TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests

Naorin Hossain*
Princeton University

Caroline Trippel*
Stanford University

Margaret Martonosi
Princeton University

***Abstract*—Memory consistency models (MCMs) specify the legal ordering and visibility of shared memory accesses in a parallel program. Traditionally, instruction set architecture (ISA) MCMs assume that relevant *program-visible* memory ordering behaviors only result from shared memory interactions that take place between *user-level program instructions*. This assumption fails to account for virtual memory (VM) implementations that may result in additional shared memory interactions between user-level program instructions and both 1) system-level operations (e.g., address remappings and translation lookaside buffer invalidations initiated by system calls) and 2) hardware-level operations (e.g., hardware page table walks and dirty bit updates) during a user-level program's execution. These additional shared memory interactions can impact the observable memory ordering behaviors of user-level programs. Thus, *memory transistency models* (MTMs) have been coined as a superset of MCMs to additionally articulate VM-aware consistency rules. However, no prior work has enabled formal MTM specifications, nor methods to support their automated analysis.**

To fill the above gap, this paper presents the TransForm framework. First, TransForm features an axiomatic vocabulary for formally specifying MTMs. Second, TransForm includes a synthesis engine to support the automated generation of litmus tests enhanced with MTM features (i.e., *enhanced litmus tests*, or ELTs) when supplied with a TransForm MTM specification. As a case study, we formally define an estimated MTM for Intel x86 processors, called `x86t_elt`, that is based on observations made by an ELT-based evaluation of an Intel x86 MTM implementation from prior work and available public documentation [23, 29]. Given `x86t_elt` and a synthesis bound (on program size) as input, TransForm's synthesis engine successfully produces a complete set of ELTs (within a 9-instruction bound) including relevant *hand-curated* ELTs from prior work, plus 100 more.

***Index Terms*—memory transistency, memory consistency, enhanced litmus tests, synthesis, axiomatic modeling**

I. Introduction

Programmers and system designers rely on interface specifications to coordinate software's correct execution on hardware systems. For example, instruction set architectures (ISAs) feature *memory consistency models* (MCMs) which specify the legal orderings and visibility of shared memory accesses in any parallel program running on an implementation of the ISA. Defining behavior as fundamental as what value can be returned when software loads from memory, both under-specified and incorrectly implemented ISA MCMs have resulted in a range of bugs in real-world programs [5, 7, 18, 53], and significant effort is devoted to specifying them and verifying their correct implementation [28, 29, 33–35, 53].

*The first two authors contributed equally to this paper.

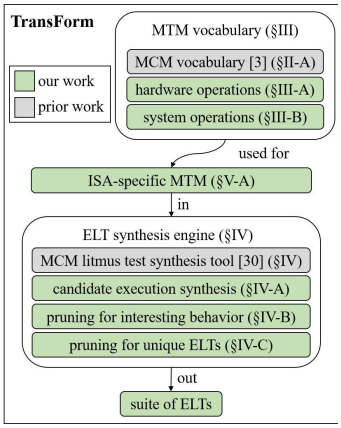


Fig. 1: TransForm features 1) an axiomatic vocabulary for specifying MTMs and 2) a synthesis engine for generating ELTs from MTM specifications.

However, this paper addresses the observation that traditional ISA MCMs fail to capture relevant shared memory interactions and therefore relevant memory ordering behaviors [29].

Transistency: MCMs assume that program-visible memory ordering behaviors only result from shared memory interactions that take place between user-level program instructions. Thus, MCMs abstract away processor virtual memory (VM) implementations that may result in additional shared memory interactions between user-level program instructions and both 1) system-level operations (e.g., address remappings and translation lookaside buffer, or TLB, invalidations initiated by system calls) and 2) hardware-level operations (e.g., hardware page table walks and dirty bit updates) during a user-level program's execution [47, 48]. Along with involving non-user-facing instructions, these additional shared memory interactions take place via shared memory state that is typically outside the purview of MCMs. This *transistency state* includes 1) page table entries (PTEs) which store virtual-to-physical address (VA-to-PA) mappings that are modifiable by the operating system (OS) and PTE status bits (e.g., dirty bits) that are modifiable directly by hardware, and 2) TLB entries which cache VA-to-PA mappings on each core and can be evicted by the OS via inter-processor interrupts (IPIs) or loaded by hardware. VM-specific systems behaviors can negatively impact concurrent program executions [4, 5, 22,

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00076

51], so ISA-level event ordering specifications like MCMs should include them. Failure to incorporate VM-aware features into ISA MCM specifications and subsequent hardware MCM verification implicitly makes the erroneous assumption that underlying VM implementations will not negatively impact program correctness.

To augment MCMs with VM-aware features, prior work proposed *memory transistency models* (MTMs): "the superset of [memory] consistency [models] which capture all [address] translation-aware sets of ordering rules" [29]. Likewise, where ISA MCM behaviors are typically specified and validated using small diagnostic programs called litmus tests (such as `sb` in Fig. 2a) [2, 3, 5, 19, 23, 30, 31], MTMs use *enhanced litmus tests* (ELTs) as a mechanism for encoding and testing the effects of VM operations on parallel program execution. ELTs are small parallel programs, comprised not just of user-facing ISA-level events (i.e., ISA instructions or micro-ops), but also of system- and hardware-level events that execute on behalf of or interleaved with user-facing instructions. Figs. 2b and 2c (explained in §II-B) give examples of ELTs, which, in contrast to Fig. 2a's standard MCM litmus test, include system- and hardware-level operations that access program-visible transistency state. Unfortunately, no prior work has formally defined MTMs, an essential step for enabling automated ELT generation and thus ELT-based validation and verification of MTM implementations. Furthermore, the ELTs of prior work were largely hand-generated[1], and therefore incomplete.

This paper presents the *TransForm framework* (short for *transistency formalized*) to support the development of formally specified MTMs, and to automate the synthesis of ELTs to validate them. TransForm (Fig. 1) consists of 1) an *axiomatic vocabulary* for formally defining arbitrary MTMs, and 2) a *synthesis engine* for automatically generating ELTs from ISA-level MTM specifications defined using the TransForm vocabulary. TransForm's axiomatic vocabulary extends beyond the standard axiomatic MCM vocabulary [3, 9, 32, 37]. It provides new constructs for modeling MTM-specific features such as particular VM-relevant shared memory state and additional MTM-relevant system- and hardware-level operations (i.e., *transistency operations*) that may interact with user-facing program instructions via this additional state.

TransForm's synthesis engine provides a mechanism to automate ELT synthesis from axiomatic MTM specifications. Together, formal MTMs and automated ELT synthesis support the specification and subsequent verification and validation of complex hardware-software event ordering scenarios. For example, a bug in AMD Athlon™ 64 and Opteron™ processors caused `INVLPG` instructions (the x86 instruction for evicting a TLB entry, which is described further in §III-B2) to fail to invalidate the designated TLB entries [4]. Such a bug, which could be detected by TransForm-synthesized ELTs, can result in the use of a stale address mapping.

[1]Prior work automates the insertion of ghost instructions (§III-A) into hand-generated ELTs based on user-defined rules.

As a case study, we define `x86t_elt`, an estimated[2] MTM for Intel x86 processors based on observations made by an ELT-based evaluation of an Intel x86 MTM implementation from prior work and available public documentation [23, 29]. We supply `x86t_elt` as input to TransForm's synthesis engine to generate an ELT suite for evaluating the model's efficacy.

We summarize our contributions as follows:

TransForm: We present the TransForm framework for formally defining MTMs and synthesizing ELTs from MTM specifications to support MTM verification and validation.

Axiomatic MTM vocabulary: To the best of our knowledge, there have been no prior efforts to formally specify MTMs. TransFrom augments MCMs with 1) transistency operations and 2) relations for articulating shared memory interactions between transistency operations and user-facing instructions.

ELT synthesis: TransForm automates ELT generation from formal MTM specifications written in TransForm's vocabulary. The synthesized ELTs can be used to automate MTM verification and validation and ultimately inform system designers about the software-visible effects of VM implementations.

x86 case study: We define an estimated MTM for Intel x86 processors, called `x86t_elt`, based on publicly-available documentation and prior ELT-based evaluation [23, 29]. Given `x86t_elt`, TransForm synthesizes all ELTs satisfying defined criteria for minimality and stressing behaviors of the model. TransForm's synthesis engine successfully produces a complete set of ELTs (within a 9-instruction bound) including all relevant *hand-curated* ELTs from prior work [29] plus 100 more. This evaluation constitutes the first automatically-synthesized and largest set of ELTs that can be used for validating Intel x86 MTM implementations.

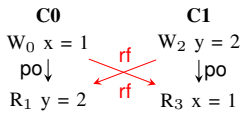
II. Background & Motivation

Since MCMs are a fundamental component for reasoning about parallel program correctness, there has been significant prior work on formally specifying them [3, 9, 11, 13, 36, 38–41, 43, 44, 55, 56]. Much of this work uses axiomatic-style (i.e. declarative) specifications, which describe the legal executions of a program with the help of logical axioms. These axioms encode the conditions that must hold true during any execution under the defined MCM. §II-A gives an overview of axiomatic ISA MCM specifications and a standard vocabulary for defining them. §II-B highlights the limitations of this vocabulary for capturing MTM-relevant program behaviors. §III extends the vocabulary in §II-A to incorporate transistency operations.

