TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests

Naorin Hossain
Princeton University

January 22, 2021

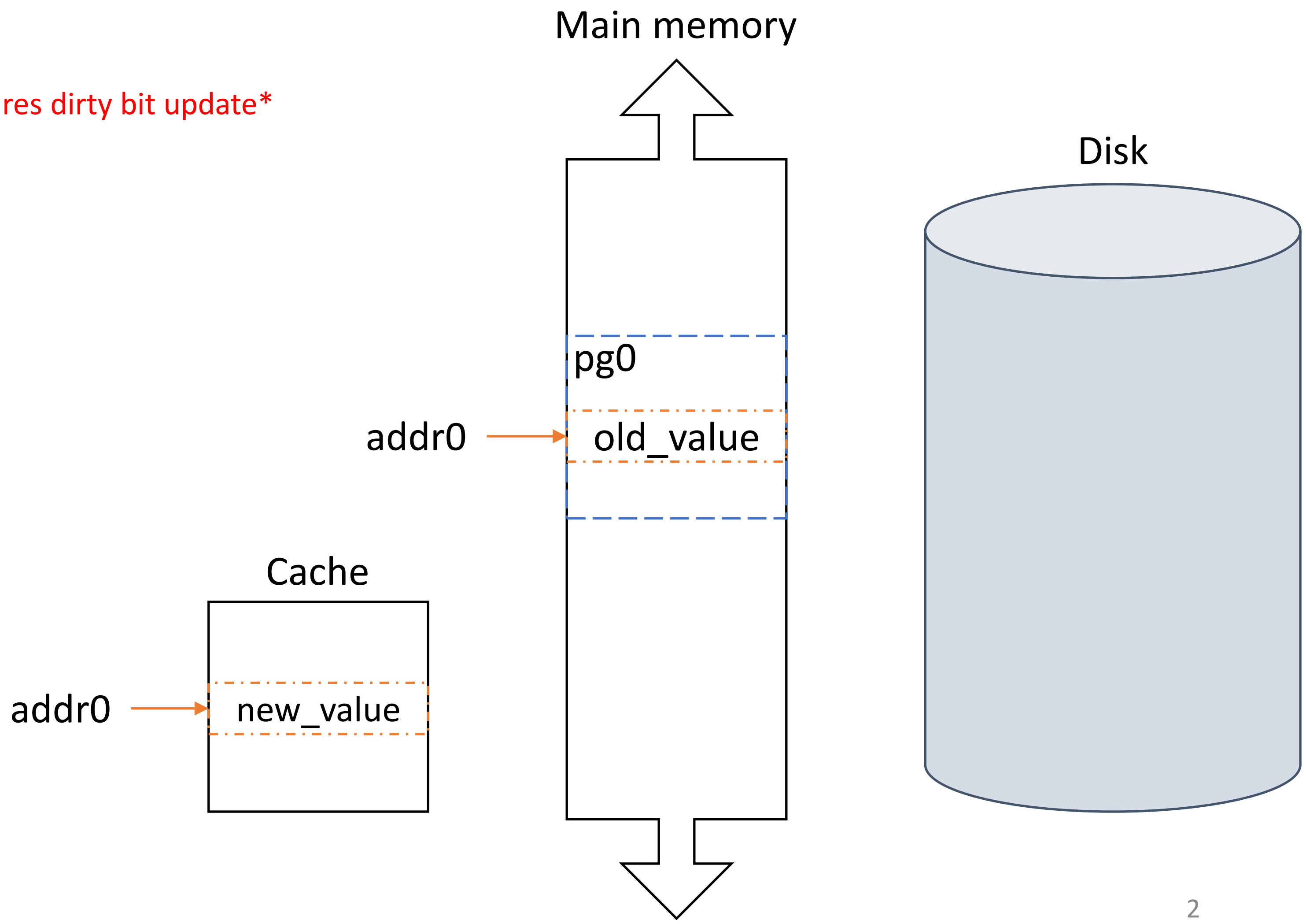
Naorin Hossain, Caroline Trippel, Margaret Martonosi. "TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests", 47th International Symposium on Computer Architecture (ISCA ’20).

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

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

requires dirty bit update

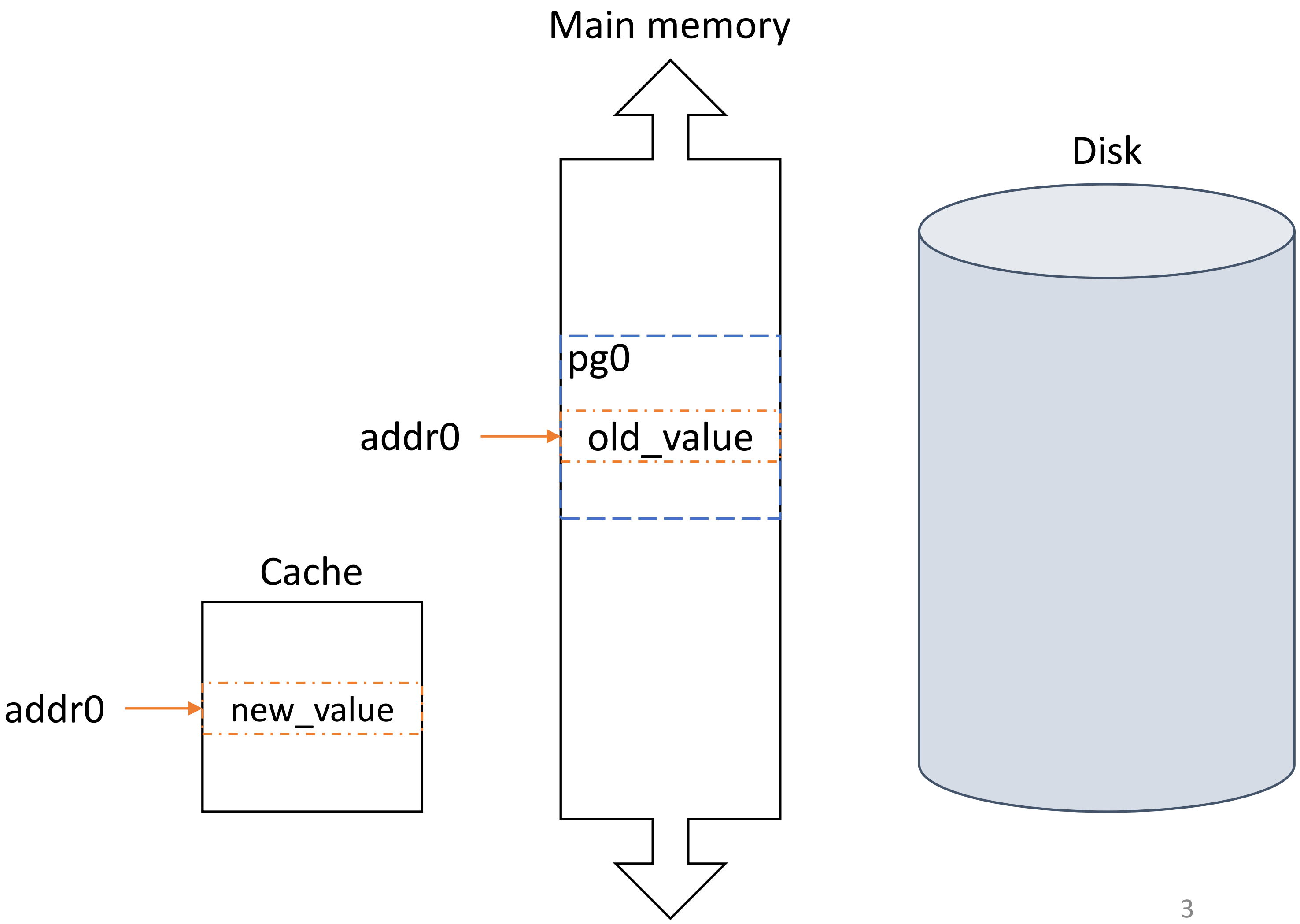
	Page Table				
	A	D	R	W	Physical page
addr0	0	1	1	1	pg0
addr1	0	0	1	0	disk



