TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests

Naorin Hossain
Princeton University

January 22, 2021

Page replacement needed when memory is full and data from disk is being accessed

*addr0 = new_value  *requires dirty bit update*

\[
data = *addr1\]

### Page Table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>pg0</td>
</tr>
<tr>
<td>addr1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>disk</td>
</tr>
</tbody>
</table>

Main memory

- addr0
- old_value
- new_value

Disk

- pg0

Cache
Page replacement needed when memory is full and data from disk is being accessed

\*addr0 = new_value

\*data = \*addr1

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>A D R W</td>
</tr>
<tr>
<td>addr0</td>
</tr>
<tr>
<td>addr1</td>
</tr>
</tbody>
</table>

Main memory

Disk

new_value

old_value

addr0

Cache
Page replacement needed when memory is full and data from disk is being accessed

\[
\begin{align*}
*\text{addr0} & = \text{new\_value} \\
* & \\
* & \\
\text{data} & = *\text{addr1}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A D R W</td>
<td>pg0</td>
</tr>
<tr>
<td>addr0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>addr1</td>
<td>0 0 1 0</td>
</tr>
</tbody>
</table>
Page replacement needed when memory is full and data from disk is being accessed

\[ \text{*addr0} = \text{new\_value} \]

\[ \text{data} = \text{*addr1} \]

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A D R W</td>
<td></td>
</tr>
<tr>
<td>addr0</td>
<td>0 0 1 1 disk</td>
</tr>
<tr>
<td>addr1</td>
<td>0 0 1 0 disk</td>
</tr>
</tbody>
</table>

Main memory

Disk

Cache

new_value
Page replacement needed when memory is full and data from disk is being accessed

*addr0 = new_value
  * 
  * 
  *

data = *addr1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>disk</td>
</tr>
<tr>
<td>addr1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>pg0</td>
</tr>
</tbody>
</table>
What if dirty bit is not updated before page swapped to disk?

\[
\text{*addr0} = \text{new_value} \quad \text{requires dirty bit update*}
\]

\[
\text{data} = \text{*addr1}
\]

<table>
<thead>
<tr>
<th>Page Table</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>pg0</td>
</tr>
<tr>
<td>addr1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>disk</td>
</tr>
</tbody>
</table>

Diagram:
- Main memory
- Disk
- Cache
- addr0
- old_value
- new_value
What if dirty bit is not updated before page swapped to disk?

*addr0 = new_value
* 
* 

\[
data = *addr1
\]

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>addr0</td>
<td>0</td>
</tr>
<tr>
<td>addr1</td>
<td>0</td>
</tr>
</tbody>
</table>

Main memory

Disk

Cache

old_value

new_value
What if dirty bit is not updated before page swapped to disk?

\[
*addr0 = \text{new\_value} \\
* \\
* \\
\text{data} = *addr1
\]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>disk</td>
</tr>
<tr>
<td>addr1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>disk</td>
</tr>
</tbody>
</table>

![Diagram of memory management](image)
What if dirty bit is not updated before page swapped to disk?

`*addr0 = new_value
*`  
`*`  
`*`  
`*`  
`data = *addr1
*`  
`*`  
`*`  
`*`  
`data2 = *addr0`

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>disk</td>
</tr>
<tr>
<td>addr1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>pg0</td>
</tr>
</tbody>
</table>
What if dirty bit is not updated before page swapped to disk?

Like user-facing memory operations, the dirty bit updates are another memory reference that requires **correct ordering** for correct program executions.
Memory Consistency Models (MCMs) specify observable behaviors for concurrent programs

- MCMs specify rules for legal values that can be returned when software loads from memory on a shared memory system.