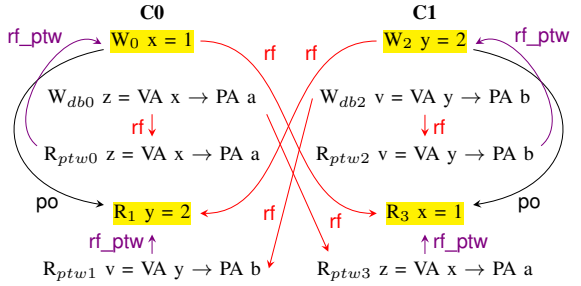
A. Axiomatic Vocabulary for Specifying Memory Models

Two primary sets, referred to here as `Event` and `Location`, can serve as the basis for defining ISA MCMs. `Event` is the set of all micro-ops (typically memory and synchronization operations) in a given program execution. `Location` is the set of all memory locations. Referring to the *store*

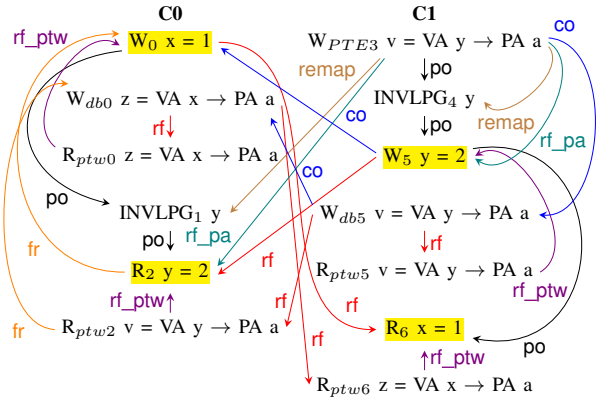
[2]"Estimated" conveys that `x86t_elt` is designed to comply with a suite of hand-generated ELTs and available public documentation [23, 29] which might be ambiguous, as pointed out in prior MCM work [15].



(a) User-level representation of the sb litmus test where the Reads on cores C0 and C1 (R_1, R_3) return the values 2 and 1, respectively.



(b) sb mapped to an ELT where the outcome remains permitted.



(c) sb mapped to an ELT where the outcome is now forbidden due to VAs x and y aliasing to the same PA a.

Fig. 2: (a) illustrates a sequentially consistent execution of the sb litmus test using traditional MCM annotations. (b) and (c) show two possible mappings of sb to ELTs using annotations representative of our new MTM vocabulary. User-facing instructions are highlighted in yellow.

buffering (sb) litmus test in Fig. 2a, $\texttt{Event} = \{\texttt{W}_0, \texttt{R}_1, \texttt{W}_2, \texttt{R}_3\}$ and $\texttt{Location} = \{\texttt{x}, \texttt{y}\}$. MemoryEvent is a subset of Event, containing only micro-ops that access memory (e.g., via reading or writing it). In Fig. 2a, all elements of Event are also elements of MemoryEvent. MemoryEvent can be further divided into Read and Write subsets that contain micro-ops that read and write memory, respectively. Each MemoryEvent element is related to exactly one Location element by the address relation. In this paper's litmus test examples, the notation <mem_op> <addr> = <data> indicates that mem_op is related to addr by the address relation. In Fig. 2a, $\texttt{address} = \{(\texttt{W}_0, \texttt{x}), (\texttt{R}_1, \texttt{y}), (\texttt{W}_2, \texttt{y}), (\texttt{R}_3, \texttt{x})\}$.

Relations can be denoted as labels (e.g., address encodes a labeling of MemoryEvents with Locations) or directed edges in program execution graphs. Directed edges indicate sequencing relationships between Events. We describe some baseline MCM "edge relations" here noting that others may be derived from this baseline set as needed. Events that are sequenced in *program order* are related by the po relation. In Fig. 2a, earlier instructions are related to subsequent same-thread instructions by po, denoted by directed po edges. Additionally, MemoryEvents that access *the same* memory location can be related by the *reads-from* (rf), *coherence-order* (co), or *from-reads* (fr) relations. rf relates Writes to Reads that they source; co relates Writes to other Writes that come later in coherence order (i.e., co is a total order on same-address Writes); and fr relates Reads to Writes that are co-successors of the Write they read from. We refer to the union of rf, co, and fr as *communication* (com) relations.

A given set of Event and Location elements along with a set of address and po relations defines a *program*. Adding com relations (which distinguish different executions of the same program) defines a *candidate execution*—i.e., a possible dynamic sequencing of program memory references and other MCM-relevant operations (e.g., synchronization operations like fences/barriers). A litmus test, as in Fig. 2a, depicts a candidate execution. In essence, com relations encode final *outcomes* of litmus test programs, where an outcome consists of the values returned by program Reads and the final state of memory. TransForm represents all stored values (and thus outcomes of candidate executions) symbolically. However, the examples in this paper feature concrete values for pedagogy.

An MCM specification defines a *consistency predicate* that renders candidate executions *consistent* or *inconsistent* with respect to the specification. For example, the total store order (TSO) MCM used by Intel x86 [23] processors, known as x86-TSO, is defined by a consistency predicate that is composed of the conjunction of three axioms (i.e., predicates that must evaluate to True): sc_per_loc, rmw_atomicity, and causality [3]. These are defined as follows:

1) sc_per_loc: The set $\{\texttt{rf} + \texttt{co} + \texttt{fr} + \texttt{po\_loc}\}$ of edges, where + indicates disjunction, is acyclic. po_loc is the subset of ^po that relates same-address MemoryEvents, where ^ is the transitive closure operator.
2) rmw_atomicity: There are no intervening same-address Writes between the Read and Write of a read-modify-write (RMW) operation. In other words, fr.co does not intersect with rmw, where relation rmw relates the Read of an RMW to its corresponding Write, and where . is the join operator.
3) causality: The set $\{\texttt{rfe} + \texttt{co} + \texttt{fr} + \texttt{ppo} + \texttt{fence}\}$ of edges is acyclic. *Preserved program order* (ppo) corresponds to a subset of ^po where the sequencing order denoted by ^po must be maintained by the architecture. fence relates Events whose ordering is explicitly architecturally-enforced by the presence of fence or barrier Events. *Reads-from external* (rfe) is the subset of rf that relates Events on different threads.

B. Limitations of Current ISA Memory Models

While MCMs of today's commercial hardware [5, 6, 8, 20, 21, 23, 40, 55] are fundamental for precisely specifying the

legal ordering and visibility of shared memory accesses in a parallel program, there are ways in which they are insufficient. Central to our work, ISA memory and synchronization operations that are fetched, decoded, and issued as part of the user-level instruction stream are *not* the only operations that may affect the outcome of a user-level program. Thus, our work encompasses typical consistency features as well as *transistency features*. In particular, our work on MTMs additionally captures shared memory interactions between user-level instructions and transistency operations (i.e., system-level and hardware-level operations). The system-level operations (i.e., *support instructions*[3]) we consider include address remappings and TLB invalidations initiated by system calls. The hardware-level operations (i.e., *ghost instructions* [29]) we consider include hardware page table walks (PT walks) and dirty bit updates. Furthermore, our work supports modeling of MTM-specific shared memory interactions by expanding the notion of "data" from MCMs beyond program variables to also include transistency-specific data (i.e., transistency state) like page table dirty bits and VA-to-PA mappings themselves.

1) Transistency Impacts Program Executions: Fig. 2 motivates augmenting MCMs with transistency features. For the MCMs of essentially all commercial processors, the litmus test execution in Fig. 2a is perfectly legal. In fact, Fig. 2a features a sequentially-consistent execution [27]. However, accounting for transistency could render the litmus test execution illegal if, for example, virtual addresses (VAs) `x` and `y` were to map to the same physical address (PA).

Figs. 2b and 2c represent two possible ways in which transistency features could affect the legality of the execution of Fig. 2a. The program executions in these figures that are enhanced with transistency features are ELTs [29]. Before explaining these ELTs, we state some assumptions made in the litmus tests we present. First, as is typical for litmus tests, memory locations in ELTs are initialized at the start of the test. Thus, a `Read` that is not involved in an `rf` relation reads from the initial program state. In keeping with MCM convention, program variables are initialized to 0 at the start of the test. Furthermore, the ELTs we present assume the following initial mappings in PTEs stored at VAs `z` and `v`: VA `z`: VA `x` $\rightarrow$ PA `a` and VA `v`: VA `y` $\rightarrow$ PA `b`, respectively. Again, all shared memory values are represented symbolically by TransForm and concretely in our examples for pedagogy. Next, `x`, `y`, and `u` are VAs and `a`, `b`, and `c` are PAs. Finally, per-core TLBs are initially empty.

Fig. 2b features one possible result of augmenting Fig. 2a's execution with `Events` related to transistency. First, each user-facing `Write`, `W`, invokes a ghost instruction, $\mathtt{W_{db}}$. Each $\mathtt{W_{db}}$ is a `Write` event that accesses a shared memory `Location` containing the dirty bit in the PTE that corresponds to the effective VA of the shared-memory `Write` that invoked it. The causal relationship between the user-facing `Writes` and their corresponding dirty bit `Writes` is denoted by matching numerical subscripts (explained further in §III-A). Next, each `MemoryEvent` invokes a PT walk ghost instruction to locate the VA-to-PA mapping corresponding to its effective VA. The mapping from Fig. 2a to Fig. 2b is an algorithmic translation that expands user-level instructions to include ghost instructions executing on their behalf. The execution it represents would be legal on essentially all commercial MCMs.