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

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

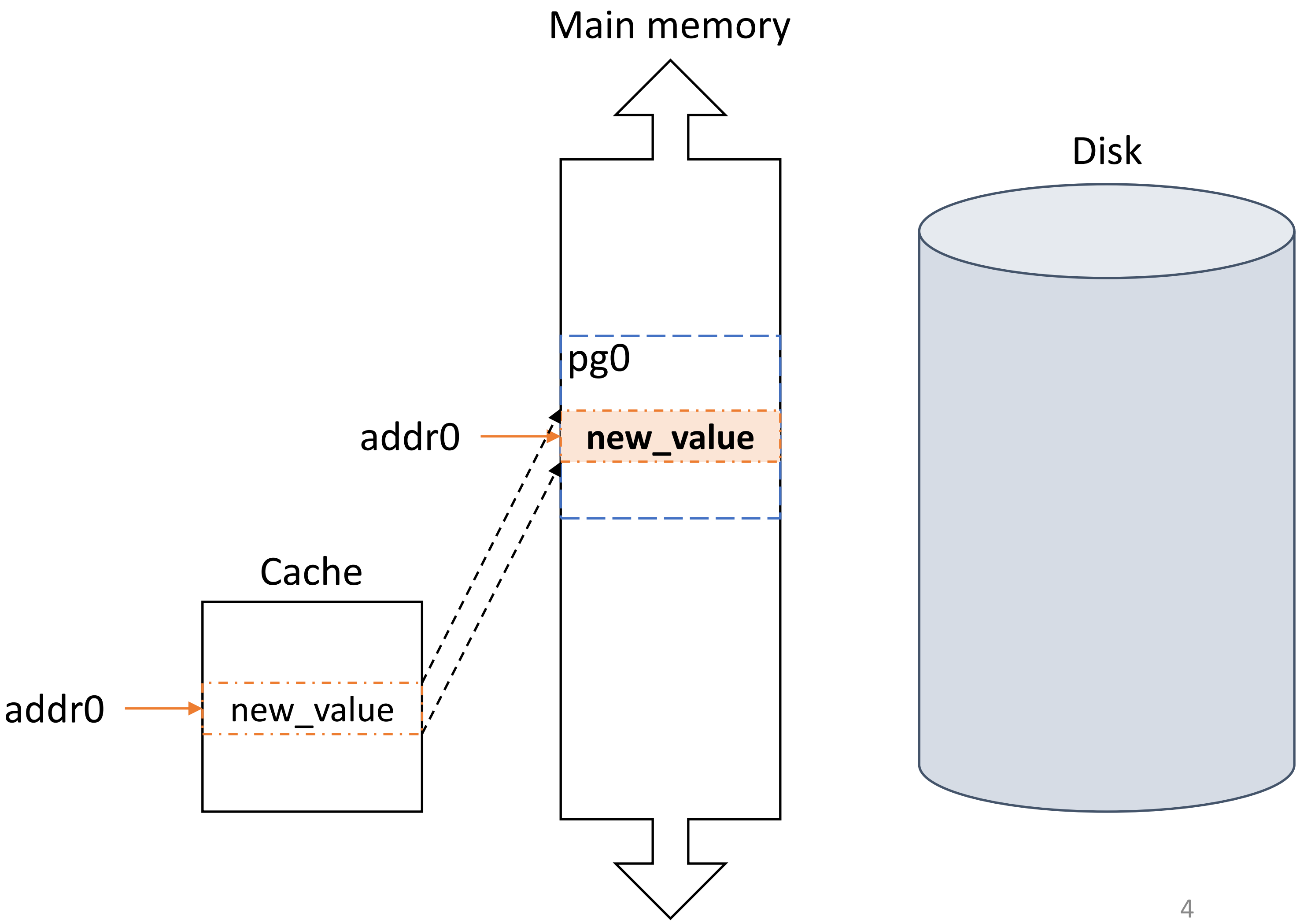
	Page Table				
	A	D	R	W	Physical page
addr0	0	1	1	1	pg0
addr1	0	0	1	0	disk



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

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

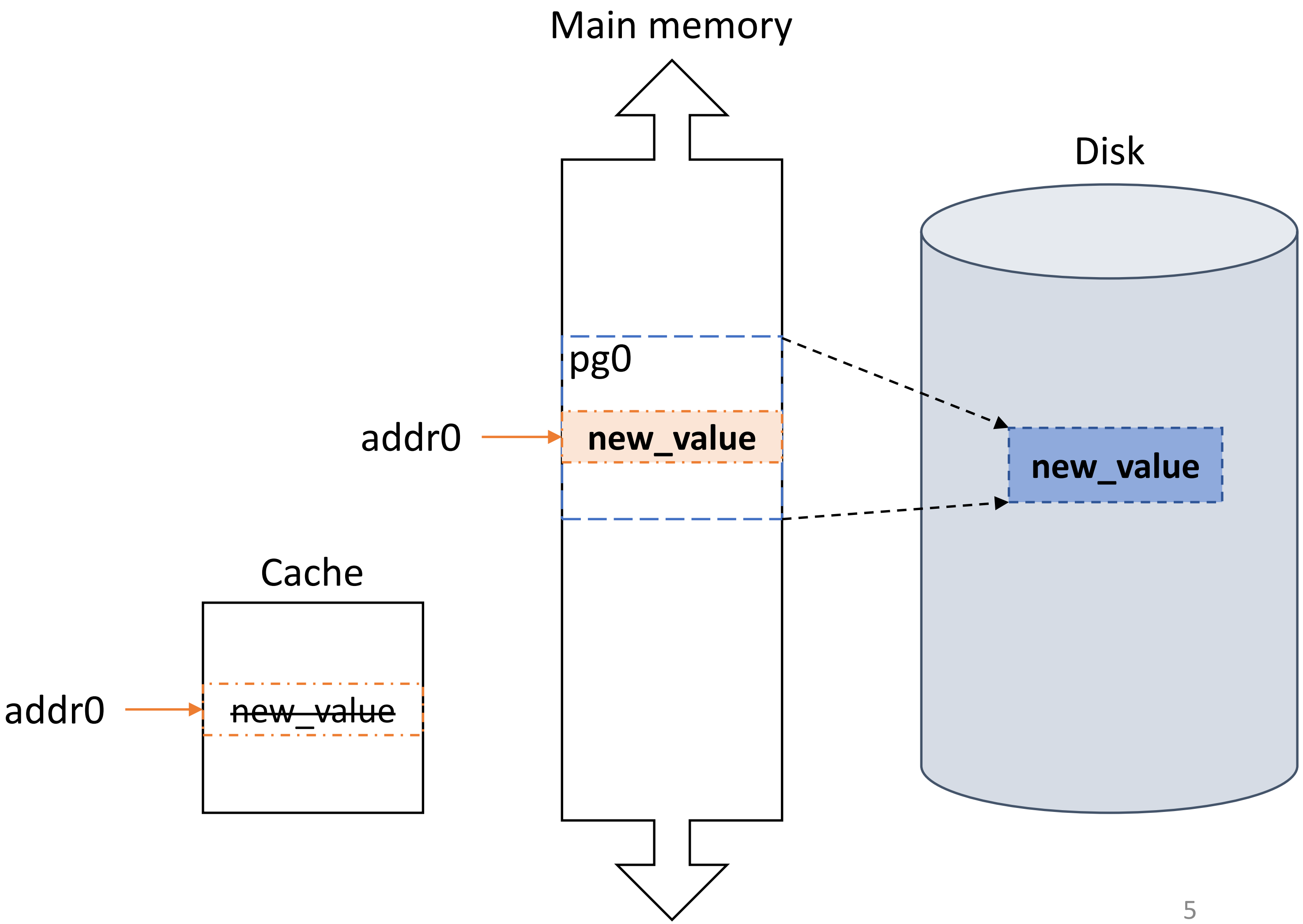
	Page Table				
	A	D	R	W	Physical page
addr0	0	**0**	1	1	pg0
addr1	0	0	1	0	disk



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

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

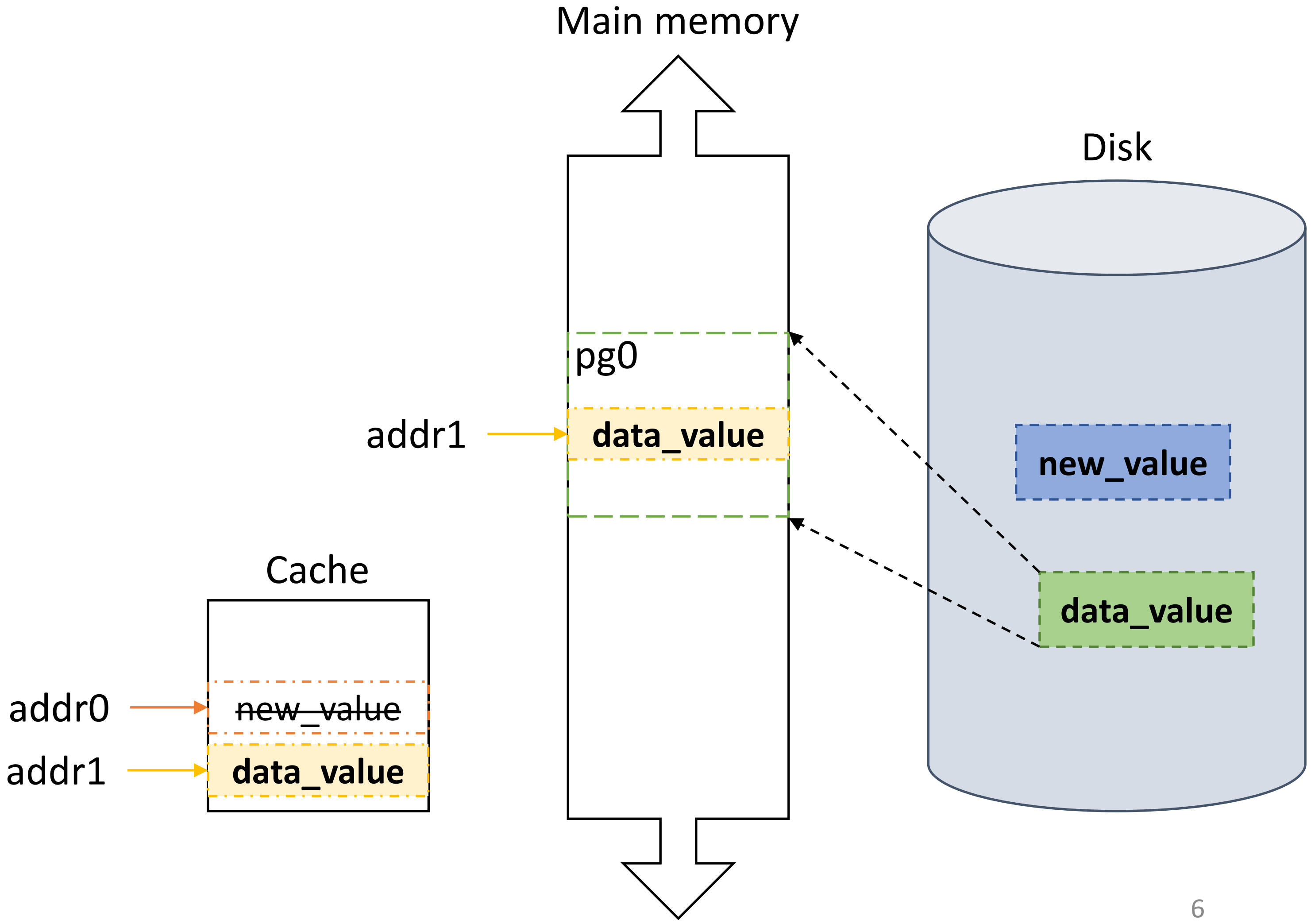
	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	disk
addr1	0	0	1	0	disk



