TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests

Naorin Hossain
Princeton University

FOCA 2020
October 30, 2020
Page replacement needed when memory is full and data from disk is being accessed

*addr0 = new_value  *requires dirty bit update*

data = *addr1

<table>
<thead>
<tr>
<th>Page Table</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>
Page replacement needed when memory is full and data from disk is being accessed.

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

---

**Page Table**

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

Diagram:
- Main memory
- Disk
- Cache
- addr0
- new_value
- pg0
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</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

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

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

<table>
<thead>
<tr>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr0</td>
</tr>
<tr>
<td>addr1</td>
</tr>
</tbody>
</table>

Diagram:
- Main memory
- Disk
- Cache

Symbols:
- addr0
- addr1
- new\_value
- data\_value
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>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>addr0</td>
</tr>
<tr>
<td>addr1</td>
</tr>
</tbody>
</table>

Diagram:
- Main memory
- Disk
- Cache
- \text{addr0} -> \text{new\_value}
- \text{addr0} -> \text{old\_value}
What if dirty bit is not updated before page swapped to disk?

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

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

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A D R W</td>
<td>pg0</td>
</tr>
<tr>
<td>addr0</td>
<td>0011</td>
</tr>
<tr>
<td>addr1</td>
<td>0010</td>
</tr>
</tbody>
</table>
What if dirty bit is not updated before page swapped to disk?

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

<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>
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>
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 correct event orderings for concurrent programs

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

- Core 0
  a. st data = 1;
  b. st flag = 1;
  c. ld r1 = flag;
  d. ld r2 = data;

- Core 1
  a. st data = 1;
  b. st flag = 1;
  c. ld r1 = flag;
  d. ld r2 = data;

**Problem**: don’t reason about program executions impacted by shared virtual memory (VM) state with existing MCM constructs

Litmus test: small diagnostic

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

• 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
• Conclusions

Prior Work

My Work
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

**Program**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_0 \ x$</td>
<td>$R_2 \ y$</td>
</tr>
<tr>
<td>$W_1 \ y$</td>
<td>$R_3 \ x$</td>
</tr>
</tbody>
</table>

**Instructions**

- **Event** = \( \{W_0, W_1, R_2, R_3\} \)
- **Memory Event** = \( \{W_0, W_1, R_2, R_3\} \)
- **Location** = \( \{x, y\} \)
- **address** \( \{W_0 \rightarrow x, W_1 \rightarrow y, R_2 \rightarrow y, R_3 \rightarrow x\} \)
- **program order (po)**
  \( \text{po} = \{W_0 \rightarrow W_1, R_2 \rightarrow R_3\} \)

**Candidate execution**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_0 \ x = 1$</td>
<td>$R_2 \ y = 1$</td>
</tr>
<tr>
<td>$W_1 \ y = 1$</td>
<td>$R_3 \ x = 0$</td>
</tr>
</tbody>
</table>

**Communication (com) relations**

- **reads from (rf)**
  \( \text{rf} = \{W_1 \rightarrow R_2\} \)
- **coherence order (co)**
  \( \text{co} = \{\} \)
- **from reads (fr)**
  \( \text{fr} = \{R_3 \rightarrow W_0\} \)

**Graph**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_0 \ x$</td>
<td>$R_2 \ y$</td>
</tr>
<tr>
<td>$W_1 \ y$</td>
<td>$R_3 \ x$</td>
</tr>
</tbody>
</table>

**Communication relations**

- **reads from (rf)**
  \( \text{rf} \)
- **coherence order (co)**
  \( \text{co} = \{\} \)
- **from reads (fr)**
  \( \text{fr} = \{R_3 \rightarrow W_0\} \)

**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>$W_0 \ x$</td>
<td>$R_2 \ y$</td>
</tr>
<tr>
<td>$W_1 \ y$</td>
<td>$R_3 \ x$</td>
</tr>
</tbody>
</table>

Instructions
- Event = \{W_0, W_1, R_2, R_3\}
- MemoryEvent = \{W_0, W_1, R_2, R_3\}
- Location = \{x, y\}
- address = \{W_0 \rightarrow x, W_1 \rightarrow y, R_2 \rightarrow y, R_3 \rightarrow x\}
- program order (po)
  - po = \{W_0 \rightarrow W_1, R_2 \rightarrow R_3\}

