an *Event-Node* may also be tagged with a *Revisitability* attribute, which indicates whether or how often that *Event-Node* may be legally revisited (never, once, more than once, or infinitely often).

J. Graph Implementation

One of the simplest implementations of flow-graphs in an HDL such as Verilog or SystemVerilog is a set of enumerations (types), bit-vectors and *2-D M x N* matrices of bits or enumeration multibit-encodings. A more detailed example of an implementation will be provided in *Section VI.*

K. Making Concrete Executions Checkable Via Trace-Graphs

Once an *Execution-Flow Multigraph* specification as described above is in place, we are now faced with the problem of how to check the *DUT*'s execution of a specified *Tracked-Transaction* (with specified type, attributes etc.) against this specification.

Hopscotch achieves this by following a *Concrete Execution Trace* of the *Tracked Transaction* to build (update state in) a *Trace Graph* and at each step (cycle), sensitize the corresponding *Safety* and *Progress Checkers*. A *Trace Graph* overlays the static *Execution-Flow Multigraph* with dynamic *Trace State.*

For a set of *Execution Event-Nodes* reported in each cycle, we update the following:

- ***Global-State*** [*IDLE* | *ACTIVE* | *ESCAPED* | *ABORTED* | *DONE* | *ILLEGAL]*
- a subset of ***Node-States*** [*IDLE* | *VISITED* | *REVISITED* | *ILLEGAL]*
- ***Last (Multi-) Hop*** (*Set of Nodes Last Visited)*

This enables *Checkers* to be automatically triggered by *Event-Nodes* mapped in the *Execution-Flow Multigraph:*

- *Safety Checks* (*Retrospective*) against *Event Nodes* that may have been visited (or not) so far, optionally including this cycle.

- *Progress Checks* (*Prospective*) against *Event Node*s that may be required to be visited in the future, optionally including this cycle.

L. Graph-based Checkers

The graph-based checkers fall into three categories:

A. Trace-State Checking:

On any node-visit, check that neither the *Trace-State* nor the *Node-State* of visited *Event-Node(s)* becomes *ILLEGAL*. The next-state computation also factors in *Node-Type* and other *Node-Attributes* (e.g., *Revisit Bound*) for the visited *Event-Node(s)*, in addition to current state.

B. *Safety Checking:*

- *Hop/Path Checkers*: For all *N Event-Nodes* visited in the current cycle:
  - Check against each of the *M Event-Nodes* visited last (*Last-Hop*) that all *M* x *N Event-Node Hops* are *LEGAL*. In other words, check against specified *Hop*-Relation across each of the *M x N* pairs of *Event-Nodes*.
  - Check against all *N Event-Nodes* that with the *M Event-Node* visits, no *ILLEGAL Event-Node Paths* (*Inclusion*/*Exclusion*) are created. In other words, check against any specified *Path-Relations* (*Inclusion/Exclusion*) for all *N Event-Nodes*
    .
- *Concurrence Checkers*: For each of *M Event-Nodes* visited in the current cycle:
  - Check that no *Positive* or *Negative Concurrence-Relations* specified are being violated with respect to the remaining *M-1 Event-Nodes*.

Figures 4(a) and **4(b)** below illustrate the sequence of hop-safety and concurrence-safety checks, each triggered on nodes visited in a legal and an illegal execution trace respectively.