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

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

	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	disk
addr1	0	0	1	0	pg0

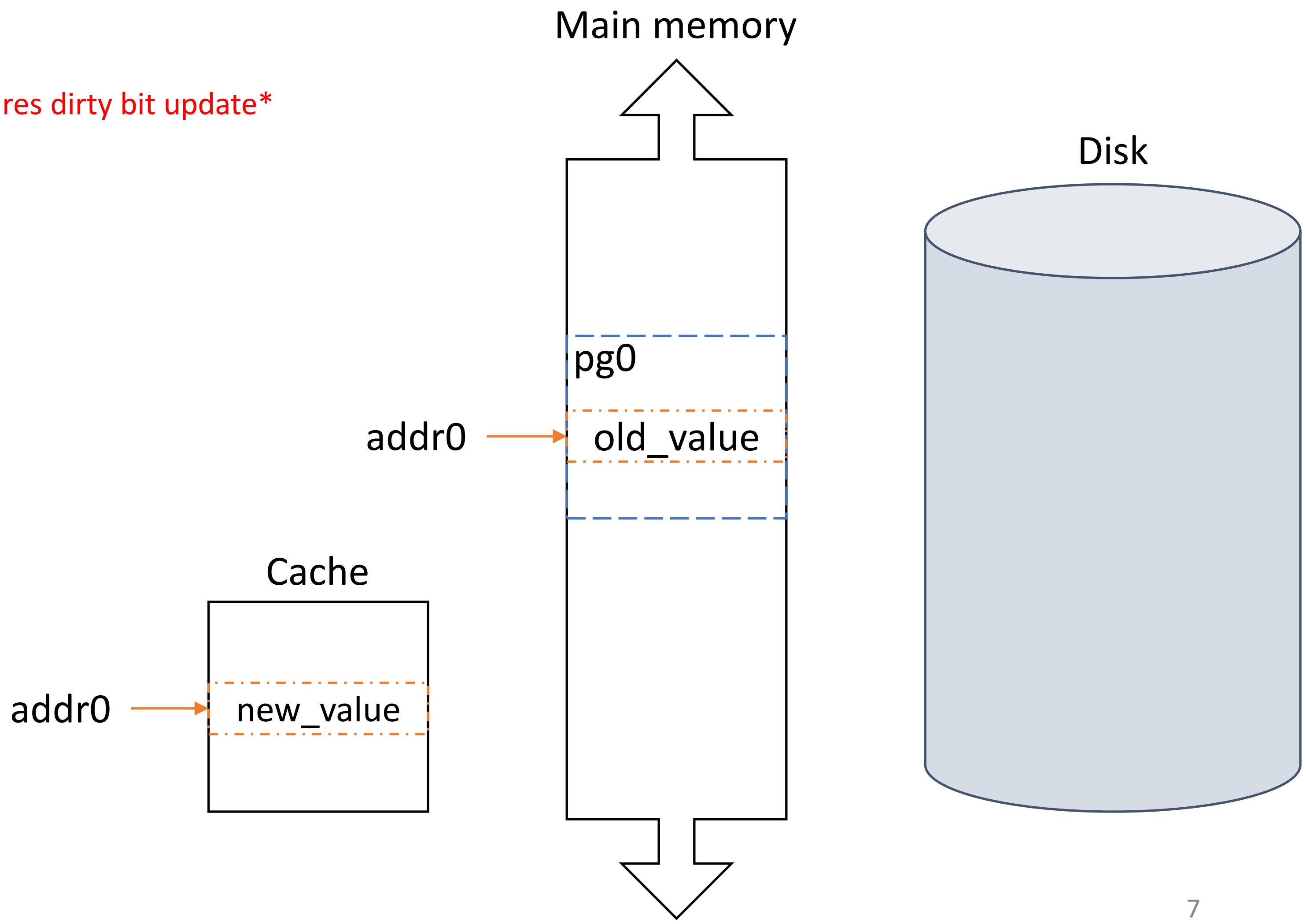


What if dirty bit is not updated before page swapped to disk?

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

requires dirty bit update

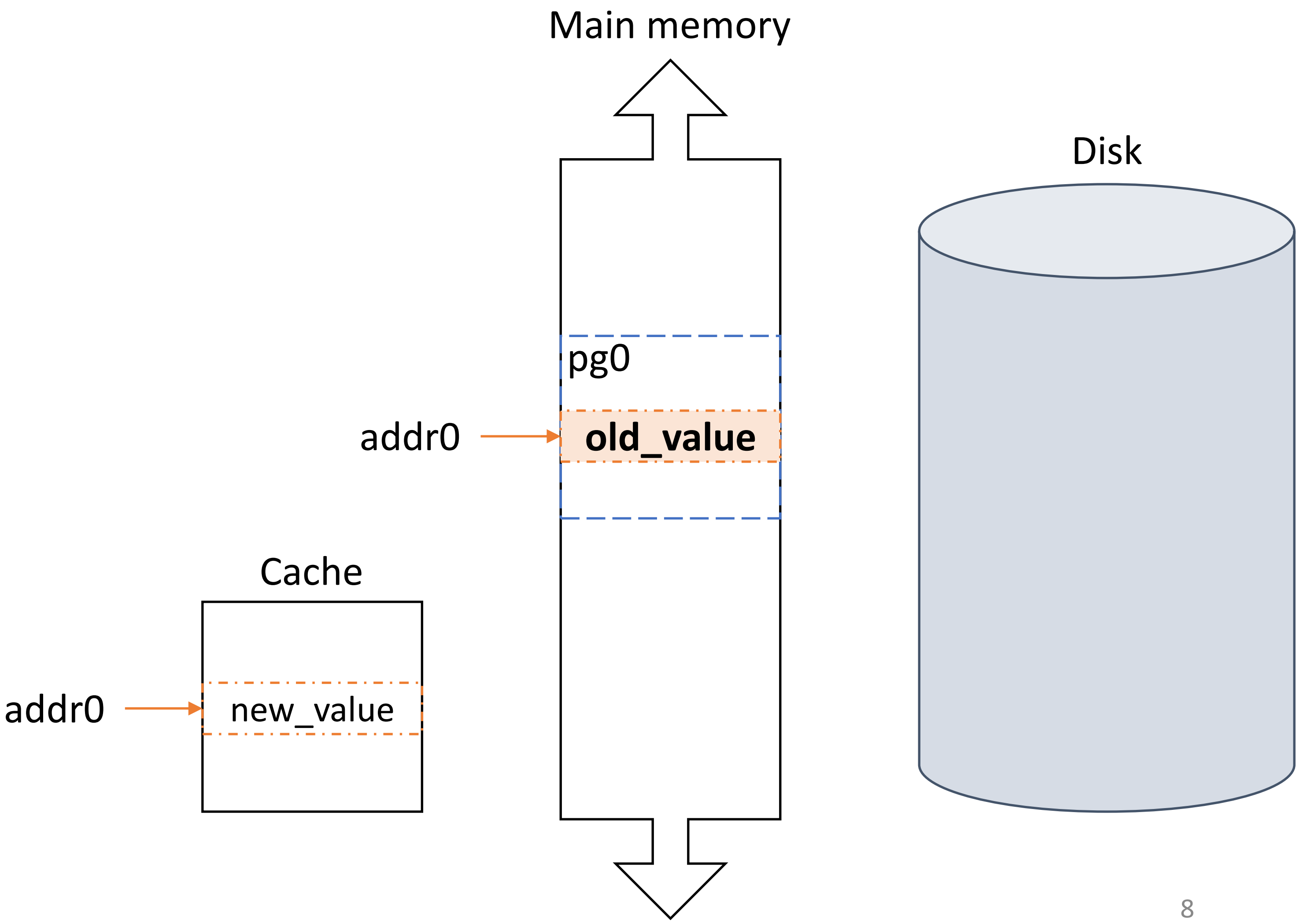
	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	pg0
addr1	0	0	1	0	disk



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

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

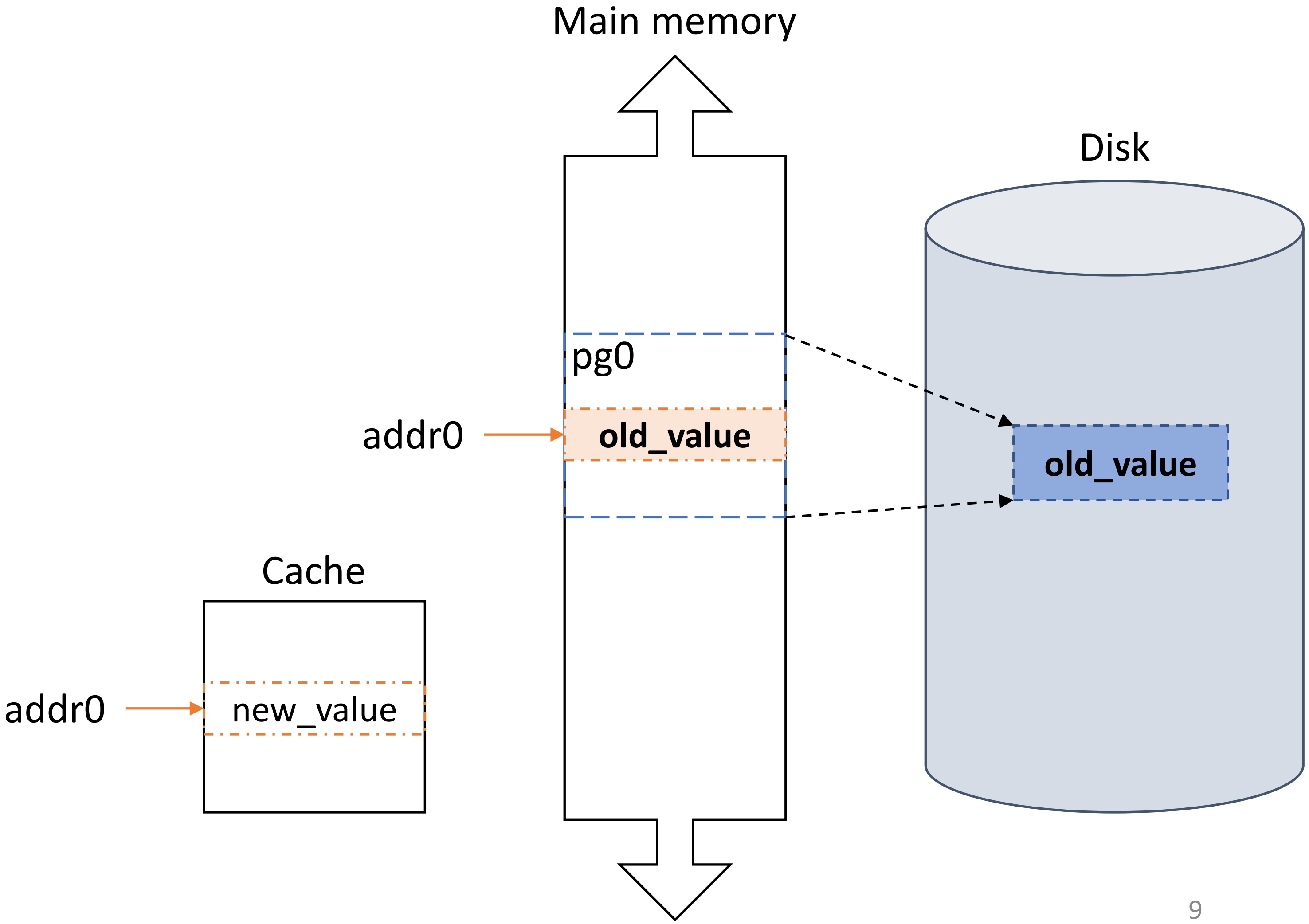
	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	pg0
addr1	0	0	1	0	disk