Communication (com) relations
- reads from (rf)
  - rf = \{W_1 \rightarrow R_2\}
- coherence order (co)
  - co = {}  
- from reads (fr)
  - fr = \{R_3 \rightarrow W_0\}

Graph

Core 0: $W_0 \ x = 1, W_1 \ y = 1, R_2 \ y = 1, R_3 \ x = 0$
Core 1: $R_0 \ x = 1, R_2 \ y = 1, R_3 \ x = 0$

Accessed data (outcome) symbolically represented by com relations

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

\[
\text{acyclic(rfe + co + fr + ppo + fence)}
\]
MCM specifications place constraints on permitted execution behaviors

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

**Sequential consistency (SC)** [Lamport, 1979]: outcome must be representative of executing instructions in order

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

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

Prior Work

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

- 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
- Conclusions

Prior Work

My Work
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
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: 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`

\[
\begin{array}{c|c}
\text{C0} & \text{C1} \\
W_0 & W_{PTE3} \\
x = 1 & v = VA \\
p_o & \rightarrow PA \\
R_2 & y = 2 \\
y = 2 & \rightarrow a \\
p_o & \\
R_6 & x = 1 \\
x = 1 & \\
p_o & \\
\end{array}
\]
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

![Diagram](image)
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

**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 \( v \)

**INVLPFG**: invalidates TLB entry

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

**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**

**Support instructions**

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

**INVLPG:** invalidates TLB entry

remap – relates PTE Writes to invoked INVLPGs

**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 → 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
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

• 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
• 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

TransForm Synthesis Engine

MTM spec in Alloy
Synthesis bound
Relation placement rules

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
in Alloy

TransForm’s synthesis engine starts by synthesizing all possible candidate executions up to a bound

MTM spec in Alloy
Candidate executions are pruned for interesting ELT behaviors and checked for minimality.

Candidate Execution Synthesis

TransForm Synthesis Engine

Candidate Executions Synthesis

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 in Alloy
Synthesis bound
Relation placement rules

Candidate Executions (Alloy)
Relaxation rules

C0
R₀ x = 1
po↓
R₁ x = 0
C1
W₂ x = 1
C0
No Writes
C1
Permitted by MTM
C0
R₀ x = 1
po↓
R₁ x = 0
R₁ y = 2
R₂ x = 0
C1
R₁ x = 1
po↓
R₃ x = 1
C0
W₀ x = 1
rf
fr
C1
R₀ x = 1
po↓
R₁ x = 0
R₁ y = 2
R₂ x = 0
R₃ x = 1
No minimal
C0
W₀ x = 1
rf
fr
C1
R₀ x = 1
po↓
R₁ x = 0
R₁ y = 2
R₂ x = 0
R₃ x = 1
No minimal
44
Unique ELTs are found by deduplicating synthesized ELTs with a post-processing script.

MTM spec in Alloy
Synthesis bound
Relation placement rules

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

TransForm Synthesis Engine

\[
\begin{array}{c|c|c}
C0 & C1 \\
R_0: x = 1 & W_2: x = 1 \\
po & \downarrow po \\
R_1: x = 0 & \end{array}
\]
\[
\begin{array}{c|c|c}
C0 & C1 \\
R_0: x = 1 & W_2: x = 1 \\
po & \downarrow po \\
R_1: x = 0 & \end{array}
\]
\[
\begin{array}{c|c|c}
C0 & C1 \\
W_0: x = 1 & W_2: y = 2 \\
po & \downarrow po \\
R_1: y = 2 & R_3: x = 1 \\
\end{array}
\]

Duplicate

\[
\begin{array}{c|c|c}
C0 & C1 \\
W_0: x = 1 & \end{array}
\]

\[
\begin{array}{c|c|c}
C0 & C1 \\
R_0: x = 1 & W_2: x = 1 \\
po & \downarrow po \\
R_1: x = 0 & \end{array}
\]

\[
\begin{array}{c|c|c}
C0 & C1 \\
R_0: x = 1 & W_2: x = 1 \\
po & \downarrow po \\
R_1: x = 0 & \end{array}
\]
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
- Conclusions
**x86t_elt** transistency predicates are composed of 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
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

• **Future work:**
  • Empirical MTM testing to validate/verify with x86t_elt
  • Specify other MTMs (e.g., RISC-V)
  • Model additional transistency interactions (e.g., updating permission bits)

• 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

FOCA 2020
October 30, 2020

https://github.com/naorinh/TransForm