Fig. 2c shows another possible augmentation of Fig. 2a. In this case, however, the resulting ELT now represents an illegal execution on virtually all commercial processors. The illegal behavior stems from a support operation, $\mathtt{W_{PTE3}}$, on Core 1 (`C1`) which modifies the VA-to-PA mapping stored at VA `v` and results in VAs `x` and `y` aliasing the same PA `a`. $\mathtt{W_{PTE}}$ operations are `Write` events that result from address remapping system calls made on behalf of the user-level program. Each $\mathtt{W_{PTE}}$ accesses a shared memory `Location` containing the VA-to-PA mapping to be modified. The address remapping in Fig. 2c results in the ELT featuring a coherence violation (i.e., a violation of `sc_per_loc`), thus rendering it illegal under `x86-TSO` (defined in §II-A).

The ELTs in Fig. 2 illustrate that a candidate execution's legality cannot necessarily be determined solely by information provided in traditional MCM litmus tests (as in Fig. 2a). `Events` and relationships related to VM implementations must be taken into account (as in ELTs) since they can impact the correctness of interactions between user-level `Events`. A given candidate execution (or ELT) is determined to be permitted or forbidden for a given MTM by evaluating the candidate execution against the MTM's *transistency predicate*.

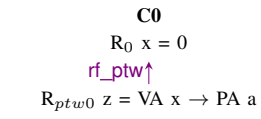
2) A Need for Formal MTM Specifications: The prior work that proposed ELTs additionally presented a transistency-aware framework, called COATCheck, for specifying and verifying microarchitectural MTM implementations [29]. COATCheck facilitates the specification of hardware designs along with their VM-relevant OS support in a way that is amenable to analysis with formal techniques. However, COATCheck conducts verification with respect to a user-provided suite of ELT programs that have been hand-curated. Improving microarchitectural MTM verification coverage to more thoroughly verify correct execution of corner-case behaviors requires a way to automatically and systematically generate relevant ELTs for a given MTM. Formal specification of an ISA's MTM is required to serve as the basis for automated ELT synthesis.

Our work gives MTMs a formal semantics. TransForm offers a language for formally specifying MTMs at the *architectural-level* whereas COATCheck demonstrated the importance of formal MTM verification at the *microarchitectural-level*. More broadly, formally specifying an ISA's MTM provides a precise interface against which tools such as COATCheck can conduct verification of hardware implementations and programs targeting those implementations, even extending to full proofs of MTM correctness in the future [33].

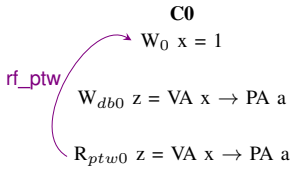
III. Towards an MTM

Starting from §II-A's baseline vocabulary for describing MCMs axiomatically, we propose additional `Events` and

[3]Prior work encompasses TransForm's support operations—address remappings and TLB invalidations initiated by system calls—in coarser-grained map-remap functions (MRFs) [48].



(a) User-facing `Reads` may also result in additional memory references in the form of a PT walk operation (R_{ptw0}).



(b) Like the `Read` in (a), user-facing `Writes` may result in a PT walk (R_{ptw0}). Additionally, `Writes` trigger a dirty bit update (W_{db0}) corresponding to the PTE of the VA-to-PA mapping.

Fig. 3: ISA `Read` and `Write` instructions invoke additional ghost instructions when executed on systems with VM. The ghost instructions access the PTE stored at address `z` to update the TLB or page table's state.

relations that are essential for defining MTMs and synthesizing ELTs. As we detail TransForm's transistency vocabulary, we reference Fig. 2 as a running example.

A. Hardware-Level Operations: Ghost Instructions

User-facing code can cause hardware to execute ghost instructions, such as hardware PT walks, on behalf of a memory access (i.e., `MemoryEvent`) [29]. Ghost instructions are not fetched, decoded, or issued as part of the program instruction stream. Rather, they are invoked on behalf of a particular user-facing instruction in the pipeline. They interact with user-facing instructions via the shared memory state that they modify, such as PTE status bits and TLB entries.

Since ghost instructions are not fetched and issued like user-facing instructions, they are not related to other `Events` on the same thread by `po`. Instead, we define the `ghost` relation to relate each user-facing instruction to the ghost instruction(s) invoked on its behalf. In the ELT examples in this paper, a `ghost` relation exists between a user-facing instruction and a ghost instruction when both have matching numerical subscripts. For example, Fig. 3a illustrates a single ghost instruction, R_{ptw0}, invoked by user-facing instruction R_0. We next describe the ghost instructions that TransForm currently supports.

1) PT walks: `MemoryEvents` operate on effective VAs; that is, they are specified to access data at a particular VA by the `address` relation (§II-A). For each memory access to a particular VA, the processor must use hardware and system support to identify the corresponding PA and physical page. If the address mapping needed by a user-facing `MemoryEvent` is not already present in the issuing core's TLB, a PT walker traverses the system's page tables to locate the mapping in a PTE and load it into a TLB entry. As with data caches, subsequent accesses to the same mapping can access this mapping from the TLB until it is evicted. Thus, a PT walk is not required for every memory access, but only those that experience TLB misses. Many systems implement hardware PT walkers for performance, which we assume here. (Our grammar is applicable for software PT walks as well.)

Fig. 3 illustrates how TransForm models PT walks and their effects on program behavior. In both subfigures, R_{ptw0} is a PT walk that loads the address mapping for VA `x` stored at VA `z`. PT walks (e.g., R_{ptw0}) must populate a TLB entry before user-facing instructions (e.g., R_0 in Fig. 3a and W_0 in Fig. 3b) can use it. The `rf_ptw` relation is introduced to model this new type of `rf` relationship; it relates a PT walk that loads a mapping into a TLB entry, to all user-facing `MemoryEvents` that "read from" that specific TLB entry. `rf_ptw` differs from `rf` in that the `Location` accessed by the PT walk is an address mapping, whereas the `Location` accessed by the user-facing instruction is a data location.

Each PT walk can only be related to one user-facing instruction with `ghost` (the one that triggered it), but it can be related to several user-facing instructions with `rf_ptw` (those that use the TLB entry it created). `MemoryEvents` that are (resp. are not) related to a PT walk operation with `ghost` represent TLB misses (resp. hits). As discussed further in §III-B2, the eviction of an address mapping from a TLB will result in a TLB miss for that address mapping in a subsequent memory access. Thus, a `MemoryEvent` that experiences a TLB miss must invoke a PT walk to re-load the required mapping back into the TLB. Referring back to Fig. 2a, each `MemoryEvent` accesses a distinct VA and thus should invoke its own PT walk. The `sb` ELTs in Figs. 2b and 2c feature these PT walks and their relationship via `rf_ptw` to the user-facing `MemoryEvents` that invoke and "read from" them.

2) Dirty Bit Updates: When an instruction writes to a memory address, the written data is typically propagated to the cache before it is written back to a physical page in memory. Thus, each PTE contains a dirty bit to indicate when the physical page corresponding to that PTE has been modified and therefore needs to be updated [23].

As with PT walks, `ghost` associates a dirty bit update with the user-facing `Write` that caused it. Fig. 3b shows this relation where W_{db0} is a dirty bit `Write` that accesses the dirty bit in the PTE stored at VA `z`. The user-facing `Write` that caused W_{db0}, namely W_0, shares the same numerical subscript. Likewise, Figs. 2b and 2c both include dirty bit `Writes` related by `ghost` to the highlighted user-facing `Writes`. Dirty bit updates are typically performed as RMW operations [23]. However, TransForm models dirty bit updates as `Write` operations. This is conservative in terms of event ordering, and it reduces the number of instructions TransForm requires to synthesize programs with `Writes` from three (user-facing `Write`, dirty bit `Read`, dirty bit `Write`) to two (user-facing `Write`, dirty bit `Write`). Furthermore, TransForm does not explicitly model the OS updating of dirty bits for synonyms (i.e., VAs that map the same PA as in Fig. 2c). We assume the OS checks all synonym dirty bits before evicting (i.e., swapping out) pages. This assumption is common in non-naive OSs including Linux and could be relaxed in future

implementations of TransForm.

B. System-Level Operations: Support Instructions

Support instructions help coordinate software's correct execution on hardware systems implementing VM. In particular, an OS has the ability to modify VA-to-PA mappings or invalidate TLB entries during a user-facing program's execution. These `Events` impact the execution of user-facing program instructions by, for example, influencing which PA is ultimately accessed by a user-facing `MemoryEvent`.

User-level MCM relations are traditionally defined assuming that each memory location accessed in a program is either a unique PA or a unique VA with no synonyms. Furthermore, user-level MCM relations do not support VA-to-PA mapping changes during a program's execution. Thus, there is no existing MCM vocabulary for articulating which PA is being accessed by a particular VA and by extension, no vocabulary for modeling synonyms that can arise from system-level `Events`. TransForm's MTM vocabulary solves both of these issues. First, TransForm enables expressing the OS's ability to alter VA-to-PA mappings via system calls by introducing support instructions for writing to PTEs as a type of `Write` event. TransForm thus supports `com` edges (§II-A) that relate `MemoryEvents` with different effective VAs as long as these VAs map to the same PA (i.e., `com` edges relate same-PA `MemoryEvents`). Second, TransForm accounts for the OS's ability to alter TLB state by introducing a support instruction for evicting specified address mappings from the TLB (`INVLPG`), either as a result of a system call changing the address mapping or spuriously.