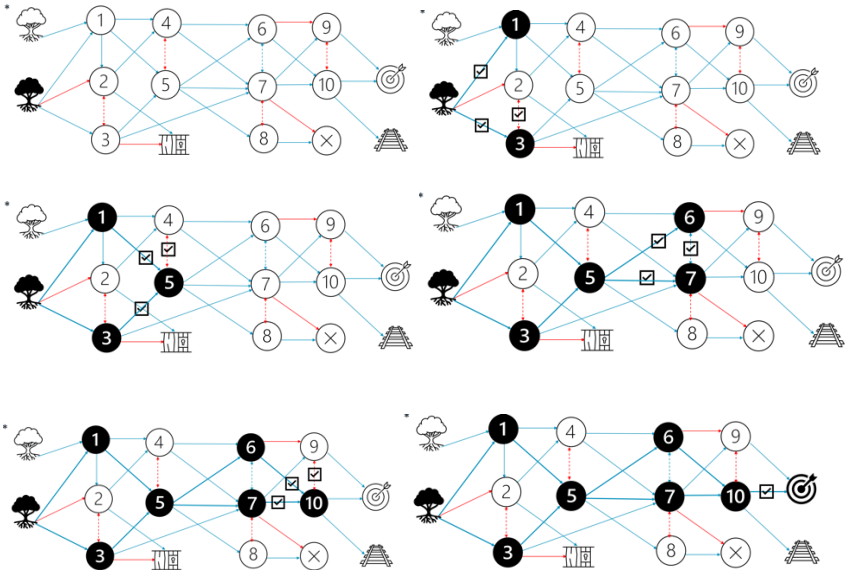

Fig. 4(a). Hop and Concurrent Safety Check Progression on a Legal Trace (from left to right, top to bottom); white nodes are unvisited and black nodes are visited; between nodes, a blue solid line represents a legal hop-relation, a red solid line represents an illegal hop-relation, a blue dashed line represents a coupled concurrence-relation and a red dashed=line represent a mutex concurrence relation

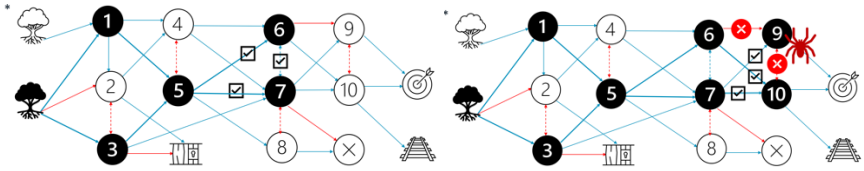

Fig. 4(b). Hop and Concurrent Safety Check Progression on an Illegal Trace (from left to right): white nodes are unvisited and black nodes are visited; between nodes, a blue solid line represents a legal hop-relation, a red solid line represents an illegal hop-relation, a blue dashed line represents a coupled concurrence-relation and a red dashed=line represent a mutex concurrence relation

Figures 5(a) and **5(b)** below illustrate the sequence of path-inclusion and path-exclusion checks, each triggered on nodes visited in a legal and an illegal execution trace respectively.

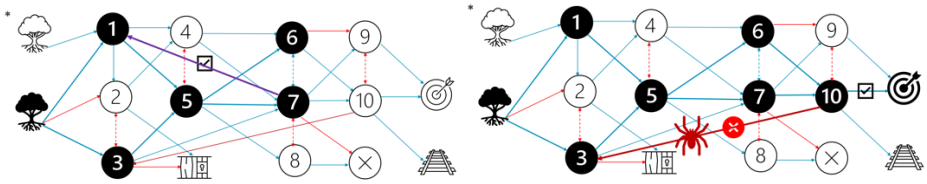

Fig. 5(a). left: Path Inclusion Check passes on a Legal Trace: Figure 5(b). right: Path Exclusion Check fails on an Illegal Trace: white nodes are unvisited and black nodes are visited; between nodes, a purple solid line represents a path-inclusion relation, while a maroon solid line represents a path-exclusion relation

C. Forward-Progress Checking

- *Progress Checkers:* Starting at a given non-leaf *Event-Node* N_i being visited, unless *escaped* or *aborted*:
  - *Liveness Variants (*require *Fairness Constraints* on *External-Wait* conditions*)* :
    - *Hop-Liveness:* will *always eventually* visit at least one node N_j to which a *LEGAL Hop-Relation* is defined from N_i
    - *Rendezvous-Liveness:* will *always eventually* visit all *Rendezvous-Nodes* (R_a, R_b…) with *LEGAL Progress-Relations* from N_i
  - *Bounded Safety Variant:*
    - Compare a predefined progress-threshold PT_i for N_i against a *Progress-Counter* for N_i, whose value is incremented (starting at the current visit to N_i) on cycles during which
      - No members of S_i, the set of *LEGALLY* allowed *Internal-Stalls* (**assume bounded**) defined for N_i are *active*, and
      - No members of W_i, the set of *LEGALLY* allowed *External-Waits* defined for N_i, are *active*, and
      - we have not *LEGALLY* visited another *hop-related Event-Node* N_j visited (clear *Progress-Counter* once visited).
- *Stall-Clear Guarantee Checkers*