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

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

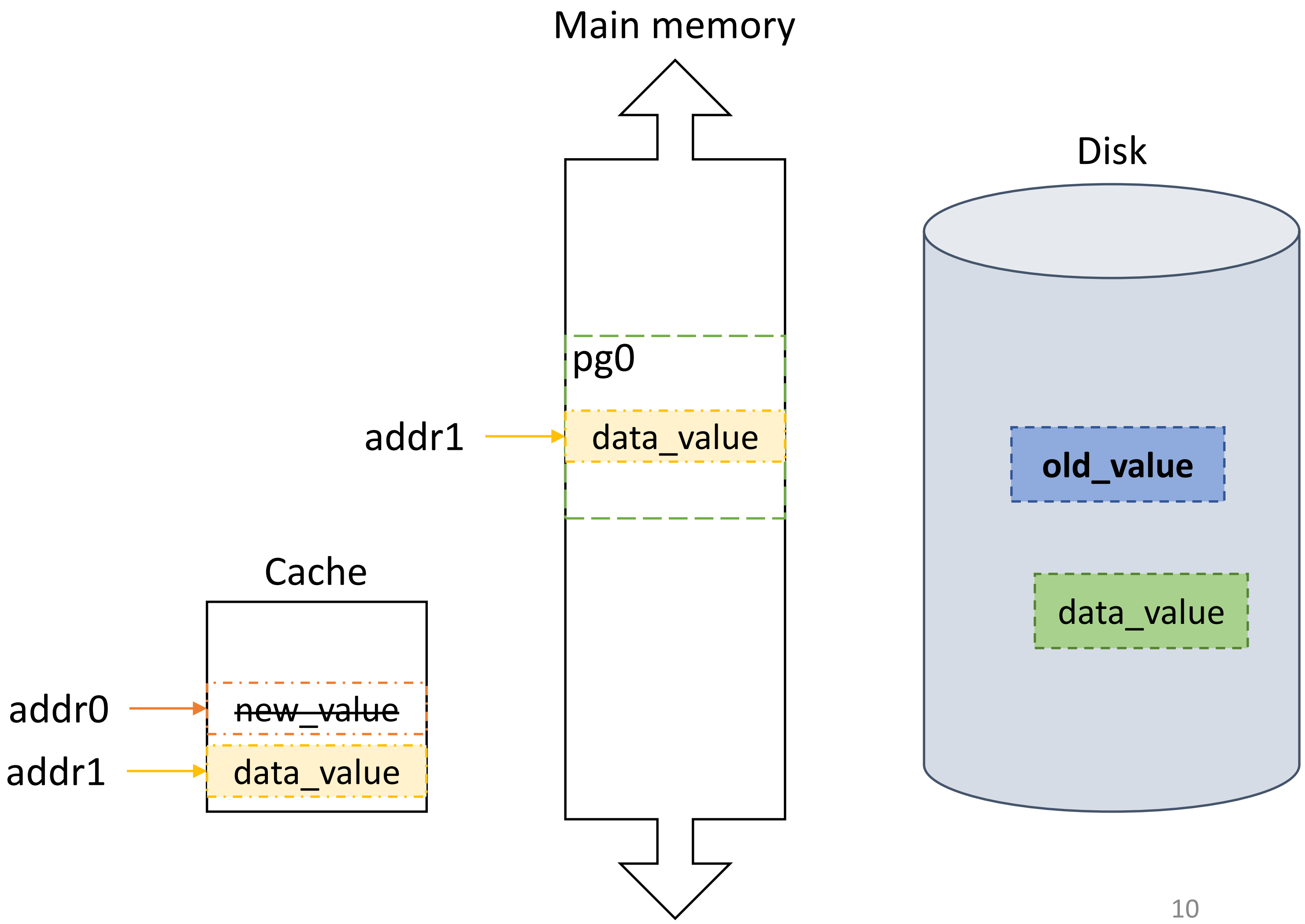
	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	disk
addr1	0	0	1	0	disk



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

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

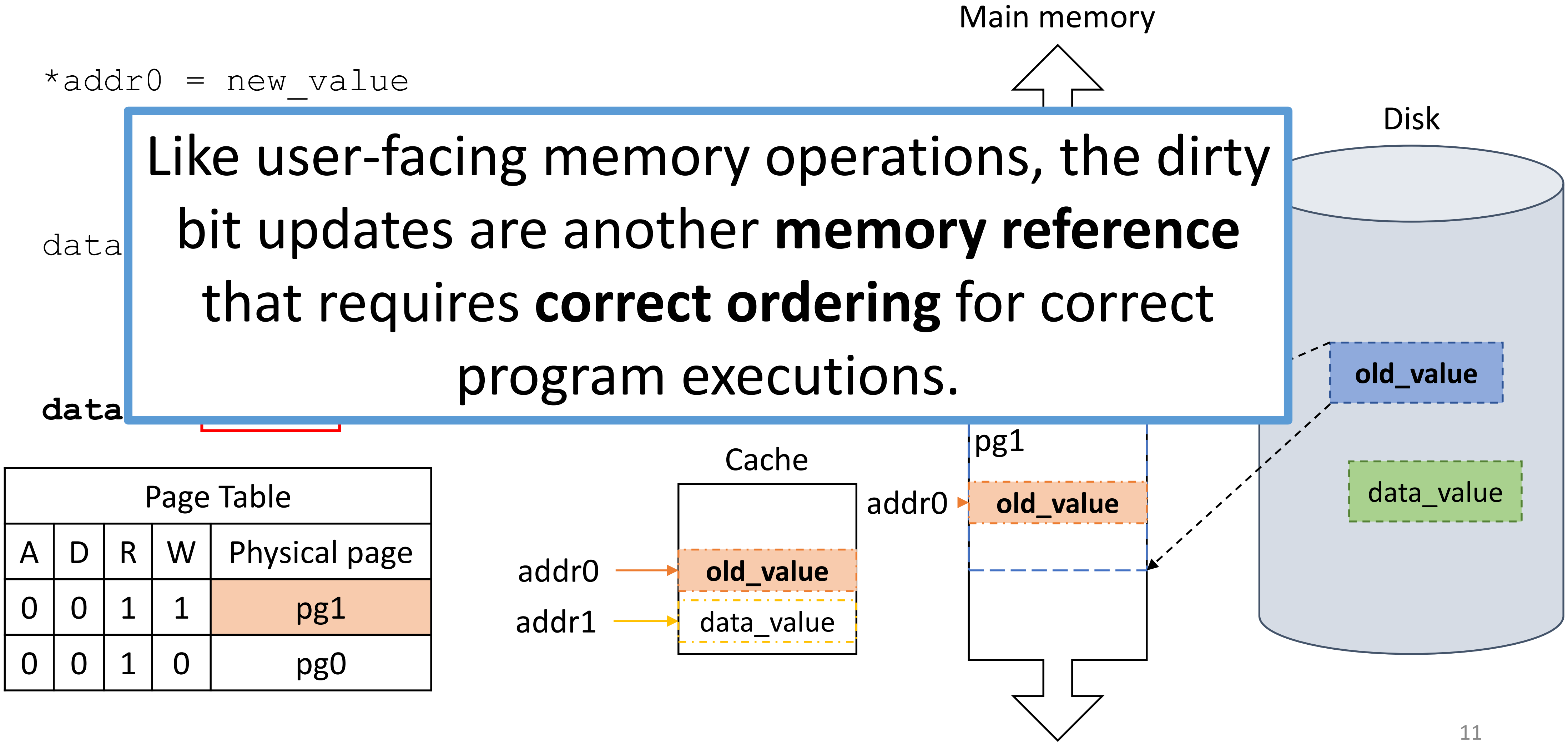
	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	disk
addr1	0	0	1	0	pg0



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

```
*addr0 = new_value
```

Like user-facing memory operations, the dirty bit updates are another **memory reference** that requires **correct ordering** for correct program executions.