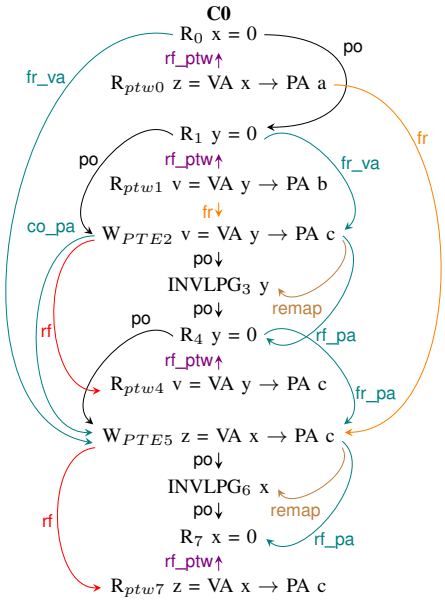
1) VA-to-PA Remappings: MTMs support the potential modification of VA-to-PA mappings (stored in PTEs) during a program's execution as a result of system calls. The implication of supporting address remappings is that we cannot assume (as MCMs do) that a specific VA is mapped to a specific PA throughout the entirety of a program's execution. Thus, TransForm provides new types of communication relations to support reasoning about which PA is accessed by a given user-facing `MemoryEvent`.

To support modeling and reasoning about the effects of VA-to-PA remappings of user-level program behavior (i.e., to deduce which interactions may take place between `MemoryEvents` with different effective VAs, yet potentially the same effective PA), we adapt the MCM `com` relations from §II-A. TransForm's adaptation of `com` edges results in four new relations that are described as follows. As with program data, TransForm represents PAs (and VAs) symbolically.

- `rf_pa`: Relates a PTE `Write` of VA `v` → PA `p` to `MemoryEvents` that access PA `p` via VA `v`.
- `co_pa`: Relates PTE `Writes` of VA `v` → PA `p` and VA `v'` → PA `p` in a total order. In other words, `co_pa` is a total order on the creation of aliases to a particular PA `p`.
- `fr_pa`: Relates a `MemoryEvent` that accesses PA `p` via VA `v` to the `co_pa`-successors of the PTE `Write` that it "reads from" in `rf_pa`.

C0
R_0 x = 0
R_{ptw0} z = VA x → PA a
R_1 y = 0
R_{ptw1} v = VA y → PA b
W_{PTE2} v = VA y → PA c
$INVLPG_3$ y
R_4 y = 0
R_{ptw4} v = VA y → PA c
W_{PTE5} z = VA x → PA c
$INVLPG_6$ x
R_7 x = 0
R_{ptw7} z = VA x → PA c

(a) ELT with two PTE `Writes` and several `Reads` that read from various mappings, depending on their location in the program. Here and throughout the paper, white cells represent user- or system-level instructions while gray lines represent ghost instructions.



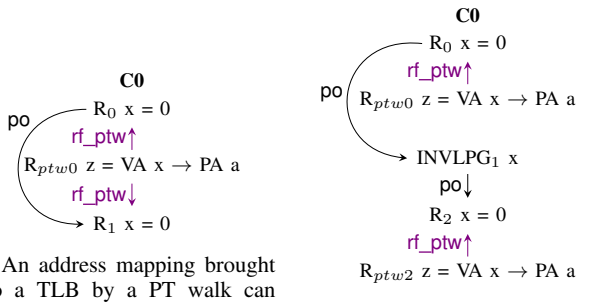
(b) ELT execution corresponding to (a), illustrating address mapping changes and resulting `_pa` edges.

Fig. 4: Example usage of each of the new `_pa` edges (§III-B1). VAs `x` and `y` are accessed before and after their mappings are changed to alias to the same PA.

- `fr_va`: Relates a `MemoryEvent` that accesses PA `p` via VA `v` to the `co`-successors of the PTE `Write` that it "reads from" in `rf_pa`.

The relations above are used by TransForm to derive `com` edges that relate `MemoryEvents` accessing the same PA but different VAs (§II-A). Notably, `rf_va` and `co_va` are not included in the above list. This is because `rf_va` and `co_va` are already captured by `rf_pa` and `co`, respectively.

Fig. 4a shows a program in which several user-facing `Reads` access either VA `x` or `y`. Two PTE `Write` instructions, W_{PTE2} and W_{PTE5}, invoked by system calls remap both `x` and `y` to a new PA `c`. W_{PTE2} and W_{PTE5} then invoke invalidations $INVLPG_3$ and $INVLPG_6$ (respectively) of the TLB entries corresponding to their remapped VAs to prevent stale mapping accesses.



(a) An address mapping brought into a TLB by a PT walk can source many same-thread instructions.

(b) A new PT walk is needed when INVLPG evicts a TLB entry.

Fig. 5: An INVLPG inserted between Reads accessing the same VA enforces reloading of the TLB entry via a PT walk.

Fig. 4b illustrates how these remapping operations relate to each other and to the user-facing program instructions using the _pa edges described above.

System-level PTE Writes via system calls can fundamentally change legality of particular MCM litmus test outcomes, as Fig. 2c shows. Here, W_{PTE3} changes the mapping of VA y to PA a so that VAs x and y map to the same PA. W_{PTE3} is related to R_2 and W_5 via rf_pa, indicating that R_2 and W_5 read from W_{PTE3}'s new address mapping and thus access the same PA as W_0 and R_6. As a result, this particular candidate execution features an illegal coherence violation, as described in §II-B.

2) TLB Entry Evictions: §III-A1 discussed how TLB state can be modified via PT Walks. Here, we discuss how TransForm handles TLB state modifications that result from TLB evictions. TLB entries can be evicted for several reasons such as 1) a change in the corresponding PTE, 2) a spurious eviction by the OS, or 3) a TLB capacity eviction. We address each of these scenarios in the following paragraphs.

First, on multicore systems, when address mappings are modified, they must be invalidated in the TLBs of *all* cores caching this mapping, not just the core performing the mapping change. The mechanism used to invoke these page invalidations on each core varies by architecture [12]. TransForm models this TLB entry invalidation using an IPI in the form of an INVLPG instruction [23]—named for an instruction in the x86 ISA, but similar operations exist in other ISAs as well—related to a PTE Write via a remap relation. remap relates a PTE Write to corresponding INVLPGs on *each* core that invalidate the TLB entries rendered stale by the PTE Write (as in Fig. 4). For example, in Fig. 2c, W_{PTE3} (on C1) invokes INVLPGs, $INVLPG_1$ and $INVLPG_4$, on C0 and C1, respectively (as denoted by the remap relation), to invalidate the appropriate TLB entries due to the address mapping change. All memory accesses to a VA affected by an INVLPG must read from the latest address mapping. In Fig. 2c, R_2 and W_5 read from the new mapping of VA y and access PA a. Currently, INVLPG is the only type of IPI modeled by TransForm. However, support for additional IPIs is possible in future TransForm extensions.

Second, the OS can initiate TLB evictions even when a PTE has not been modified by a system call (e.g., by spuriously invoking INVLPG instructions). When an INVLPG is invoked by the OS and the corresponding PTE has not changed, TransFrom does not instantiate a remap edge. MemoryEvents following these spurious INVLPGs can read from the unchanged PTE mapping but must bring the mapping back into the TLB with a PT walk. Fig. 5a illustrates two Reads, R_0 and R_1, to VA x that use the mapping brought into the TLB by the same PT walk, R_{ptw0}. In Fig. 5b, there is an intervening page invalidation, $INVLPG_1$, between the two Reads to VA x so the second Read, R_2, invokes a new PT walk, R_{ptw2}, to bring the previously evicted mapping back into the TLB. When synthesizing ELT candidate executions with TransForm's synthesis engine, spurious INVLPGs are only inserted on threads if they can affect the thread's execution.

Finally, TLB capacity evictions occur when a TLB entry must be evicted to make room for a new entry. (This could occur due to capacity or conflict effects.) TransForm models these evictions with the invocation of a PT walk by a user-facing MemoryEvent. As explained in §III-A1, the loading of a TLB entry by a PT walk indicates that there was a TLB miss. A TLB miss occurs when 1) the address mapping is first being used, 2) the entry is evicted by the OS (i.e., using INVLPG), or 3) there is a TLB capacity eviction. Thus, when ELTs feature PT walks that are not accessing an address mapping for the first time (i.e., prior PT walks have been issued for this mapping) and simultaneously do not feature OS evictions of the address mapping from the TLB (i.e., INVLPG has not been called for this address mapping), then the PT walk is caused by a TLB capacity eviction. In TransForm's automatic synthesis, it explores all three of these possibilities for TLB events.

C. Simplifying Assumptions in ELTs

Compared to MCM litmus tests, the presence of additional operations, relevant state, and shared memory interactions in ELTs mean that ELTs can become significantly larger (by instruction count) and more complex (by the number of potential interactions between ELT operations). Therefore, the ELTs considered in this paper feature some simplifying assumptions enumerated below. These assumptions do not sacrifice the generality of our approach, but they do result in improved performance when ELTs are analyzed or synthesized using TransForm (e.g., in §VI).

1) Each thread of an ELT is assumed to execute on a distinct processor core. Therefore, each thread has access to private storage, including a private TLB. While TransForm can support hyperthreading, the ELTs we consider in this paper represent individual multi-threaded processes as is common practice in MCM analysis.
2) Prior to the execution of an ELT, it is assumed that each VA maps to a unique PA. Without this assumption, a PTE Write, along with corresponding INVLPGs, would need to be explicitly modeled and included in the ELT for *each* VA accessed in the program to appropriately derive which corresponding PAs are accessed. These additional

MTM elements	Descriptions
Event	instruction representing a micro-op in a program
MemoryEvent	Read or Write memory access in a program
address	relates MemoryEvent to Location being accessed
po	program order, same-thread sequencing of Events
rf	relates Write to Reads it sources
co	relates Write to other Writes in coherence order
fr	relates Read to co-successors of Write it reads from
ghost	relates user-facing MemoryEvent to induced ghost instructions
rf_ptw	relates PT walk to user-facing MemoryEvents that read from loaded TLB entry
rf_pa	relates PTE Write to user-facing MemoryEvents that access written address mapping
co_pa	relates PTE Write to other subsequent PTE Writes for same PA in coherence order
fr_pa	relates user-facing MemoryEvent to co_pa-successors of PTE Write they read address mapping from
fr_va	relates user-facing MemoryEvent to subsequent PTE Write that changes address mapping for accessed VA
remap	relates PTE Write to invoked INVLPGs

TABLE I: Summary of MTM vocabulary. The elements in gray are baseline MCM elements that TransForm builds on.

instructions would unnecessarily degrade performance of ELT synthesis with TransForm.