  1. Hardware can be implemented against specification
  2. Software can be modified accordingly (e.g., insert memory fences where needed)

This work: Memory Transistency Models (MTMs) – the superset of MCMs that additionally capture VM-aware ordering specifications
Prior work

Leslie Lamport defined first MCM: **sequential consistency** [Lamport 1979]

**Formal MCM specification examples:**
- Java [Manson et al., POPL ’05]
- C11 [Boehm & Adve, PLDI ’08]
- Linux Kernel [Alglave et al., ASPLOS ’18]
- x86-TSO [Owens et al., TPHOLs ’09]
- POWER [Mador-Haim et al., CAV ‘12]

**ISA-level MCM tools aiding verification:**
- diy tool [Alglave et al., CAV ‘10]
- litmus tool [Alglave et al., TACAS ‘11]
- herd tool [Alglave et al., TOPLAS ‘14]
- Automated MCM litmus test synthesis [Lustig et al., ASPLOS 2017]

**Check tool suite:**
- TriCheck [Trippel et al., ASPLOS ’17]
- COATCheck [Lustig et al., ASPLOS ’16]
- PipeCheck [Lustig et al., MICRO ‘14]
- CCICheck [Manerkar et al., MICRO ‘15]
- PipeProof [Manerkar et al., MICRO ‘18]
- RTLCheck [Manerkar et al., MICRO ‘17]
Prior work

Specification and verification of MTMs

Romanescu et al. introduce the concept of separate MCMs for VAs and PAs [ASPLOS ’10]

Check tool suite:

COATCheck [Lustig et al., ASPLOS ‘16]
TransForm introduces constructs for ISA-level MTM specification and ELT synthesis

• Formal MTM vocabulary captures system- and hardware-level VM events and interactions with user-facing program instructions
• Enables ISA-level MTM specification
• Enables automated *enhanced litmus test (ELT)* synthesis

Allows for verification against *formally specified MTM*
Outline

TransForm

- Background on ISA-level MCM vocabulary
- Background on virtual memory systems
- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: an estimated MTM for x86
- Future Work & Conclusions

Prior Work

My Work

MTM vocabulary

MCM vocabulary

hardware operations

system operations

our work

prior work

used for

ISA-specific MTM

ELT synthesis engine

suite of ELTs
Approach to defining vocabulary for formally reasoning about MTMs

• MCMs can be defined **axiomatically**
  • Axiomatic MCM specifications use sets of **relations** that can describe user-facing program executions
  • MCM relations describe user-facing event executions of programs with one-to-one V-to-P address mappings

• MTMs are **superset** of MCMs
  • Axiomatic MTM specifications can use MCM relations but require additional relations to similarly describe transistency events and V-to-P address mappings that can have synonyms and be modified
ISA-level MCM relations can describe programs and their candidate executions

Candidate executions – set of possible executions of a program and their outcomes

<table>
<thead>
<tr>
<th>Program</th>
<th>Instructions</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td>a. st data = 1;</td>
<td>W₀ x = 1</td>
</tr>
<tr>
<td></td>
<td>b. st flag = 1;</td>
<td>R₂ y = 1</td>
</tr>
<tr>
<td>Core 1</td>
<td>c. ld r1 = flag;</td>
<td>po</td>
</tr>
<tr>
<td></td>
<td>d. ld r2 = data;</td>
<td>fr</td>
</tr>
</tbody>
</table>

Candidate execution

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀ x = 1</td>
<td>R₂ y = 1</td>
</tr>
<tr>
<td>W₁ y = 1</td>
<td>R₃ x = 0</td>
</tr>
</tbody>
</table>

Communication (com) relations

reads from (rf)

rf = {W₁ → R₂}

coherence order (co)

co = {}

from reads (fr)

fr = {R₃ → W₀}

Numerical subscripts serve as instruction ID.

Accessed data (outcome) symbolically represented by com relations

[Shasha & Snir, 1988]
[Alglave et al., 2014]
ISA-level MCM relations can describe programs and their candidate executions

Candidate executions – set of possible executions of a program and their outcomes

Program

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀ x</td>
<td>R₂ y</td>
</tr>
<tr>
<td>W₁ y</td>
<td>R₃ x</td>
</tr>
</tbody>
</table>

Instructions

- **Event** = \{W₀, W₁, R₂, R₃\}
- **MemoryEvent** = \{W₀, W₁, R₂, R₃\}
- **Location** = \{x, y\}
- **address** = \{W₀ \rightarrow x, W₁ \rightarrow y, R₂ \rightarrow y, R₃ \rightarrow x\}
- **program order (po)**
  \[ po = \{W₀ \rightarrow W₁, R₂ \rightarrow R₃\} \]