	Page Table				
	A	D	R	W	Physical page
addr0	0	0	1	1	pg1
addr1	0	0	1	0	pg0

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

r

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

a. s
b. st flag = 1;
	c. ld r1 = flag;
	if (r1 != 1) goto c;
	d. ld r2 = data;

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

time

Litmus test: small diagnostic

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

memory fences where needed)

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]

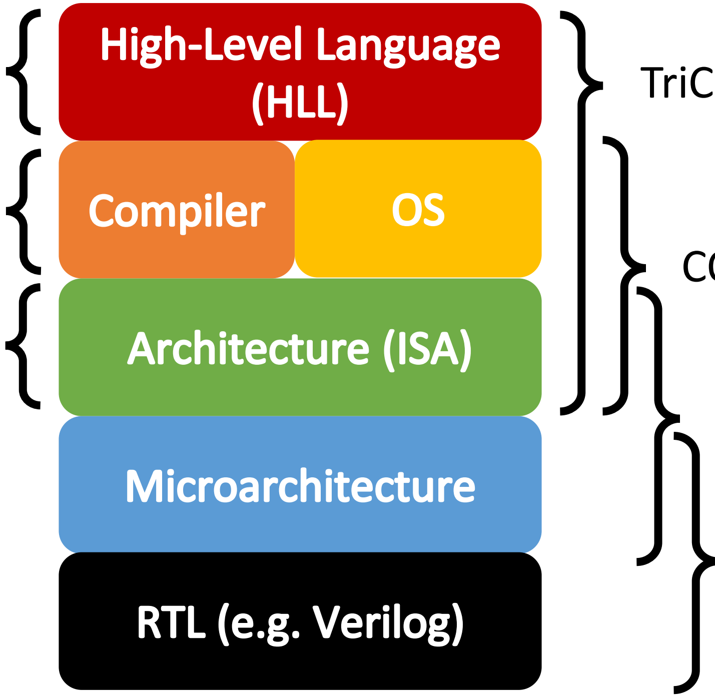
ISA-level MCM tools aiding verification:

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]

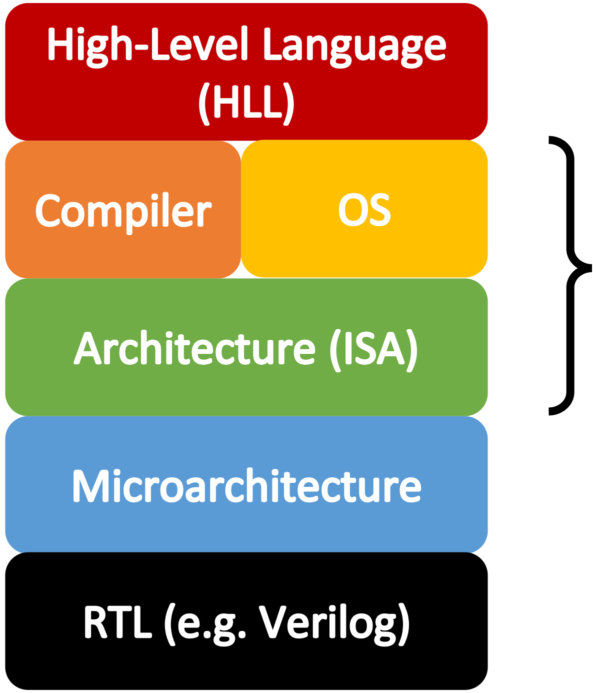
RTLCheck [Manerkar et al., MICRO ‘17]

Prior work

Specification and verification of MTMs

Check tool suite:

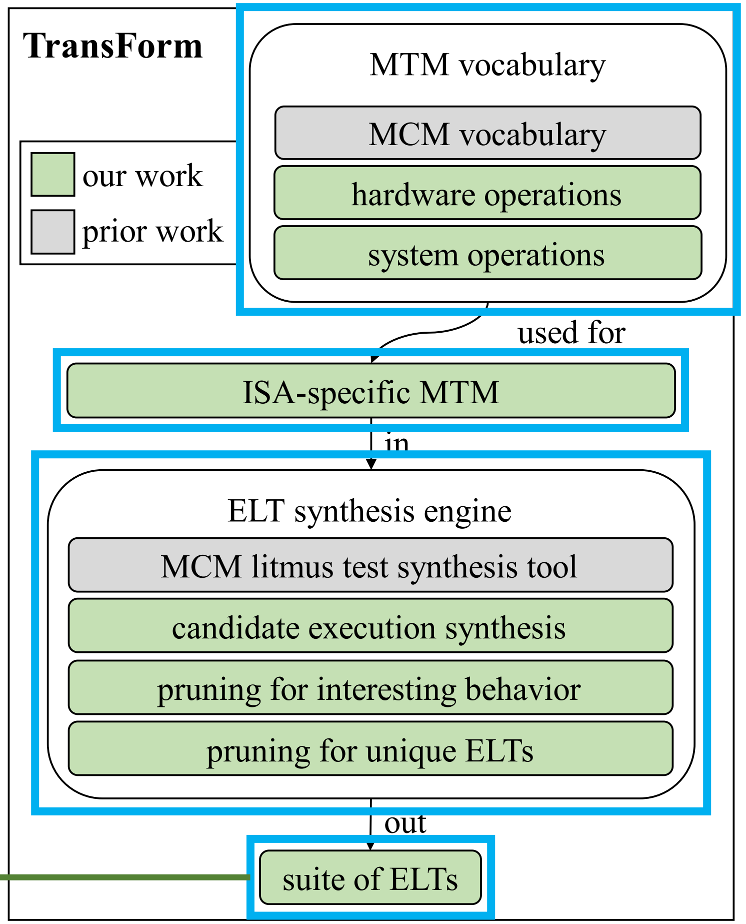
Romanescu et al. introduce the concept of separate MCMs for VAs and PAs [ASPLOS ‘10]



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



TransForm

🟩 our work

⬜ prior work

MTM vocabulary

MCM vocabulary

hardware operations

system operations

used for

ISA-specific MTM

in

ELT synthesis engine

MCM litmus test synthesis tool

candidate execution synthesis

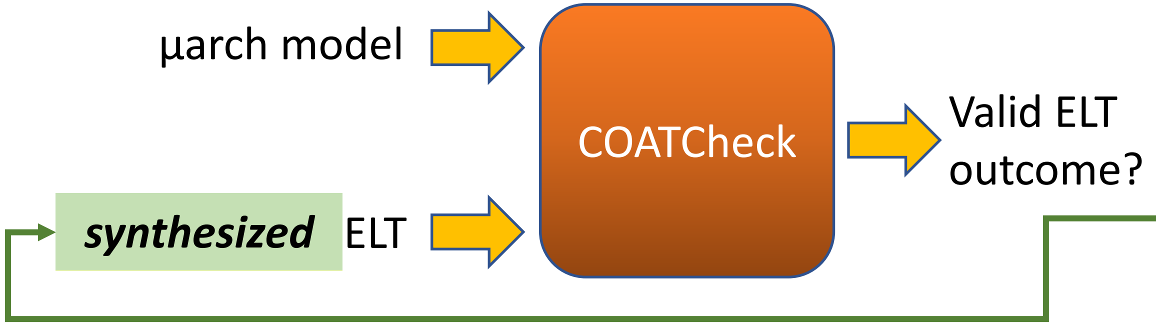
pruning for interesting behavior

pruning for unique ELTs

out

suite of ELTs

Automated!



µarch model

synthesized ELT

COATCheck

Valid ELT outcome?

Allows for verification against *formally specified* MTM

Outline

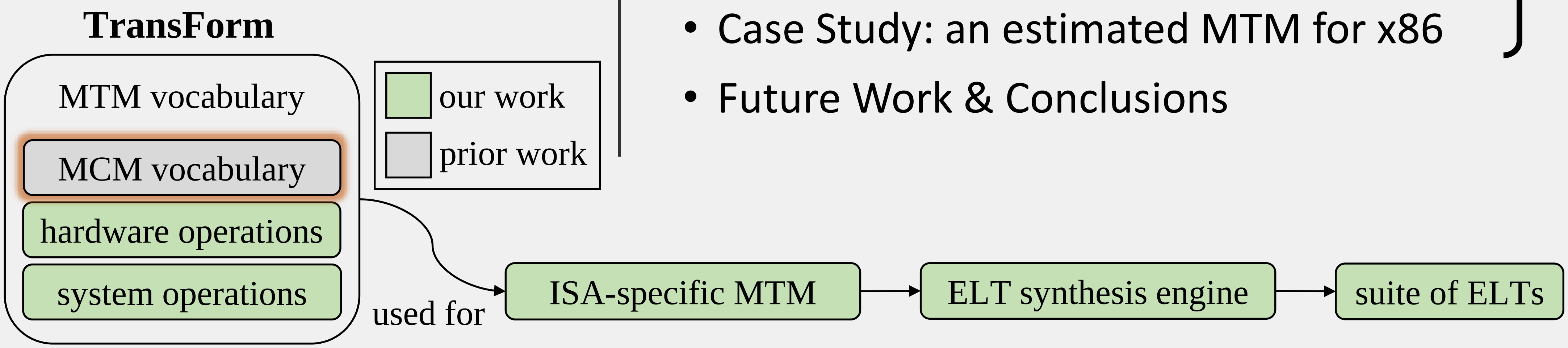
- Background on ISA-level MCM vocabulary
- Background on virtual memory systems