3) We do not model recursive page tables. Stemming from this design choice, we do not model ghost instructions for PTE Writes. In reality, PTE Writes modify some VA whose mapping to a PA is stored in *another* PTE. The process for finding address translations in these higher levels of page tables is very similar to finding them for the base case that we model.

D. Illustrative Example of Vocabulary Usage

Table I summarizes TransForm's MTM vocabulary. This section illustrates how TransForm's MTM vocabulary can be used to reason about observable program behaviors in a more complex ELT example than the running example of Fig. 2.

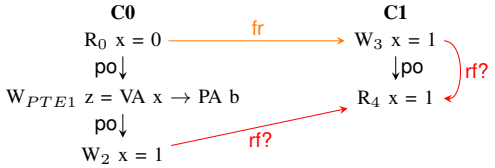
Consider the ELT in Fig. 6. Fig. 6a features an MCM litmus test consisting of four user-facing instructions and one support instruction (induced by a system call), W_{PTE1}. As Fig. 6b shows, using only MCM relations results in ambiguity as to which Write, W_2 or W_3, R_4 is reading from. Although reading from either Write constitutes a valid program execution for x86-TSO, the ambiguity can be cleared up with MTM relations.

Fig. 6c extends Fig. 6a to one possible ELT formulation (illustrated with MTM relations in Fig. 6d) as follows. First, Fig. 6b is augmented with system-level INVLPGs that are invoked via OS-issued IPIs on each core. Fig. 6d shows $INVLPG_2$ and $INVLPG_5$ related to W_{PTE1} via remap relations. Second, ghost instructions are added for each user-facing MemoryEvent (R_0, W_3, W_4, and R_6). Since $INVLPG_2$ and $INVLPG_5$ evict the TLB entries loaded by R_{ptw0} and R_{ptw4}, R_{ptw3} and R_{ptw6} bring updated address mappings for VA x into the TLB that are accessed by W_3 and R_6, respectively. Third, the inserted ghost instructions result in the addition of the appropriate _pa and _va relations (§III-B1). Since W_3 and R_6 access the mapping for VA x written by W_{PTE1}, there are rf_pa edges relating each to W_{PTE1}. Similarly, R_0 and W_4 read from the initial address mapping for VA x so there are fr_va edges relating them to W_{PTE1}. Fourth (and finally), based on the _pa and _va relations, the effective PAs that are accessed by each MemoryEvent can be derived and appropriate com edges can be added to the ELT. It is now clear that R_6 reads from W_3 in this particular execution (due to the rf relation between them) while W_4 accesses a different PA.

As this section's examples show, Table I's proposed MTM vocabulary facilitates modeling of transistency Events that encompass consistency issues well beyond traditional MCMs. We next use this vocabulary to automate ELT synthesis.

C0	C1
R_0 x = 0	W_3 x = 1
W_{PTE1} z = VA x → PA b	R_4 x = 1
W_2 x = 1	

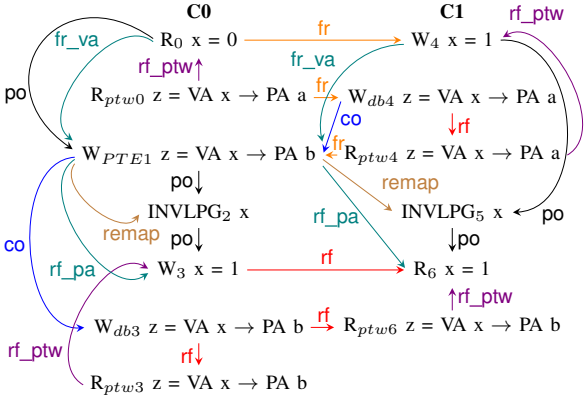
(a) A user-level test in which a Read (R_0) reads 0 at VA x and the address mapping for x is changed. R_4 reads a 1 at address x.



(b) Mapping of (a) showing an ambiguous data access by R_4.

C0	C1
R_0 x = 0	W_4 x = 1
R_{ptw0} z = VA x → PA a	W_{db4} z = VA x → PA a
W_{PTE1} z = VA x → PA b	R_{ptw4} z = VA x → PA a
$INVLPG_2$ x	$INVLPG_5$ x
W_3 x = 1	R_6 x = 1
W_{db3} z = VA x → PA b	R_{ptw6} z = VA x → PA b
R_{ptw3} z = VA x → PA b	

(c) A possible ELT for the program in (a).



(d) ELT illustrating the program in (c).

Fig. 6: The program in (a) results in an ambiguous MCM relation mapping in (b). The ELT in (c) and (d) presents a transistency view of a possible candidate execution.

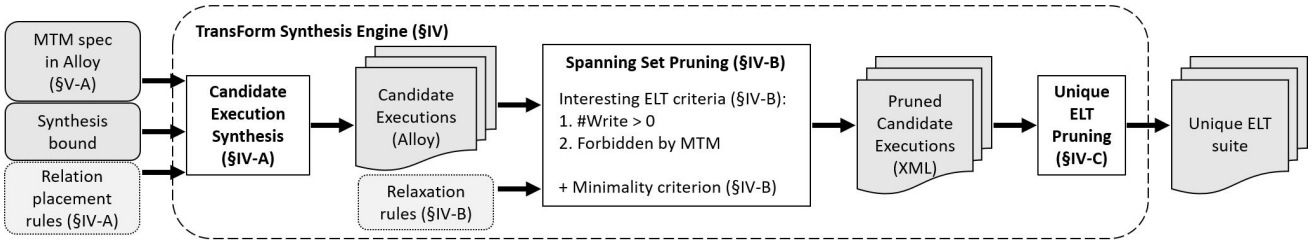


Fig. 7: TransForm's synthesis engine uses MTM vocabulary from Table I to axiomatically define the inputted MTM and synthesize candidate executions in Alloy which are pruned and deduplicated to find unique, interesting ELT programs. Relation placement and relaxation rules are implicit inputs that are defined to apply broadly across systems.

IV. Automating ELT Synthesis

TransForm features a synthesis engine for automatically generating a suite of ELTs from a formal, axiomatic MTM specification supplied using Table I's vocabulary. As shown in Fig. 7, TransForm's synthesis engine performs bounded ELT synthesis in conceptually three main steps elaborated in the subsections below. First, TransForm synthesizes the set of all possible ELT executions up to a user-specified instruction bound. Second, this set of candidate executions is pruned based on which executions feature *interesting* transistency behaviors. Finally, the subset of interesting candidate executions are deduplicated to output a suite of unique (and interesting) ELT programs.

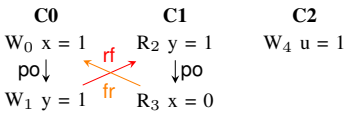
A. Candidate Execution Synthesis

Synthesizing candidate ELT executions based on our formal MTM vocabulary requires axioms (i.e., rules defined in terms of our vocabulary) that articulate what a valid ELT *looks like*. Synthesizing traditional MCM litmus tests requires relatively few axioms to describe a legal program execution. For example, consider the MCM features that uniquely define a MCM candidate execution: `Event`, `address`, `po`, `rf`, `co`, and `fr`. One axiom might state that `po` must be acyclic in any valid program execution. Another might state that `co` must represent a total order. To summarize, MCM litmus tests have virtually no constraints on which `Locations` can be related to which `MemoryEvents` via the `address` relation, where individual `Events` can be placed within a program thread, and which (same-address) instructions can interact via `com` relations. As shown in Fig. 7, TransForm performs synthesis based on the axioms provided to specify the MTM, as well as a set of placement rules that guide synthesis regarding how operations, such as ghost instructions, can be placed and inserted.

With TransForm's augmented MTM vocabulary (Table I), synthesizing candidate ELTs requires a more complex set of axioms to describe a legal program execution. For example, `com` edges must relate `MemoryEvents` accessing the *same PA*. PTE `Writes` must induce `INVLPGs` on each core. Likewise, when a program features an `INVLPG`, a `MemoryEvent` following that `INVLPG` in $^{\wedge}$`po` that accesses the address mapping evicted from the local TLB by the `INVLPG` must reload the mapping back into the TLB with a PT walk. Moreover, as described in §III-A, ghost instructions and their corresponding relations (`ghost` and `rf_ptw`) have very specific rules that dictate their legal behavior in a candidate execution.

C0	C1	C2
W_0 x = 1	R_2 y = 1	W_4 u = 1
W_1 y = 1	R_3 x = 0	

(a) A litmus test with `Writes` on `C0` and `Reads` on `C1` to addresses `x` and `y`. There is also a `Write` on `C2` to address `u`.



(b) Mapping of the program from (a). There is a cycle formed by the `po`, `rf`, and `fr` edges on `C0` and `C1`.