Candidate execution

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀ x = 1</td>
<td>R₂ y = 1</td>
</tr>
<tr>
<td>W₁ y = 1</td>
<td>R₃ x = 0</td>
</tr>
</tbody>
</table>

Communication (com) relations

- **reads from (rf)**
  \[ rf = \{W₁ \rightarrow R₂\} \]
- **coherence order (co)**
  \[ co = {} \]
- **from reads (fr)**
  \[ fr = \{R₃ \rightarrow W₀\} \]

Graph

- **Core 0**
  - \( W₀ x = 1 \)
  - \( po \leftarrow fr \)
  - \( W₁ y = 1 \)
- **Core 1**
  - \( R₂ y = 1 \)
  - \( R₃ x = 0 \)
  - \( po \rightarrow fr \)

Numerical subscripts serve as instruction ID.

Accessed data (outcome) symbolically represented by com relations

[Shasha & Snir, 1988]
[Alglave et al., 2014]

19
MCM specifications place constraints on permitted execution behaviors

Axiomatic MCM specifications use MCM relations to describe axioms that constrain candidate execution behaviors

Intel x86 processors use the total store order (TSO) memory model (x86-TSO) [Owens et al., 2009]: strict sequential memory access orderings but relaxed Store->Load orderings to allow for store buffering

Causality – axiom for x86-TSO: acyclic(rfe + co + fr + ppo + fence)
MCM specifications place constraints on permitted execution behaviors

Axiomatic MCM specifications use MCM relations to describe axioms that constrain candidate execution behaviors

Intel x86 processors use the **total store order (TSO)** memory model (x86-TSO) [Owens et al., 2009]: strict sequential memory access orderings but relaxed Store->Load orderings to allow for store buffering

**Causality** – axiom for x86-TSO: acyclic(rfe + co + fr + ppo + fence)
Outline

TransForm

- MTM vocabulary
- MCM vocabulary
- hardware operations
- system operations

our work
prior work

used for

ISA-specific MTM → ELT synthesis engine → suite of ELTs

Prior Work

• Background on ISA-level MCM vocabulary
• Background on virtual memory systems
• Novel ISA-level MTM vocabulary
• Automating synthesis of ELTs
• Case Study: a estimated MTM for x86
• Future Work & Conclusions

My Work
V-to-P address mappings need to be stored and accessed during memory events

V-to-P address mappings are stored in **page tables**.

**Page table entries (PTEs)** hold address mapping and status bits (permissions, access, dirty).

Page tables are usually structured hierarchically.

When address translation is needed, a **page table walk** traverses the page table levels to find the desired address mapping.

---

V-to-P address mappings are cached in the *translation lookaside buffer* (TLB).

Hierarchical page tables require **additional** memory accesses during address translation – big performance hit.

Page mappings cached in TLB to reduce latency of memory accesses.
V-to-P address mappings can be changed by OS

Operating system (OS) may change address mappings in page tables.

Corresponding TLB entries must be invalidated on each core to prevent stale mapping accesses.
V-to-P address mappings can be changed by OS

Operating system (OS) may change address mappings in page tables.

Corresponding TLB entries must be invalidated on *each core* to prevent stale mapping accesses.

New page table walk needed to load new mapping into TLB.
Virtual memory events TransForm needs to support

<table>
<thead>
<tr>
<th>Hardware-level events</th>
<th>System-level events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page table walk</td>
<td>Address mapping changes</td>
</tr>
<tr>
<td>Loads TLB entries on memory access</td>
<td>V-to-P address mapping must be modifiable like data</td>
</tr>
<tr>
<td>PTE status bit updates</td>
<td>TLB entry invalidations</td>
</tr>
<tr>
<td>TransForm supports dirty bit updates on memory stores</td>
<td>May be invoked on multiple cores by address mapping changes</td>
</tr>
</tbody>
</table>
Outline