- Currently coupled only with the *Bounded-Safety Variant* of the *Progress Checkers*. The soundness (or practicality) of placing *Fairness Constraints* directly on *Internal-Stall* conditions or even converting (proven) liveness checks on *Internal-Stall* conditions (using *External-Wait* fairness) into *Fairness Constraints* for *Liveness Progress-Checkers* to be used in an *assume-guarantee* paradigm is not clearly established in our understanding.

- For each internal design stall S_i ***assumed*** as ***bounded*** at a given *Event-Node*, independently check (*guarantee*) that it will either clear:

    - ***Liveness Variant***: *always eventually*, assuming fairness on all *External-Wait* dependencies, or

    - ***Bounded Safety Variant***: within a predefined stall-duration threshold ST_i, by comparing it to the counter-value of a *Stall-Clear Counter* for S_i, whose value is incremented (starting at the current stall *Si* being asserted) in the absence of *External-Wait* conditions and cleared when S_i clears.

Figures 6(a) and **6(b)** below illustrate a sequence of hop-progress checks, triggered on nodes visited in a legal execution trace.

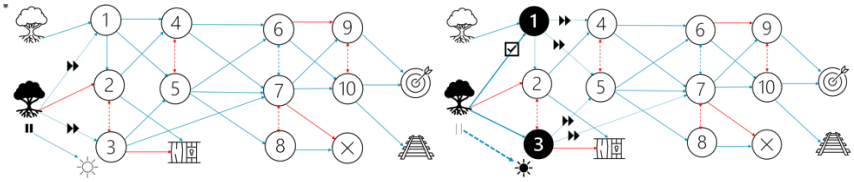

Fig. 6(a). left: Hop-Progress Check and Stall-Clear Check triggered on a visit to a Root node with an internal stall: Fig. 6(b), right: Progress-Check and Stall-Clear check passes from Root Node if a hop to one of intermediate nodes 1 or 3 is always taken; on visits to nodes 1 and 3, new Hop-Progress Checks are triggered to their respective legally-defined hops (3,5 and 4,7 respectively); white nodes are unvisited and black nodes are visited. The Pause icon indicates a Stall-Presence, and the Sun icon indicates a Stall-Absence. The Fast-Forward icon indicates a Progress Check being triggered along the corresponding edge.

Figures 7(a), 7(b) and **7(c)** below illustrate a sequence of node-to-rendezvous progress checks, triggered on nodes visited in a legal execution trace.

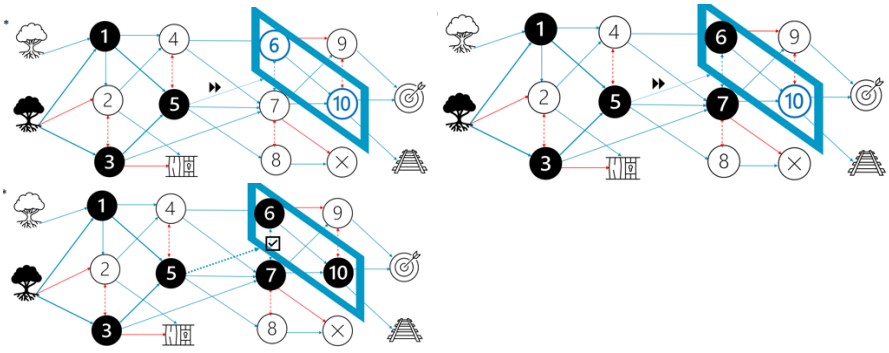

Fig. 7(a)-7(c). Progression of a Rendezvous-Progress Check to a Rendezvous Node (type ALL, composed of nodes 6 and 10) is triggered on a visit to node 5 in Fig. 7(a); this check is still active once node 6 is visited in Fig. 7(b) and is satisfied only when downstream node 10 is also visited; white nodes are unvisited and black nodes are visited in Fig. 7(c).