Fig. 8: This candidate execution violates `x86-TSO` axioms described in §II-A, and therefore would be included in the vector space of interesting ELTs. However, it does not satisfy the minimality criterion; removing the `Write` (W_4) would not make this program satisfiable under `x86-TSO`, even though removing any of the remaining `Events` would. Because it is not minimal, TransForm would not synthesize this ELT.

Given rules for describing legal ELT formulations (as discussed in Section III), TransForm can synthesize all conceivable ELTs up to a user-specified bound on the number of program instructions. The following sections describe how this set of tests is pruned down to a minimal and interesting subset.

B. Spanning Set Pruning

TransForm's synthesis engine defines and generates a *spanning set* of ELTs. In linear algebra, every vector in a vector space `V` can be written as a linear combination of the vectors in the spanning set `S`. In our work, TransForm synthesizes a spanning set `S` of ELTs where the space of all relevant MTM behaviors (that are realizable up to a user-provided instruction bound) can be captured by the ELTs in `S`.

TransForm requires the following criteria for inclusion of ELTs in the vector space of relevant (i.e., interesting) MTM behaviors. First, an ELT must contain at least one `Write`. This requirement enables multiple possible outcomes (i.e., executions) for the ELT. Second, an ELT must be able to produce an outcome that can violate the transistency predicate

of the user-provided MTM. This rule ensures that synthesized ELTs have the potential to expose forbidden MTM behaviors when used for verification and validation.

After pruning the set of all legal candidate ELTs to produce only those that belong in our vector space of interesting MTM behaviors, ELTs are evaluated for inclusion in our spanning set based on a *minimality criterion*. Minimality requires an ELT execution to have a forbidden outcome that becomes legal (according to the transistency predicate) under every possible isolated *relaxation* of the ELT program [30]. For TransForm's synthesis engine, Fig. 7 depicts relaxation rules as an implicit input to the synthesis pruning stage. A relaxation corresponds to the removal of an `Event` (or group of `Events` as described below) or dependency[4] in the ELT. Relaxations are applied to each candidate ELT in the vector space to determine whether it satisfies the minimality criterion. Fig. 8 shows a simplified example of a candidate ELT with only user-facing `Events` presented that *would not* meet the minimality criterion.

The most common relaxation performed by TransForm's synthesis engine when evaluating minimality is the removal of an `Event`. Conceptually, this relaxation is intended to remove just a *single isolated event*. However, the removal of some `Events` from an ELT may render the ELT invalid. For example, ghost instructions are not permitted to exist in an ELT if they do not correspond to some user-facing `MemoryEvent` that invokes them. Alternately, some user-facing `MemoryEvents` *require* the invocation of particular ghost instructions. Due to these requirements, when performing a relaxation intended to remove a single `Event` TransForm removes additional `Events` to maintain legality of the ELT. For example, TransForm permits the removal of a ghost instruction if and only if its corresponding user-facing `MemoryEvent` is itself removed. Likewise, `INVLPGs` that are invoked by a system-level PTE `Write` can only be removed if and only if the PTE `Write` itself is also removed. Spurious `INVLPGs` that are *not* a result of PTE changes, however, are free to be removed in isolation. Unlike the restricted relaxations in this work, user-level MCM litmus test synthesis from prior work permitted relaxations that remove *any* arbitrary `Event` from a litmus test in isolation [30].

C. Alloy Implementation and Unique ELT Pruning

We use the Alloy relational modeling domain-specific language (DSL), specifically Alloy 4.2 [25], to encode axiomatic MTMs written in TransForm's vocabulary and to implement the synthesis engine described in this section. Alloy's relational model-finding backend, Kodkod [52], enables us to transform the ELT synthesis problem into a SAT problem to be fed to any off-the-shelf SAT solver; our experiments use the MiniSat SAT solver [17]. Once TransForm's synthesis engine determines which ELTs are eligible for inclusion in the spanning set, Alloy outputs them in XML form. XML ELTs are post-processed using a deduplication engine built on prior work to return a set of unique ELT *programs* [30]. Our experiments synthesize ELTs via the process in Fig. 7 up to our provided instruction count bound.

[4]In our evaluation we only consider `rmw` dependencies, which are modeled as relations that relate the `Read` and `Write` of an RMW operation.

V. Case Study: x86 MTM

This section uses Table I's axiomatic vocabulary to define and develop an MTM, `x86t_elt`, that estimates the MTM of Intel x86 processors, based on a range of public information and analysis from prior work [23, 29]. Then, TransForm's synthesis engine automatically generates the suite of ELTs that encode the spanning set of `x86t_elt`'s MTM behaviors.

A. Defining x86t_elt

As with the consistency predicate for `x86-TSO`, the transistency predicate for `x86t_elt` consists of the conjunction of several axioms. Since transistency is a superset of consistency, the axioms that comprise the `x86t_elt` transistency predicate include, as a subset, the axioms that comprise the `x86-TSO` consistency predicate (§II-A) [3]. We identify and evaluate two additional `x86t_elt` transistency axioms, listed below. The first axiom (`invlpg`) is required for capturing software-visible effects of x86 transistency implementations, while the second (`tlb_causality`) is a "diagnostic" axiom to aid hardware designers in localizing transistency bugs caused by incorrect TLB implementations.

1) `invlpg`: The set $\{\texttt{fr\_va} + {}^\wedge\texttt{po} + \texttt{remap}\}$ of edges must be acyclic.
2) `tlb_causality`: The set $\{\texttt{ptw\_source} + \texttt{com}\}$ of edges must be acyclic.

The remainder of this section describes the derivation of these MTM axioms. From analysis of public x86 documentation and prior work [23, 29], we identify forbidden MTM behaviors and use TransForm's vocabulary to define axioms that prevent them.

1) `invlpg`*:* `invlpg` enforces that a `MemoryEvent e` must read from the latest VA-to-PA mapping associated with its effective VA when it follows an `INVLPG i` in $^\wedge$`po` and both `e` and `i` access the same VA. `MemoryEvents` can only access PA `p` via VA `v` as long as this address mapping remains intact in their local TLB. If a system call remaps VA `v` to some PA `p`′ with a PTE `Write` and invokes `INVLPGs` (represented with `remap` relations) on each core, the previous mapping of VA `v` to PA `p` is rendered invalid for `MemoryEvents` following the `INVLPGs` in $^\wedge$`po`. The relation `fr_va` is an architecturally-enforced ordering that relates a user-facing `MemoryEvent` to PTE `Writes` that remap its effective VA to a new PA. `remap` represents an architecturally-enforced ordering between a PTE `Write` and the `INVLPGs` it invokes. Furthermore, it is enforced architecturally that a `MemoryEvent` that accesses a TLB entry that was evicted by an `INVLPG` occurring earlier in $^\wedge$`po` cannot access an "old" address mapping. More specifically, `x86t_elt` enforces that some `MemoryEvent e` following some `INVLPG i` in $^\wedge$`po` (where both access the same VA) must access a VA-to-PA mapping that is a `co`-successor of the mapping invalidated by `i`. Thus, we require acyclicity of the union of `fr_va`, `remap`, and $^\wedge$`po`.

2) `tlb_causality`*:* `tlb_causality` prevents a causal relationship between some `MemoryEvent` e′ and some other `MemoryEvent` `e` whose corresponding PT walk sourced the TLB entry accessed by e′. Since `MemoryEvents` locate VA-to-PA mappings in the TLB of their local core, `rf_ptw` represents an architecturally-enforced ordering between a `MemoryEvent` and the PT walk that populates the TLB entry it accesses. Furthermore, our `x86t_elt` model assumes an architecturally-enforced ordering between the user-facing `MemoryEvent` that invokes (i.e., is related by `ghost` to) a PT walk and other user-facing `MemoryEvents` (on the same core) that access the TLB entry populated by this PT walk. To represent this ordering relationship, we derive `ptw_source` to relate a user-facing `MemoryEvent` that invokes a PT walk to all other user-facing `MemoryEvents` that are related to this PT walk by `rf_ptw`. Thus, some `MemoryEvent` e′ that is ordered after some other `MemoryEvent` `e` in `ptw_source` cannot be related to `MemoryEvent` `e` by a causal communication relationship.

As noted above, we include `tlb_causality` in `x86t_elt` for the purpose of diagnosing hardware bugs in TLB implementations. In particular, the architecturally-visible effects of `tlb_causality` violations are already subsumed by violations of another `x86t_elt` axiom, specifically `causality` (hence the naming convention). However, including `tlb_causality` enables TransForm to specifically identify which ELTs may be used by hardware verification engineers to localize transistency bugs to incorrectly implemented TLBs. Of the 103 unique ELTs that TransForm synthesizes for `x86t_elt` (§VI), five can be attributed to violations of `tlb_causality`.

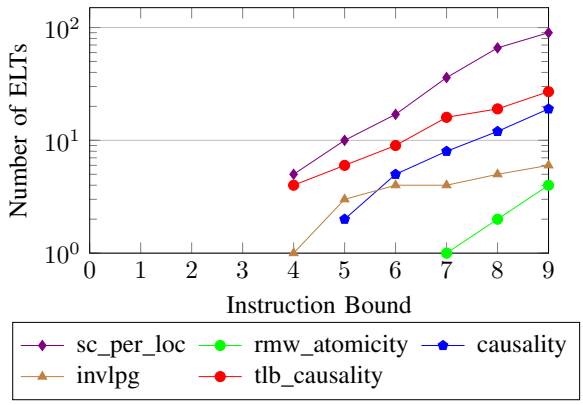
B. Synthesis Approach

Given `x86t_elt` as defined in §V-A as input, TransForm's synthesis engine generates a suite of ELT programs. The synthesized ELTs must constitute a forbidden program execution (according to the `x86t_elt` transistency predicate in this case) that becomes permitted under every possible relaxation. To synthesize ELTs that can result in forbidden outcomes, we identify (in turn) each of the axioms that comprise `x86t_elt` (i.e., `sc_per_loc`, `rmw_atomicity`, `causality`, `invlpg`, `tlb_causality`) as an axiom to be violated (and thus render the synthesized ELT executions forbidden). Synthesizing an ELT that violates one of these axioms directly corresponds to synthesizing a forbidden ELT execution.