TransForm

MTM vocabulary
MCM vocabulary
hardware operations
system operations

our work
prior work

• Background on ISA-level MCM vocabulary
• Background on virtual memory systems
• Novel ISA-level MTM vocabulary
• Automating synthesis of ELTs
• Case Study: a estimated MTM for x86
• Future Work & Conclusions

Prior Work

My Work

used for
ISA-specific MTM
ELT synthesis engine
suite of ELTs
MTM Vocabulary: Hardware-level events

TransForm supports **page table walks (PTW)** and **dirty bit updates**

**Ghost instructions**

**PTW**: loads translation lookaside buffer (TLB) entry

**dirty bit update**: modifies dirty bit in PTE
MTM Vocabulary: Hardware-level events

TransForm supports **page table walks (PTW)** and **dirty bit updates**

**Ghost instructions**

**PTW**: loads translation lookaside buffer (TLB) entry

**dirty bit update**: modifies dirty bit in PTE

**ghost** – relates user-facing MemoryEvent to invoked ghost instructions (numerical subscripts)
MTM Vocabulary: Hardware-level events

TransForm supports **page table walks (PTW)** and **dirty bit updates**

**Ghost instructions**

**PTW**: loads translation lookaside buffer (TLB) entry

**Dirty bit update**: modifies dirty bit in PTE

**ghost** – relates user-facing MemoryEvent to invoked ghost instructions (numerical subscripts)

**rf_ptw** – relates PTW to user-facing MemoryEvents that access loaded TLB entry
MTM Vocabulary: Hardware-level events

TransForm supports **page table walks (PTW)** and **dirty bit updates**

**Ghost instructions**

**PTW**: loads translation lookaside buffer (TLB) entry

**dirty bit update**: modifies dirty bit in PTE

**ghost** – relates user-facing MemoryEvent to invoked ghost instructions (numerical subscripts)

**rf_ptw** – relates PTW to user-facing MemoryEvents that access loaded TLB entry
MTM Vocabulary: Hardware-level events

TransForm supports **page table walks (PTW)** and **dirty bit updates**

**Ghost instructions**

- **PTW**: loads translation lookaside buffer (TLB) entry
- **dirty bit update**: modifies dirty bit in PTE
- **ghost** – relates user-facing MemoryEvent to invoked ghost instructions (numerical subscripts)
- **rf_ptw** – relates PTW to user-facing MemoryEvents that access loaded TLB entry
TransForm supports **address remappings via PTE Writes** and **TLB entry invalidations**

**Support instructions**

**PTE Write**: changes address mapping stored in a PTE for some VA v

VAs y and x both map to PA a now so they are synonyms.
MTM Vocabulary: System-level events

TransForm supports **address remappings via PTE Writes** and **TLB entry invalidations**

**Support instructions**

**PTE Write**: changes address mapping stored in a PTE for some VA $v$

**INVLPG**: invalidates TLB entry (named after x86 instruction)

**remap** – relates PTE Writes to invoked INVLPGs
MTM Vocabulary: System-level events

TransForm supports **address remappings via PTE Writes** and **TLB entry invalidations**

**Support instructions**

**PTE Write**: changes address mapping stored in a PTE for some VA v

**INVLPG**: invalidates TLB entry (named after x86 instruction)

remap – relates PTE Writes to invoked INVLPGs

rf_pa – relates PTE Write for VA v → PA p to user-facing MemoryEvents accessing PA p via VA v

fr_pa – relates user-facing MemoryEvents accessing PA p via VA v to PTE Writes for VA v’ → PA p

c_o_pa and fr_va follow similarly

![Diagram showing PTE Write and INVLPG interactions]
MTM Vocabulary: System-level events

TransForm supports **address remappings via PTE Writes and TLB entry invalidations**

**Support instructions**

**PTE Write**: changes address mapping stored in a PTE for some VA v

**INVLPQ**: invalidates TLB entry (named after x86 instruction)

**remap** – relates PTE Writes to invoked INVLPQs

**rf_pa** – relates PTE Write for VA v → PA p to user-facing MemoryEvents accessing PA p via VA v

**fr_pa** – relates user-facing MemoryEvents accessing PA p via VA v to PTE Writes for VA v’ → PA p

**co_pa** and **fr_va** follow similarly
MTM Vocabulary: System-level events

TransForm supports **address remappings via PTE Writes and TLB entry invalidations**

These new com relations can be used to derive same PA accesses.