Prior Work

- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: an estimated MTM for x86

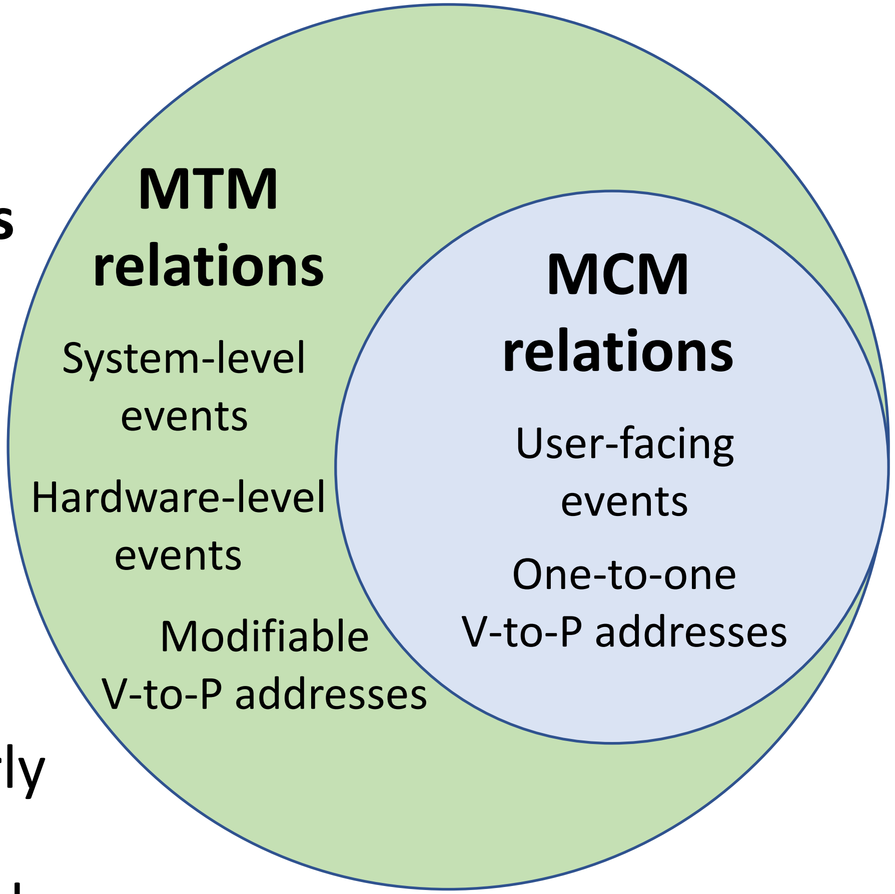
My Work

- Future Work & Conclusions



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

Program

Instructions

Core 0	Core 1

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

, R_2, R_3}

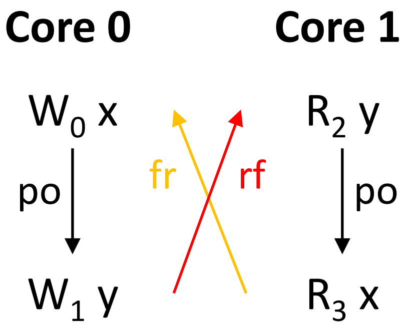
{W_0, W_1, R_2, R_3}

x, W_1 → y, R_2 → y, R_3 → x}

program order (po)

po = {W_0 → W_1, R_2 → R_3}

Graph



Candidate execution

Core 0	Core 1
W_0 x = 1	R_2 y = 1
W_1 y = 1	R_3 x = 0

Communication (com) relations

reads from (rf)

rf = {W_1 → R_2}

coherence order (co)

co = {}

from reads (fr)

fr = {R_3 → W_0}

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

Core 0	Core 1
W_0 x	R_2 y
W_1 y	R_3 x

Candidate execution

Core 0	Core 1
W_0 x = 1	R_2 y = 1
W_1 y = 1	R_3 x = 0

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 = $\{W_1 \rightarrow R_2\}$

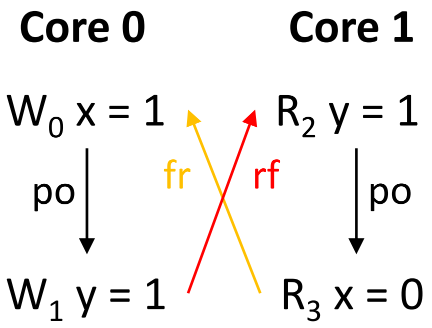
coherence order (co)

co = {}

from reads (fr)

fr = $\{R_3 \rightarrow W_0\}$

Graph



Core 0

Core 1

W_0 x = 1

R_2 y = 1

po

fr

rf

po

W_1 y = 1

R_3 x = 0

Numerical subscripts serve as instruction ID.

Accessed data (outcome) symbolically represented by com relations

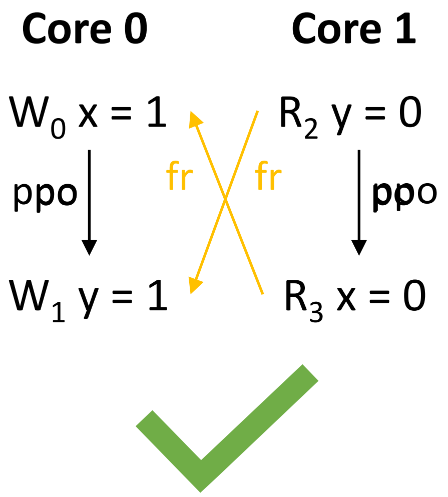
[Shasha & Snir, 1988]
[Alglave et al., 2014]

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:
acyclic(rfe + co + fr + ppo + fence)



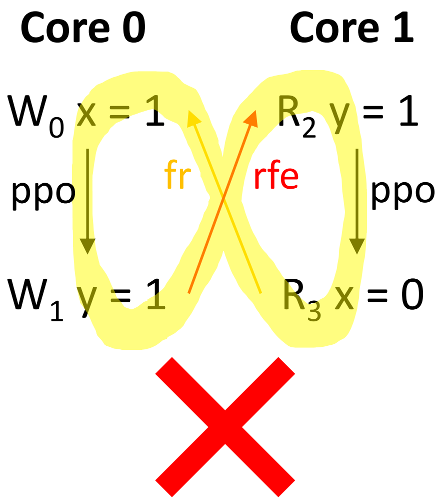
MCM specifications place constraints on permitted execution behaviors

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)

mp ("message passing") litmus test



Outline

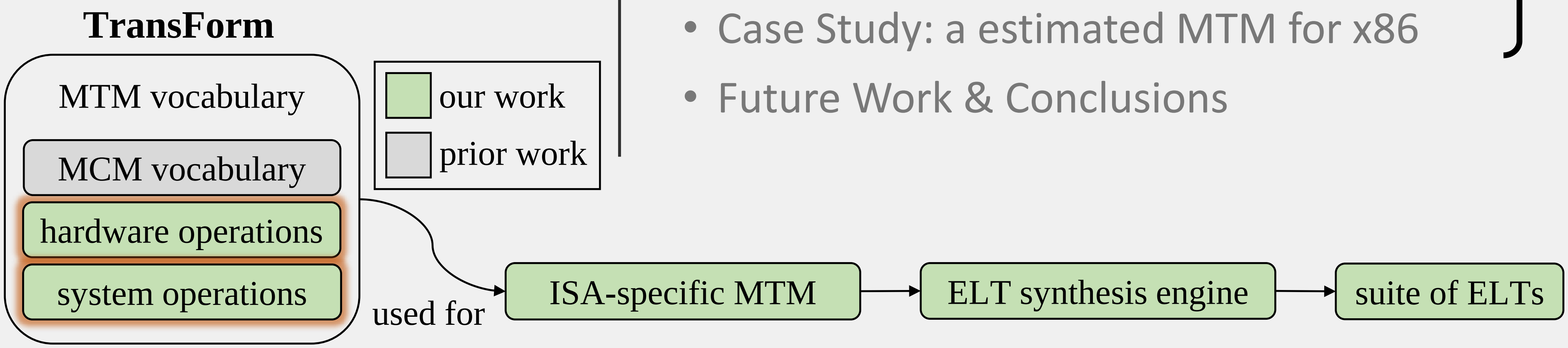
- Background on ISA-level MCM vocabulary
- Background on virtual memory systems

Prior Work

- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: a estimated MTM for x86

My Work

- Future Work & Conclusions



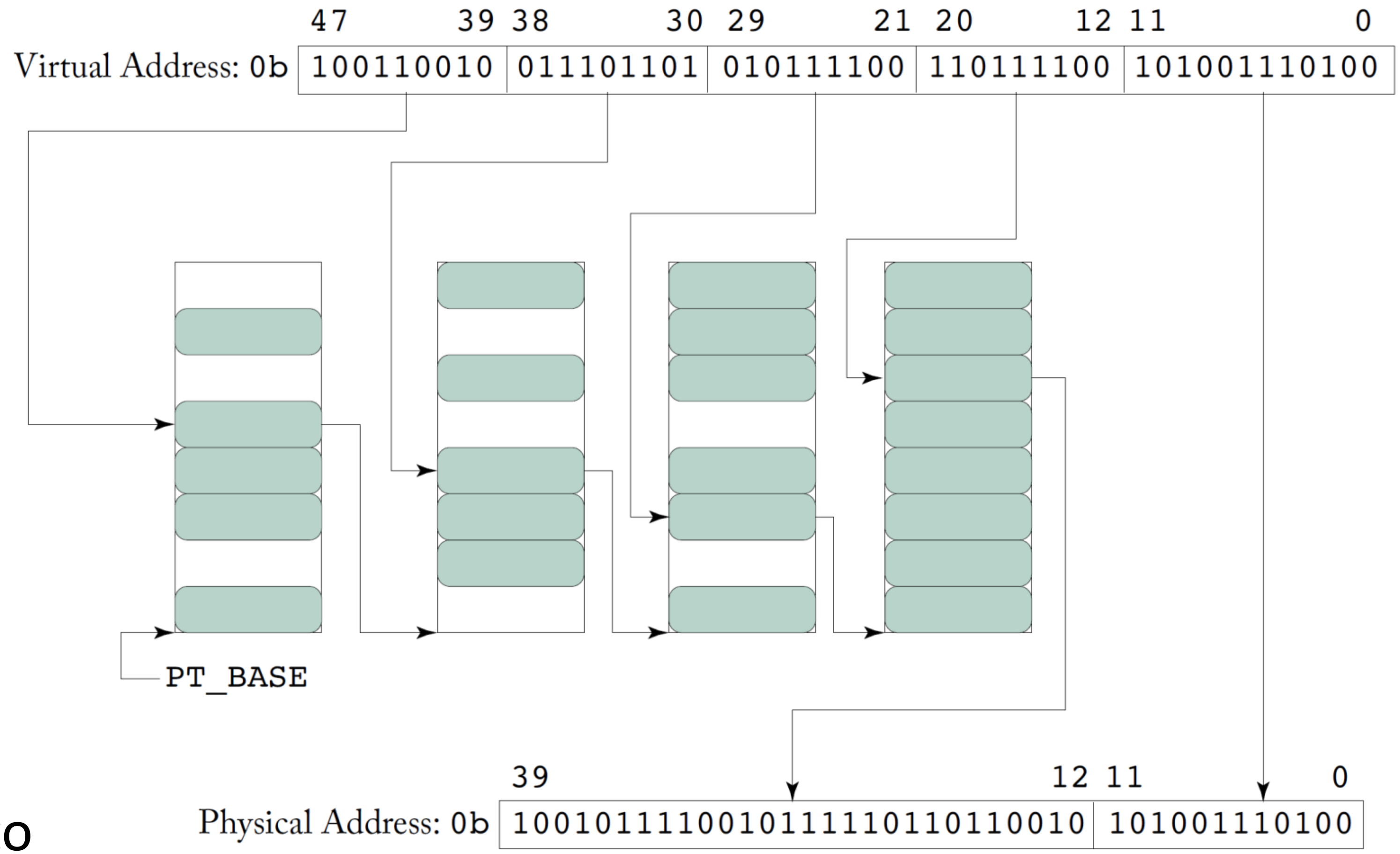
V-to-P address mappings need to be stored and accessed during memory events