VI. Case Study: Verifying Store-Execution with MTE

A. Overview

We next describe a composite *Hopscotch* based *Store-Execution Checker (STEXEC)* developed to formally verify a complex RTL implementation of the *Arm v8.5 Memory Tagging Extension (MTE)* functionality on the *Load-Store* unit of a next-generation A-class CPU designed at the Arm Austin Design Center.

B. Store Execution Lifecycle Checker

This checker spans the lifetime of a *DAT/PAT* Write to the *Tracked Data Bit* within the *Tracked Cache-Line* by a *Tracked Store* (*Address* + *Data*) *µOp* from the point of acceptance (*roots*) into Load-Store (following issue) through a set of *intermediate hops*, leading to either an *abort,* an *escape,* or an *endpoint.*

The entire set of *roots*, *hops, escapes,* and *endpoints* is represented by nodes (vertices) in a static *Store-Execution Flow Graph,* its edges representing the specified set of non-deterministic legal transitions, where *(A)* the acceptance *Event-Node* (or other concurrent ones) represents *root(s)* (no inbound edges), *(B)* non-terminal *Event-Nodes* represent *intermediate-hops* (with at least one inbound edge and one outbound edge to a child Event-Node) and *(C) aborts, escapes*, and *endpoints* represent leaf *Event-Nodes* (no outbound edges)

STEXEC is decomposable along two different directions: *(A)* partitioned into distinct *Store-Execution Flow-Graphs* for various types of *Tracked Store µOp*s, *cacheability* attributes and *MTE* modes e.g., *checked Precise*-Mode Store, Checked *Imprecise*-Mode Store, *Unchecked Cacheable* Store, *STG* (Store-Tag). This can be further refined by passing minor attributes as arguments to the Flow-Graph to either support lightweight control-flow in the graph specification or tunneled as qualifier arguments to flow-graph macros and *(B)* decomposable along an arbitrary number of bidirectional paths on the *Store Execution Flow Graph* for both backward (*safety*) and forward (*progress*) checks.

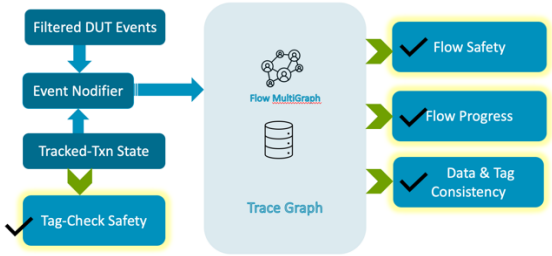

Fig. 8. Schematic of the STEXEC checker

STEXEC instantiates the following structures, as depicted in **Figure 8** above:

- The *Trace Graph* described above, which is notified of events related to the *Tracked Cache-Line* and *Tracked Store µOp* (originally reported by the *Event Monitor*).
- an intermediate *Event "Nodifier"* that maps events reported from the *Event-Monitor* to the corresponding *Event-Node(s)* in the *Store-Execution Graph*.
- a *Safety Checker* that contains/executes checker algorithm for hop, concurrence and path safety checks described above
- a *Progress Checker* that contains/executes a generalized checker algorithm for forward-progress checks described above
- a *Hop-Predicate Checker* that checks the correctness of both arguments and results of qualifications that determines the next hop for a subset of *Event-Nodes* e.g., *Tag-Checks* against other modeled state
- a *Stall-Clear Guarantee Checker* that guarantees the (bounded) fairness of legal micro-architectural dependencies at each *Event-Node* (and models fairness of external dependencies).