- **rf_pa** – relates PTE Write for VA \( v \rightarrow PA \ p \) to user-facing MemoryEvents accessing PA \( p \) via VA \( v \)
- **fr_pa** – relates user-facing MemoryEvents accessing PA \( p \) via VA \( v \) to PTE Writes for VA \( v' \rightarrow PA \ p \)
- **co_pa** and **fr_va** follow similarly
MTM Vocabulary: System-level events

TransForm supports **address remappings via PTE Writes** and **TLB entry invalidations**

These new com relations can be used to derive same PA accesses.

- **rf_pa** – relates PTE Write for VA $v \rightarrow PA\ p$ to user-facing MemoryEvents accessing PA $p$ via VA $v$
- **fr_pa** – relates user-facing MemoryEvents accessing PA $p$ via VA $v$ to PTE Writes for VA $v' \rightarrow PA\ p$
- **co_pa** and **fr_va** follow similarly

Forbidden!
Program executions with transistency events and relations can get quite complex but they allow us to capture these additional interactions that can occur and impact the program’s execution.
Outline

TransForm

MTM vocabulary
MCM vocabulary
hardware operations
system operations

 ISA-specific MTM
ELT synthesis engine
suite of ELTs

• Background on ISA-level MCM vocabulary
• Background on virtual memory systems
• Novel ISA-level MTM vocabulary
• Automating synthesis of ELTs
• Case Study: an estimated MTM for x86
• Future Work & Conclusions

Prior Work

My Work
MCMs can be verified with litmus tests. MTMs can be verified with *enhanced* litmus tests.

- **Litmus tests**: small diagnostic programs for validating MCM behaviors
  - Executions and their outcomes can be deemed *permitted* or *forbidden* by MCM specification

- **Enhanced litmus tests (ELTs)**: litmus tests enhanced with system- and hardware-level events that facilitate address translation
From Specification to Test Synthesis

- ELTs can be described with MTM relations and support verification against an MTM spec
- Goals:
  - Automated
  - Interesting and minimal ("Spanning set")
  - Deduplicated
  - Comprehensive (to a bound)
TransForm’s synthesis engine starts by synthesizing all possible candidate executions up to a bound.
Candidate executions are pruned for interesting ELT behaviors and checked for minimality.

TransForm Synthesis Engine

MTM spec in Alloy → Candidate Execution Synthesis → Candidate Executions (Alloy) → Relaxation rules → Spanning Set Pruning

Interesting ELT criteria:
1. #Write > 0
2. Forbidden by MTM
+ Minimality criterion

Pruned Candidate Executions (XML) → Unique ELT Pruning → Unique ELT suite

MTM spec
Synthesis bound
Relation placement rules

Candidate Executions

Candidate Execution Synthesis

Relaxation rules

Spanning Set Pruning

Interesting ELT criteria:
1. #Write > 0
2. Forbidden by MTM
+ Minimality criterion

Pruned Candidate Executions (XML)

Unique ELT Pruning

Unique ELT suite

C0
R0 x = 1
po↓
R1 x = 0

C1
W2 x = 1

C0
R0 x = 1
po↓
R1 x = 0

No Writes

C0
W0 x = 1
po↓
R1 y = 2

Permitted by MTM

C1
W0 x = 1
po↓
R1 y = 2

fr
C1
R1 x = 1

C0
W0 x = 1
po↓
R2 x = 0

R3 x = 1

fr
C1
R1 x = 1

C0
R0 x = 1
po↓
R1 x = 0

R3 x = 1

fr
C1
R1 x = 1

...
Unique ELTs are found by deduplicating synthesized ELTs with a post-processing script.
Outline

TransForm

- MTM vocabulary
- MCM vocabulary
- hardware operations
- system operations

Outline

- Background on ISA-level MCM vocabulary
- Background on virtual memory systems
- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: an estimated MTM for x86
- Future Work & Conclusions
**x86t_elt** predicates are composed of x86-TSO axioms and new transistency-specific axioms