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.

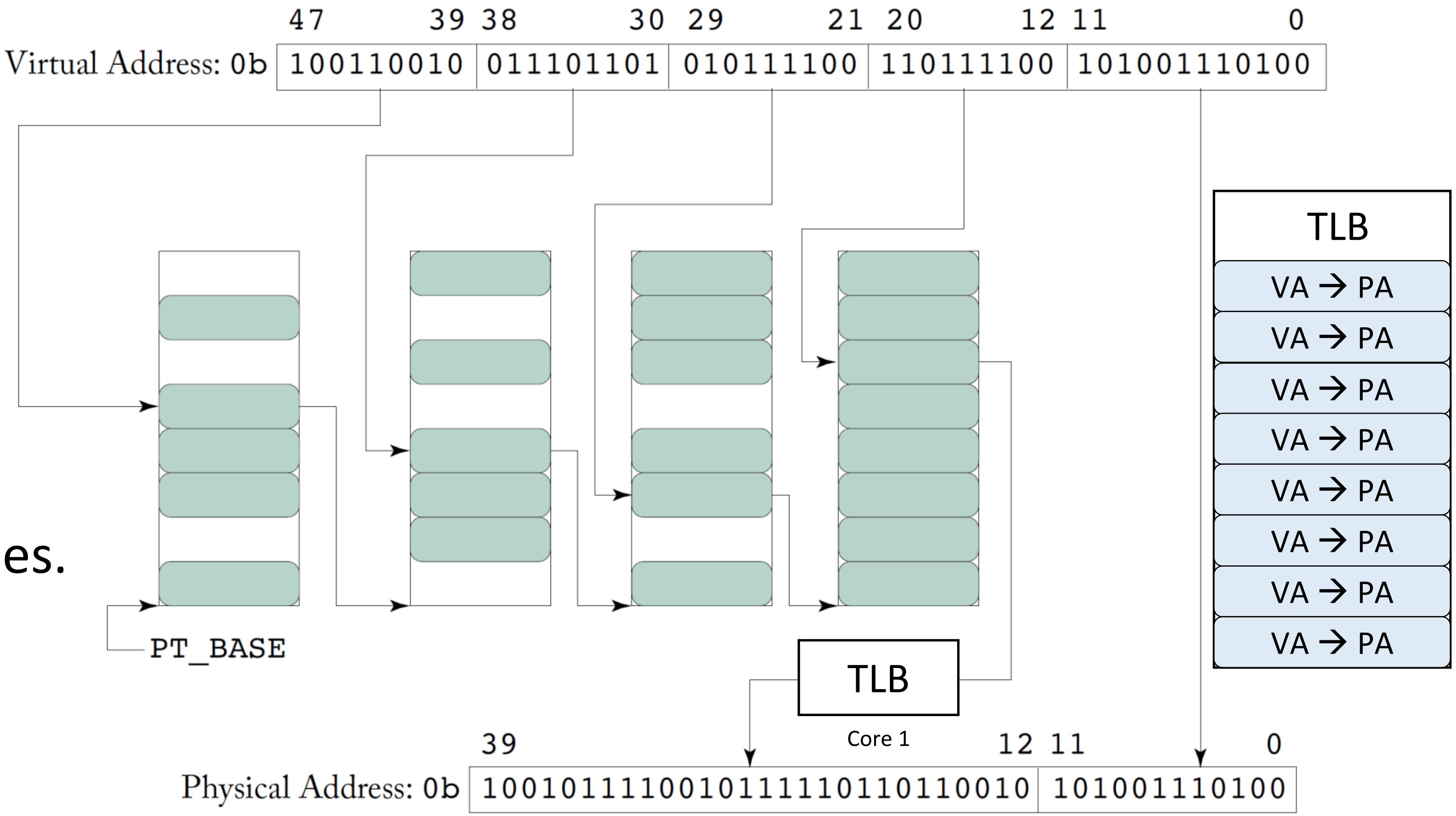


A. Bhattacharjee and D. Lustig, "Architectural and operating system support for virtual memory", *Synthesis Lectures on Computer Architecture*, 2017.

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.

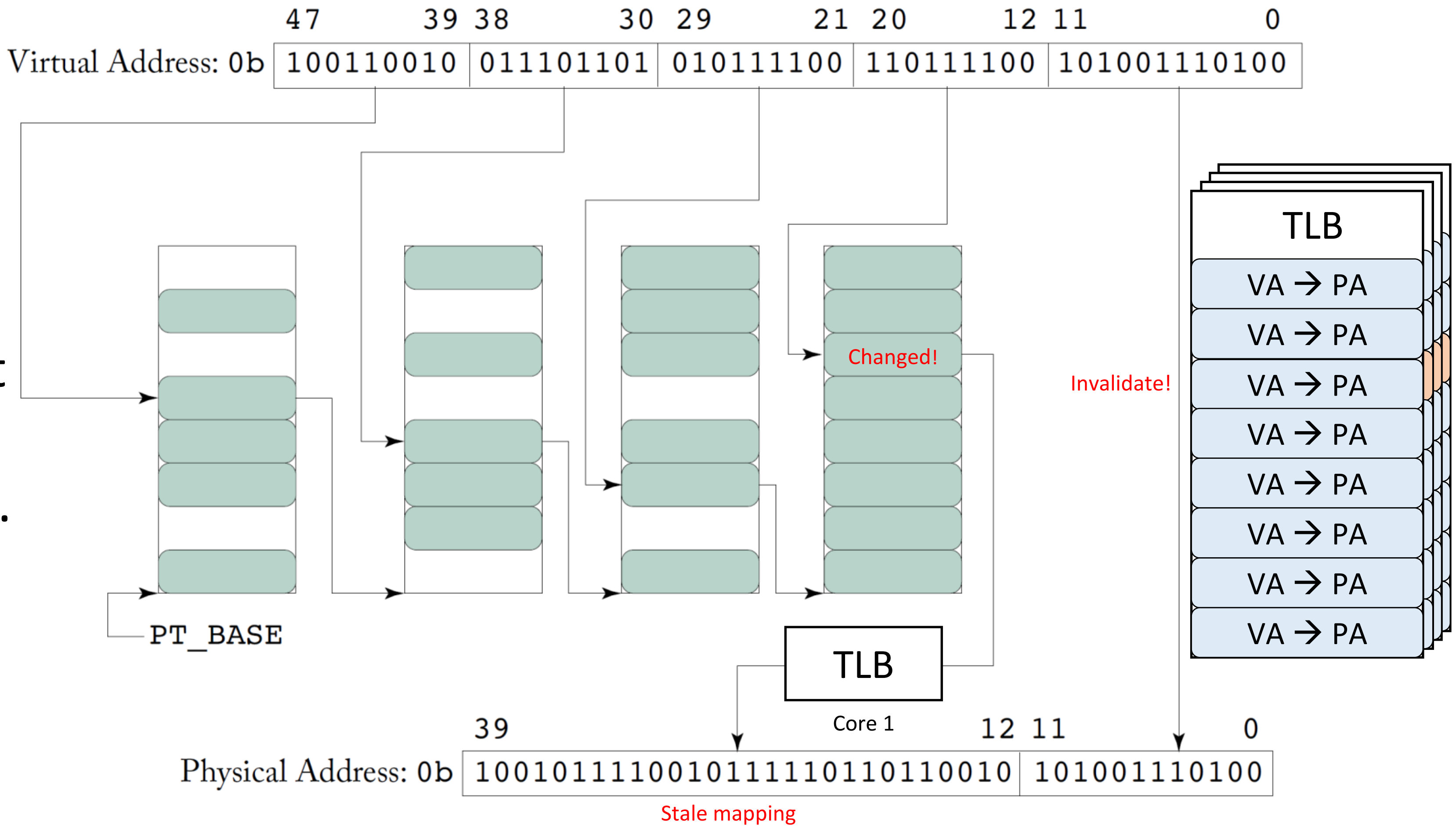
Page mappings cached in TLB to reduce latency of memory accesses.


47 39 38 30 29 21 20 12 11 0
Virtual Address: 0b 100110010 011101101 010111100 110111100 101001110100
PT_BASE
TLB
Core 1
39 12 11 0
Physical Address: 0b 1001011110010111110110110010 101001110100
TLB
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA

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.