C. *Store Execution-Flow Multigraph: Implementation*

The *Store Execution-Flow Multigraph* is specified via a set of macros, one per value of each relation-type (hops, paths, rendezvous etc.) or progress/stall-clear counter thresholds. It is implemented as a set of SystemVerilog enumerations, 1-D and 2-D arrays:

- An enumeration of *Atomic Event-Nodes*, each mapped to the following *Node-Attributes*
  - *Type* (see **Table 1**)
  - *Thread* + *Strand*
  - *Revisitability* [*NEVER* |*ONCE* | *UNLIMITED]*
- *Hop-Relation, Concurrence-Relation, Path-Inclusion* & *Path-Exclusion* Graphs
- An enumeration of *Rendezvous Event-Nodes*, each mapped to a *Rendezvous-Type*
- *Rendezvous-Relation* Graphs (*Atomic* → *Rendezvous* & *Cross-Rendezvous*)

The utility of specifying *Event-Node* membership in *Threads* and *Strands* is to enable concise specification via support for node-affinity. We initialize all *Ordering-Relations* to *Don't-Care* across *Event-Nodes* belonging to different *Threads/Strands.*

Figure 9 below illustrates specification code for a *STREX_FAIL Event-Node* (*Store-Exclusive Failure*) with *Hop-Relation*s and *Path-Relations* defined with respect to other nodes.

```
1388 //------------------------------------------------------------------
1389 //{ Legal Path/Hops to STREX FAIL
1390 //------------------------------------------------------------------
1391
1392     if (trk_excl) begin
1393
1394       //STREX Fail can occur after RST-allocation if monitor unarmed
1395       `LEGAL_OPEN_HOP( STEXEC_NODE_EVENT_SB_RST_ALLOC_LNK_OWN, STEXEC_NODE_EVENT_FAIL_EXMONCHK)
1396
1397       //STREX Fail can also occur after TSB_MERGE (if monitor was armed at the time of RST allocation)
1398       `LEGAL_OPEN_HOP( STEXEC_NODE_EVENT_TSB_MERGE, STEXEC_NODE_EVENT_FAIL_EXMONCHK)
1399
1400       //At least OWN RST-alloc must have passed before STREX-FAIL
1401       `LEGAL_OPEN_PATH_MUST_INCLUDE( STEXEC_NODE_EVENT_SB_RST_ALLOC_LNK_OWN, STEXEC_NODE_EVENT_FAIL_EXMONCHK)
1402
1403       //Cannot write a store to L1D if fill has aborted
1404       `LEGAL_OPEN_PATH_MUST_EXCLUDE( STEXEC_NODE_EVENT_L1D_FILL_ABORT, STEXEC_NODE_EVENT_FAIL_EXMONCHK)
1405
1406     end
1407
1408 //------------------------------------------------------------------
1409 //{ Legal Path/Hops to L1D_WR (GO/Endpoint)
1410 //------------------------------------------------------------------
1411
1412     //TSB Merge allowed to occur only before L1D_WR
1413     `LEGAL_OPEN_HOP( STEXEC_NODE_EVENT_TSB_MERGE, STEXEC_NODE_EVENT_L1D_WR)
1414
```

Fig. 9. Example of an Event-Node Specification using macros for *Event-Ordering Relations*

G. *Additional MTE Checkers*

The following additional checkers were also implemented as part of the *STEXEC* suite, but are only indirectly dependent on the Hopscotch methodology, and hence not discussed here:

- *MTE Tag-Check* Correctness (triggered at multiple *Event-Nodes*)
- Data/Allocation-Tag Consistency (check against *Value-Binding* of *Tracked DAT/PAT Bit*)

VII. Results, Limitations, Challenges and future work

A. Summary of Results

1. Significantly accelerated bring-up of the STEXEC suite supporting the complex MTE feature-set in the Load-Store formal environment. It took 5 weeks from scratch till first late RTL bug was found with this checker.
2. Significant improvement in functional coverage and proof-depth achieved by graph-based checkers.
3. Found 19 unique late RTL bugs, some with multiple variants and helped rapidly qualify MTE-related bug-fixes in RTL close to release.
4. Increased debug productivity across the board based on event-based logging.
5. Capability to easily explore and understand micro-architectural flows.