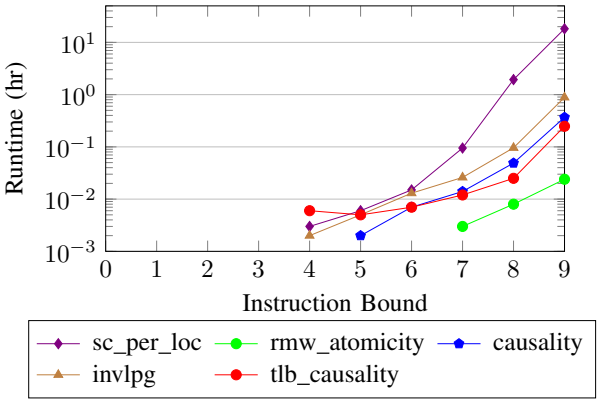
We synthesize five ELT suites, each containing tests that correspond to violations of one of the five `x86t_elt` axioms, up to a bound of 9 instructions. Each suite requires a minimum instruction bound of 4 instructions or higher, depending on the number of instructions needed to form interactions that can violate the respective axiom and constitute the test a part of the spanning set. Thus, synthesis begins at a bound of 4 instructions and increases up to the specified bound.

VI. Results

We supplied our `x86t_elt` MTM (from §V) consisting of five high-level axioms to TransForm's synthesis engine. We evaluated TransForm's ability to synthesize spanning sets of ELTs for `x86t_elt` with increasing instruction counts up to a bound of 9 instructions. For each axiom, the synthesis starts at the minimal instruction count for ELTs for that axiom—4-7 instructions for the axioms shown here. For each of the five high-level axioms TransForm synthesizes a set of ELTs up to the provided bounds. We refer to each of these five cases as a per-axiom suite. The rest of this section details our synthesis observations, compares the TransForm-generated ELTs to a baseline (the hand-generated COATCheck ELT suite [29]), and gives examples of synthesized ELTs.



(a) Plot of the number of ELTs synthesized in each suite by instruction bound. The first point for each type of suite corresponds to the minimum instructions required to synthesize that type of suite.



(b) Plot of the runtimes for synthesizing each suite by instruction bound. Although the synthesis runtimes grow super-exponentially with instruction bound, our ELT optimizations (§III) enable 9-instruction synthesis bounds to result in over a hundred useful ELTs within practical runtimes. We believe that future work on symmetry reductions and other optimizations can further accelerate these synthesis times.

Fig. 9: Statistics on our synthesized ELTs. (a) plots the number of instructions in each synthesized test suite while (b) plots the runtimes for synthesizing each of them.

A. Overview of Synthesized Suite

Fig. 9a summarizes the number of ELT programs synthesized in each per-axiom suite, at increasing instruction bounds. Fig. 9b shows the corresponding execution time required to synthesize them. Instruction bounds with no data points (i.e. fewer than 4 instructions) were too restrictive to synthesize ELTs satisfying our spanning set criteria. Due to the variability among our evaluated axioms, they require a variable minimal instruction bound to produce non-empty ELT spanning sets.

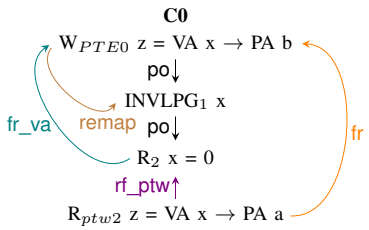
Over a hundred ELTs are generated automatically. At each instruction bound, the `sc_per_loc` suite makes up the largest component of the full synthesized suite. This is in part because the `sc_per_loc` axiom specifies ordering constraints on *all* types of instructions that TransForm models (i.e., user-facing, support, and ghost). This translates into more possibilities for violating this axiom, therefore more ELTs that qualify as interesting based on our criteria for vector space inclusion (in §IV-B). The prior automated MCM litmus test synthesis in [30] reports that the `sc_per_loc` suite saturates at 10 tests for the `x86-TSO` MCM. Because of the richer interactions in MTMs, many more tests are generated in our corresponding synthesis runs here for `x86t_elt`. Overall, the value of TransForm's ELT synthesis is two-fold. First, the automatic synthesis of hundreds of minimal and interesting MTM ELTs offers huge support for systems programmers and transistency verifiers. Second, the methodology rests on a foundational definition of *completeness* up to the specified synthesis bound; this gives designers a clear understanding of the comprehensiveness of their verification approach.

The TransForm synthesis approach generates a complete suite of ELTs for the 9-instruction synthesis bound provided for these experiments.

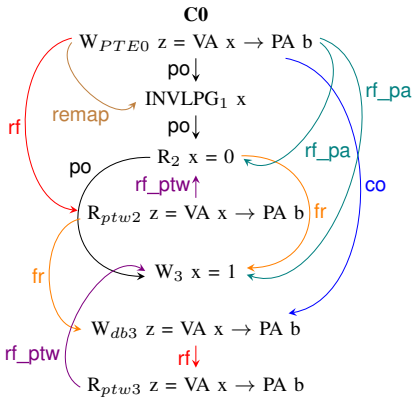
B. Comparison Against Prior Work

For reference, we compare our TransForm-synthesized ELT suite for `x86t_elt` with [29]'s handwritten suite of 40 ELTs. Of the original 40, 22 ELTs are relevant for comparison. Of the other 18, 9 deal with particular IPIs that are not presently supported by TransForm, and 9 others do not meet TransForm's spanning set criteria for ELTs. In contrast, TransForm synthesizes a total of 103 unique ELTs across all per-axiom suites (for a 9-instruction synthesis bound).

To facilitate comparison, we consider the 22 prior handwritten ELTs as two categories. First, there are ELTs which pass the minimality criterion and would be synthesized verbatim by TransForm. Second, there are ELTs which are not minimal as-is but are a superset of a minimal ELT. The extraneous instructions in the latter set of ELTs can be removed, exposing a minimal ELT that TransForm would synthesize. We automate the ELT comparison process via a tool that first checks if TransForm would synthesize the ELT verbatim in the synthesized suite (category 1), and if not, subsequently tests for category 2 by trying to remove subsets of instructions from the ELT to see if it can be minimized to a TransForm-synthesizable test. We bound this comparison at 9 instructions to match our experimental synthesis bound.



(a) This figure illustrates the forbidden `ptwalk2` ELT from the COATCheck suite that TransForm synthesized.



(b) This figure illustrates the permitted `dirtybit3` ELT from the COATCheck suite that can be reduced to a program that meets the minimality criterion and can be synthesized by TransForm.

Fig. 10: (a) and (b) illustrate COATCheck ELTs that were synthesized by TransForm either verbatim or by removing extraneous instructions, respectively.

Six of the 22 ELTs from the COATCheck suite fall into the first category and are synthesized verbatim. These 6 ELTs match 3 synthesized ELT programs. Recall that our tool outputs ELT *programs* whereas ELTs typically describe programs *and* their *outcomes* (i.e., an ELT execution), so some of our synthesized ELT programs might correspond to more than one ELT execution from the COATCheck suite. We additionally find 15 ELTs from the COATCheck suite that fall into the second category. These ELTs can be reduced to at least 1 minimal ELT which is synthesized by TransForm. We consider such minimal ELTs synthesized by TransForm to be unique new ELTs, as they were not explicitly part of the handwritten COATCheck suite. Finally, in addition to the 6 and 15 previously discussed, the remaining ELT from the handwritten 22 ELT COATCheck suite requires an instruction bound greater than 9 to be synthesized. Thus, of the 103 ELTs synthesized by TransForm, we find that 3 match ELTs from the 22 ELT COATCheck suite and 100 are new.

C. Examples of Synthesized ELTs

Figs. 10a and 11 provide examples of synthesized ELTs. Fig. 10a illustrates an ELT synthesized by TransForm and an exact match to a category-1 example (`ptwalk2`) from

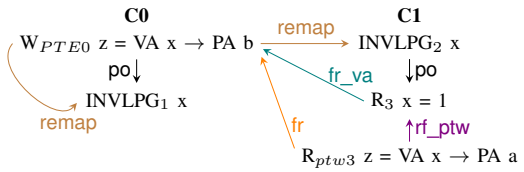


Fig. 11: This figure illustrates a forbidden candidate execution of a new synthesized ELT.

the COATCheck suite. The outcome shown violates both `sc_per_loc` and `invlpg` so it is forbidden.

Fig. 10b illustrates a handwritten dirty-bit ELT from the COATCheck suite. It is one of the 15 category-2 ELTs in our comparison, meaning it is not minimal in its handwritten form. TransForm automatically synthesizes a reduction of this ELT. Our comparison tool identifies a possible reduction (i.e., a subset of extraneous instructions that can be removed) of this ELT (specifically, the removal of W_3) that renders it a minimal TransForm-synthesized test.