- **x86t_elt**: an approximate x86 transistency model based on prior work and publicly available documentation
- **x86-TSO**: `sc_per_loc`, `rmw_atomicity`, causality
  
  - **invlpg (required)**
    - acyclic[fr_va + remap + ^po]
  
  - **tlb_causality (auxiliary)**
    - acyclic[ptw_source + com]
A per-axiom suite was synthesized for each `x86t_elt` axiom

140 total unique ELTs!
The synthesized x86t_elt suite consisted of all relevant ELTs from COATCheck and more

- All 22 relevant ELTs from COATCheck synthesized
  - 7 ELTs synthesized verbatim → map to 4 ELT programs in x86t_elt suite
  - 15 ELTs can be reduced to a minimal ELT that is synthesized
- 4 ELTs from COATCheck, 136 new ELTs
Outline

**TransForm**

- MTM vocabulary
- MCM vocabulary
- hardware operations
- system operations

**Legend**

- Green: our work
- Gray: prior work

**Pipeline**

- ISA-specific MTM
- ELT synthesis engine
- suite of ELTs

**Prior Work**

- Background on ISA-level MCM vocabulary
- Background on virtual memory systems
- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: an estimated MTM for x86

**My Work**

- Future Work & Conclusions
Ongoing and Future Work

- Empirical MTM testing
- Validate x86t elt and verify x86 processors using synthesized ELTs
- Specify other MTMs (e.g., RISC-V)
- Model additional transistency interactions (e.g., permission bit updates)
- Formally reason about transistency and security
Conclusions

• **TransForm**: framework for formal specification of MTMs and ELT synthesis

• Enables modern ISAs to have a formal specification that includes VM

• Offers systems programmers and hardware designers a stronger opportunity for verification of full systems

• Available at: [https://github.com/naorinh/TransForm](https://github.com/naorinh/TransForm)
TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests

Naorin Hossain
Princeton University
January 22, 2021

https://github.com/naorinh/TransForm
Backup Slides
Prior work developed a tool for verifying correctness of hardware MTM implementations

- **Enhanced litmus tests (ELTs):** litmus tests enhanced with hardware- and system-level interactions that facilitate the VM abstraction

1. Formal MTM verification at *microarchitectural*-level
2. Lacked formal MTM specification
3. Hand-generated ELT suites

Are ELT outcomes observable on μarch as expected?
Communication relations can be extended to apply to virtual-to-physical address mappings

- **rf_pa**: maps a PTE write mapping $\text{VA} v \rightarrow \text{PA} p$ to instructions that access $\text{VA} v \rightarrow \text{PA} p$.
- **fr_pa**: maps instructions that access $\text{VA} v \rightarrow \text{PA} p$ to PTE writes that map $\text{VA} v' \rightarrow \text{PA} p$.
- **co_pa**: maps PTE writes that change mappings for different VAs to the same PA.
- **fr_va**: maps instructions that access $\text{VA} v \rightarrow \text{PA} p$ to PTE writes that map $\text{VA} v \rightarrow \text{PA} p'$. 
Tests with instructions that do not contribute to forbidden outcomes violate minimality criterion

**Minimality criterion**: any relaxation on the test results in satisfying the transistency predicates.

**Relaxations** for transistency purposes are the removal of instructions and the following dependent instructions:
1. Invoked ghost instructions
2. Invoked INVLPGs
3. Dependent RMW operations

ELT execution should have a forbidden outcome that becomes legal under *every* possible isolated relaxation.

- **x86-TSO – causality axiom**: acyclic(rfe + co + fr + ppo + fence)
- Removal of $W_4$ as relaxation does not result in satisfying causality

\[
\begin{align*}
C0 & : W_0 \ x = 1 \\
C1 & : R_2 \ y = 1 \\
C2 & : W_4 \ u = 1
\end{align*}
\]

\[
\begin{align*}
& \text{po} \quad \text{rf} \\
W_0 & : x = 1 \\
R_2 & : y = 1 \\
W_1 & : y = 1 \\
R_3 & : x = 0
\end{align*}
\]