B. Illustrative Design Bugs Hit with STEXEC

- **Bug A (Node-State Checker)**: Each *Event-Node* in the *Store-Execution Flow-Graph* is allowed to be visited only a specified number of times. *Bug*: RTL was writing Tracked *STGP* op's *PAT* and *DAT* at different times (writing the same *PAT* twice). This violates a subtle MTE architectural requirement.

- **Bug B (Concurrence Checker)**: If *Event-Node A* is visited, then *Event-Node B* should also get visited concurrently (in the same cycle). *Bug*: RTL was writing Tracked *STGP* op's *PAT* to cache without also writing its *DAT*.

- **Bug C (Progress Checker)**: If *Event-Node A* is visited, then at least one hop-related *Event-Node B* should eventually get visited. *Bug*: In MTE Precise Mode, RTL executes a Tag-Check for a Tracked *Store-Exclusive* op, which fails due to hitting a poisoned *PAT* in L1D cache. Since the *Exclusive-Monitor* was not armed, it correctly signals *STREX* completed (failed) but never arbitrates for and broadcasts the *FAIL* result because the *Merge Buffer* was never written to because of the *Tag-Check* error, resulting in a *hang*. A *Progress-Checker* fails here because a required hop to a downstream Rendezvous node (*STREX_RESULT)* was never taken. Progress Counter at the *STREX_FAIL* node saturated after discounting any transient stalls and external waits.

C. Impact on Scalability

This is harder to measure directly since it would require comparison against a conventional checking approach, which was never implemented. However, as a proxy, we compare times required to hit raw nodes (directly from the Event Monitor) vs. times required to hit the witness cover for the corresponding hop-safety checker (see **Figures 10(**a) and **10(b)** below). These results are based on 8-hour runs for both cover groups using an identical set of single-property formal engines.