Fig. 11 illustrates an example of a new synthesized ELT that is not found in the handwritten suite. In it, a system-level PTE `Write`, W_{PTE0}, is invoked by a system call, and remaps VA `x`. W_{PTE0}'s mapping update induces two `INVLPG`s: $INVLPG_1$ and $INVLPG_2$. $INVLPG_2$ precedes R_3 which reads from VA `x`. This particular execution has a forbidden outcome because even though R_3 comes after $INVLPG_2$ in `po`, it accesses a stale address mapping, as indicated by `fr_va`. This execution violates `invlpg` since there is a resulting cycle in `remap`, `fr_va`, and $^\wedge$`po`, and is thus forbidden by `x86t_elt`.

TransForm's framework for specifying MTMs and automatically synthesizing ELTs paves the way for systems programmers to perform deep verification and validation of MTM implementations. In future work, we plan to use the synthesized ELTs to empirically validate `x86t_elt` against real-world operating systems and x86 processor implementations.

VII. Related Work

Formal MCM Specifications: From their earliest roots [27], MCMs have been studied extensively over the years. Programming language-level MCMs have been formalized for Java, C11, and OpenCL [9, 11, 13, 36, 39, 43, 56]. Additionally, formal ISA-level MCM specifications exist for x86-TSO, Power, ARMv7, ARMv8, RISC-V WMO and TSO, and NVIDIA PTX [3, 38, 40, 41, 44, 55]. These MCM specification efforts have given way to verified compiler mapping schemes from C11 and Java MCM primitives to the x86, ARMv7, ARMv8, and Power ISAs [10, 11, 26, 43, 49, 50, 54]. Recently, the MCM for the Linux Kernel was also formalized [1]. TransForm assists programmers and compiler writers in developing correct system code for VM implementations by offering specification and verification support.

Verification of Hardware MCM Implementations: Formal ISA MCM specifications have prompted research on verifying the correctness of hardware MCM implementations [28, 33–35, 53]. Much of this prior work conducts bounded verification for suites of MCM litmus tests [28, 34, 35, 53] while some is proof-based [16, 33]. The COATCheck tool from this line of work also proposed a mechanism for verifying MTM implementations [29]. However, this work relied on hand-crafted ELTs and did not formally describe them or an MTM which could be used to generate them. In contrast, TransForm can be used to generate ELTs for expanding coverage of hardware MTM verification.

Expanding the Scope of Concurrency Specifications: A variety of research efforts formally define concurrency specifications beyond memory consistency. *Crash consistency* has been proposed to describe the ordering behavior of file system state updates across crashes [14]. *Memory persistency* has been coined for reasoning about the order in which nonvolatile memory (NVRAM) writes persist to memory [42]. Recently, persistency models have been formalized in the context of the release consistency [24], x86-TSO [45], and ARMv8 [46] MCMs.

VIII. Conclusion

TransForm is a framework for formally specifying MTMs and synthesizing ELTs, to support systems programmers and hardware designers verifying MTM behaviors. MTMs are central for assuring correct consistency behavior in the face of intricate VM interactions. TransForm includes a vocabulary for formally specifying MTMs and an automated synthesis approach for corresponding ELTs. To evaluate TransForm, we used its vocabulary to specify `x86t_elt`, an estimated MTM for x86 processors. From this MTM, we used TransForm to automatically synthesize its corresponding ELTs. TransForm's synthesis engine automatically produces a set of ELTs including relevant hand-curated ELTs from prior work, plus 100 more. TransForm is open source and publicly available at github.com/naorinh/TransForm.git. Overall, this work showcases the value and potential impact TransForm brings to MTM verification.

Acknowledgments

Thanks to Daniel Lustig and Yatin Manerkar for helpful feedback. This work was supported in part by Intel Corp. and through NSF CISE XPS-16-28926.

References

[1] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening small children and disconcerting grown-ups: Concurrency in the Linux Kernel," in *Proc. 23rd Intl. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, 2018.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," in *17th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS): Part of Joint European Confs. on Theory and Practice of Software (ETAPS)*, 2011.

[3] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 2, 2014.

[4] AMD, "Revision guide for AMD Athlon 64 and AMD Opteron processors," 2009, https://www.amd.com/system/files/TechDocs/25759.pdf.

[5] AMD, "Revision guides for AMD family processors," 2020, https://developer.amd.com/resources/developer-guides-manuals/.

[6] ARM, "Architecture reference manual, ARMv7-A and ARMv7-R edition," 2008.

[7] ARM, "Cortex-A9 MPCore, programmer advice notice, read-after-read hazards, ARM reference 761319," 2011.

[8] ARM, "ARM architecture reference manual, ARMv8, for ARMv8-A architecture profile," 2013, https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.188333416.1311159459.1564164180-4703051.1564164131.

[9] M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling SC atomics in C11 and OpenCL," in *43rd ACM Symp. on Principles of Programming Languages (POPL)*, 2016.

[10] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ concurrency: From C++11 to POWER," *39th ACM Symp. on Principles of Programming Languages (POPL)*, 2012.

[11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," *38th ACM Symp. on Principles of Programming Languages (POPL)*, 2011.

[12] A. Bhattacharjee and D. Lustig, "Architectural and operating system support for virtual memory," in *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, 2017.

[13] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *29th Conf. on Programming Language Design and Implementation (PLDI)*, 2008.

[14] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang, "Specifying and checking file system crash-consistency models," *21st Intl. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, 2016.

[15] J. Bornholt and E. Torlak, "Synthesizing memory models from framework sketches and litmus tests," *38th Conf. on Programming Language Design and Implementation (PLDI)*, 2017.

[16] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Prog. Lang.*, 2017.

[17] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Springer Berlin Heidelberg, 2004, pp. 502–518.

[18] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symp. on Security and Privacy (S&P)*, 2016.

[19] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," *31st Intl. Symp. on Computer Architecture (ISCA)*, 2004.

[20] IBM, "Power ISA version 2.07," 2013. [Online]. Available: https://ibm.ent.box.com/s/jd5w15gz301s5b5dt375mshpq9c3lh4u

[21] Intel, "Intel® Itanium architecture software developer's manual, revision 2.3," 2010.

[22] Intel, "Intel® Xeon® Processor 5400 Series Specification Update," 2013.

[23] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[24] J. Izraelevitz, H. Mendes, and M. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *30th Intl. Symp. on Distributed Computing (DISC)*, 2016.

[25] D. Jackson, "Alloy Analyzer website," 2012, http://alloy.mit.edu/.

[26] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," *38th Conf. on Programming Language Design and Implementation (PLDI)*, 2017.

[27] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computing*, 1979.

[28] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," in *47th Intl. Symp. on Microarchitecture (MICRO)*, 2014.

[29] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying memory ordering at the Hardware-OS interface," in *21st Intl. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, 2016.

[30] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *22nd Intl. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, 2017.

[31] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Generating litmus tests for contrasting memory consistency models," in *22nd Intl. Conf. on Computer Aided Verification (CAV)*, 2010.

[32] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, "An axiomatic memory model for POWER multiprocessors," *24th Intl. Conf. on Computer Aided Verification (CAV)*, 2012.

[33] Y. A. Manerkar, D. Lustig, M. Martonosi, and A. Gupta, "PipeProof: Automated memory consistency proofs for microarchitectural specifications," *51st Intl. Symp. on Microarchitecture (MICRO)*, 2018.

[34] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," in *50th Intl. Symp. on Microarchitecture (MICRO)*, 2017.

[35] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb graphs to verify the coherence-consistency interface," in *48th Intl. Symp. on Microarchitecture (MICRO)*, 2015.

[36] J. Manson, W. Pugh, and S. Adve, "The Java memory model," *32nd Symp. on Principles of Programming Languages (POPL)*, 2005.

[37] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the ARM and POWER relaxed memory models (2012)," 2012.

[38] V. Nagarajan, D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[39] K. Nienhuis, K. Memarian, and P. Sewell, "An operational semantics for C/C++11 concurrency," in *31st Intl. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2016.

[40] NVIDIA, "Parallel thread execution ISA version 6.0." 2017, http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[41] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," *22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[42] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *41st Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[43] G. Petri, J. Vitek, and S. Jagannathan, "Cooking the books: Formalizing JMM implementation recipes," *29th European Conf. on Object-Oriented Programming (ECOOP)*, 2015.

[44] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8," *ACM Programming Languages*, 2017.

[45] A. Raad and V. Vafeiadis, "Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model," *33rd Intl. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2018.

[46] A. Raad, J. Wickerson, and V. Vafeiadis, "Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models," *34th Intl. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2019.

[47] B. Romanescu, A. Lebeck, and D. J. Sorin, "Address translation aware memory consistency," *IEEE Micro*, vol. 31, no. 1, pp. 109–118, 2011.

[48] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin, "Specifying and dynamically verifying address translation-aware memory consistency," in *Proc. 15th Intl. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, 2010.

[49] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams, "Synchronising C/C++ and POWER," in *33rd Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2012.

[50] P. Sewell, "C/C++11 mappings to processors," 2016. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

[51] A. L. Shimpi, "AMD's B3 stepping phenom previewed, TLB hardware fix tested," 2008. [Online]. Available: https://www.anandtech.com/show/2477/2

[52] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *13th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[53] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *22nd Intl. Conf. on Arch. Support for Prog. Languages and Operating Systems (ASPLOS)*, 2017.

[54] J. Ševčík and D. Aspinall, "On validity of program transformations in the Java memory model," *Proceedings of the 22nd European Conf. on Object-Oriented Programming*, 2008.

[55] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20190608-base-ratified," RISC-V Foundation, Tech. Rep., March 2019. [Online]. Available: https://riscv.org/specifications/

[56] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-scope promotion: Clarified, rectified, and verified," *30th Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.