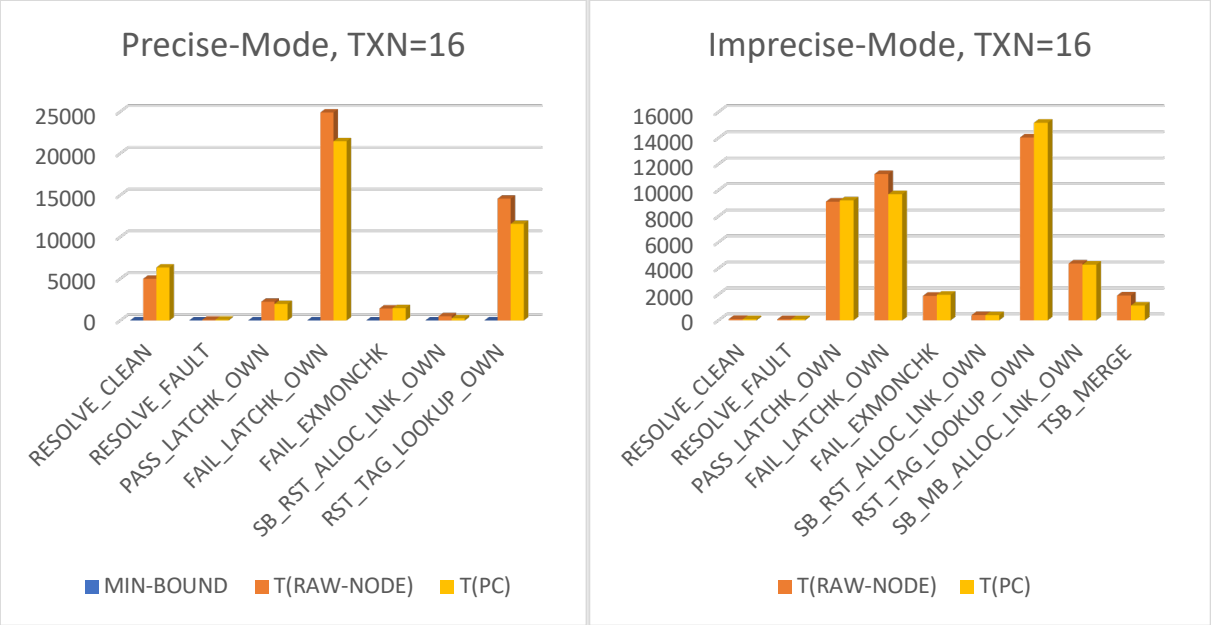

Fig. 10(a) and 10(b) Time-to-Cover (Raw-Node) vs. Time-to-Cover (Hop-Safety Checker Witness)

D. Direct Impact of Hopscotch

Flow-graph framework accelerated overall *STEXEC* bring-up. Safety checkers proved valuable for sanity testing. Automatic loop-iteration over set of nodes led to concise, low-maintenance code. It also caught lots of otherwise-subtle TB issues quickly. Debug productivity boost from visual logging of node visits and allowed speedy triage of failure traces

E. Challenges and Failures

- Identifying and reporting suitable node-events required variable level of white-boxing. The strategy was to limit to high-level events at first and next model low-level micro-architectural implementation details/decisions as required. The key tradeoff here was checker precision/decomposition vs. modeling effort involved.
- There was little payoff from liveness checkers. This likely points to a different undiagnosed source of formal complexity. Recent experiments pointed to a potential source being counters instantiated outside the COI of these properties (once cut/abstracted, started producing counterexamples). This needs to be further investigated.
- Unintentional OCs / Behavior-Removal in E2E realm (especially within abstractions) were a constant obstruction to hitting bugs. These are a silent menace and undermine confidence in formal environment, besides being challenging and expensive to both detect and debug. To address this, we have implemented a transaction attribute coverage model for tracked transactions in the E2E realm.

F. Limitations of Hopscotch

Graph implementation overhead includes that of both memory and computation. Performance of matrix implementation for flow-graphs sensitive to total number of nodes. Additional nodes increase combinational complexity as well as memory requirements and may also lower sequential complexity. This requires further experimentation and analysis.

G. *Future Work (Enhancements, Extensions and Optimizations)*

- *Coverage:* Flow-graphs are naturally extendable to generate lower-level coverage models (for both sign-off +/ DBH) e.g., pairwise hop-coverage, path coverage, per-*Event-Node* stall coverage
- More efficient graph implementations to reduce graph size/complexity. We need to investigate the feasibility of sparse matrix implementations.
- Improved (more efficient) traversal/check algorithms used in safety and progress checkers
- Richer set of more powerful *Event-Ordering* primitives. May allow for more qualified/nuanced specifications and checkers, increasing both expressivity and conciseness.

VIII. CONCLUSIONS

Investments in complexity reduction has had a huge impact on baseline formal TB performance. Focusing on the right problems with the right toolset is critical to adding value with formal. Time and effort invested in careful planning of both scope and implementation was well-spent. The *Hopscotch* framework provided a rapid, flexible, and scalable way to specify, build and test E2E checkers, allowing us to develop a mature formal environment for the LS Store-path within a couple of months and to add confidence in RTL release quality.

IX. REFERENCES

[1] Erik Seligman, Tom Schubert, M V Achuta Kiran Kumar, "Formal Verification, An Essential Toolkit for Modern VLSI Design", Chapter 10, Page 310

[2] Kostya Serebryany, "ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety", https://www.usenix.org/system/files/login/articles/login_summer19_03_serebryany.pdf, :login Summer 2019 Vol.44 No.2

[3] Armv8.5-A Memory Tagging Extension White Paper https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[4] Arm Architecture Reference Manual Armv8, for A-profile architecture https://developer.arm.com/documentation/ddi0487/latest