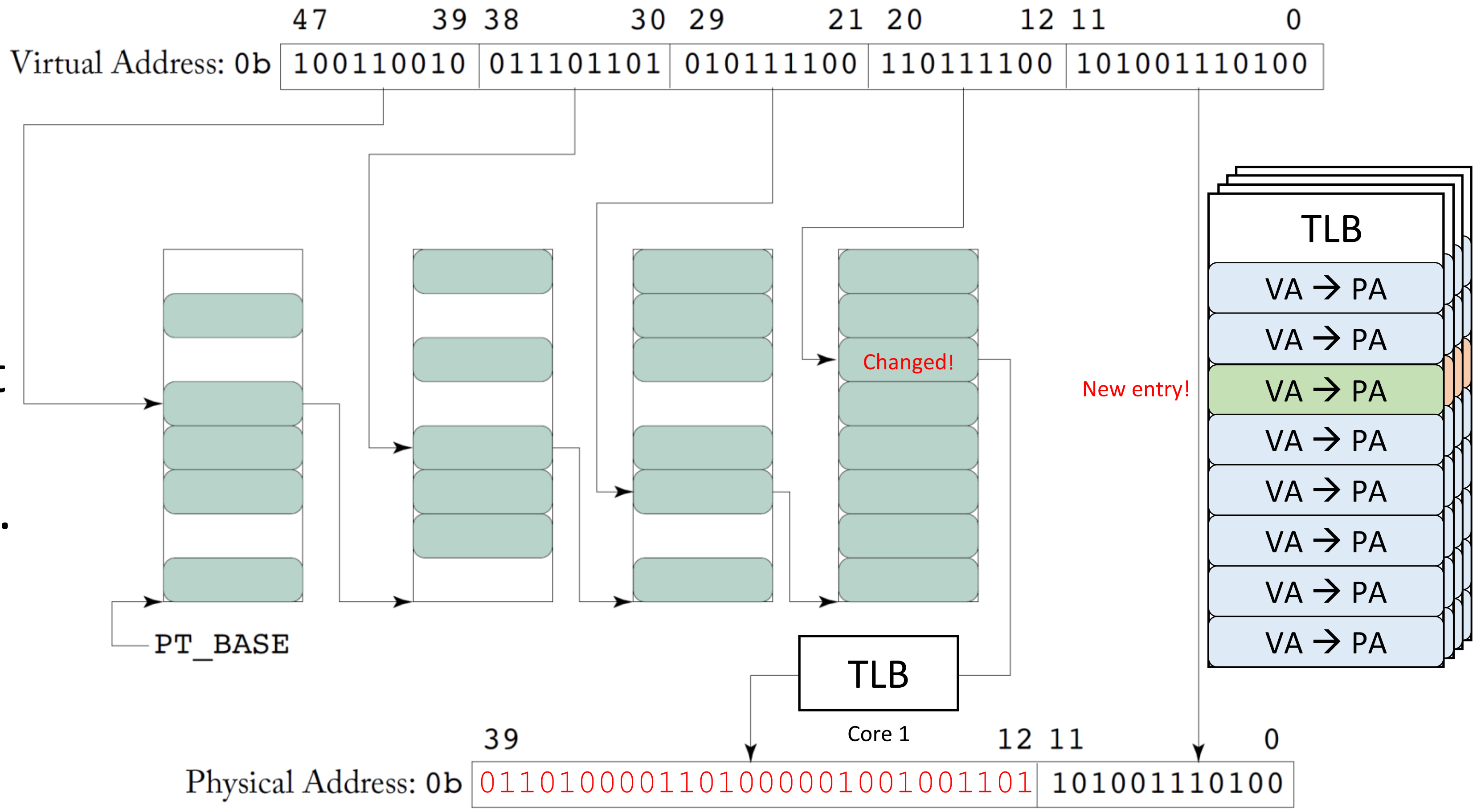


V-to-P address mappings can be changed by OS

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.


47 39 38 30 29 21 20 12 11 0
Virtual Address: 0b 100110010 011101101 010111100 110111100 101001110100
Changed!
New entry!
TLB
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
VA → PA
PT_BASE
TLB
Core 1
39 12 11 0
Physical Address: 0b 0110100001101000001001001101 101001110100

Virtual memory events TransForm needs to support

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

Outline

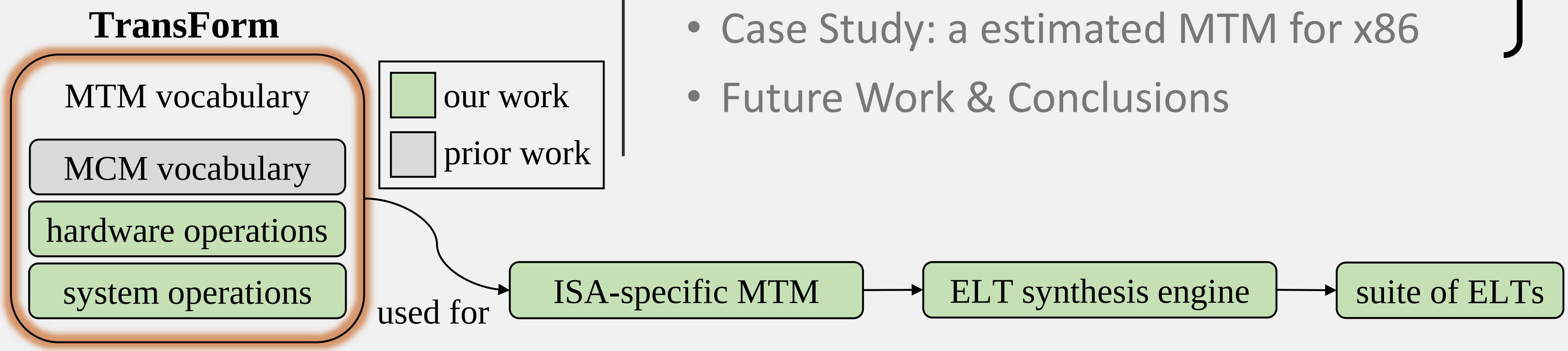
- Background on ISA-level MCM vocabulary
- Background on virtual memory systems

Prior Work

- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: a estimated MTM for x86

My Work

- Future Work & Conclusions



TransForm

MTM vocabulary

MCM vocabulary

hardware operations

system operations

🟩 our work

▨ prior work

used for

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

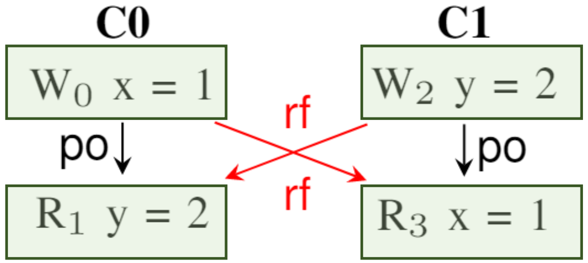
MTM Vocabulary: Hardware-level events

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

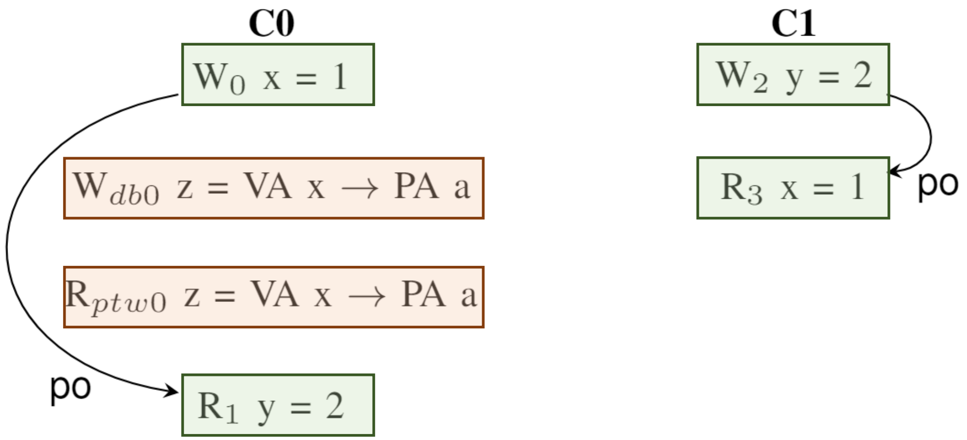
TransForm supports **page table walks (PTW)** and **dirty bit updates**

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**

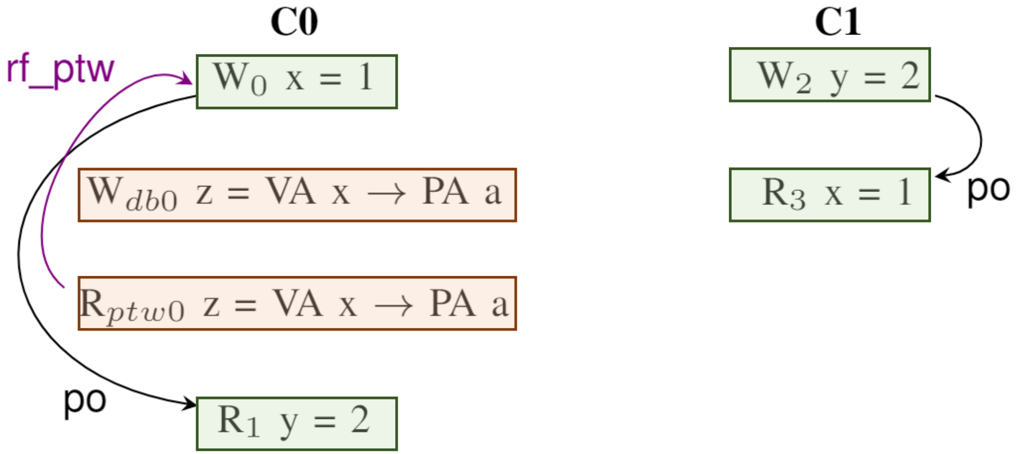
Ghost instructions

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**

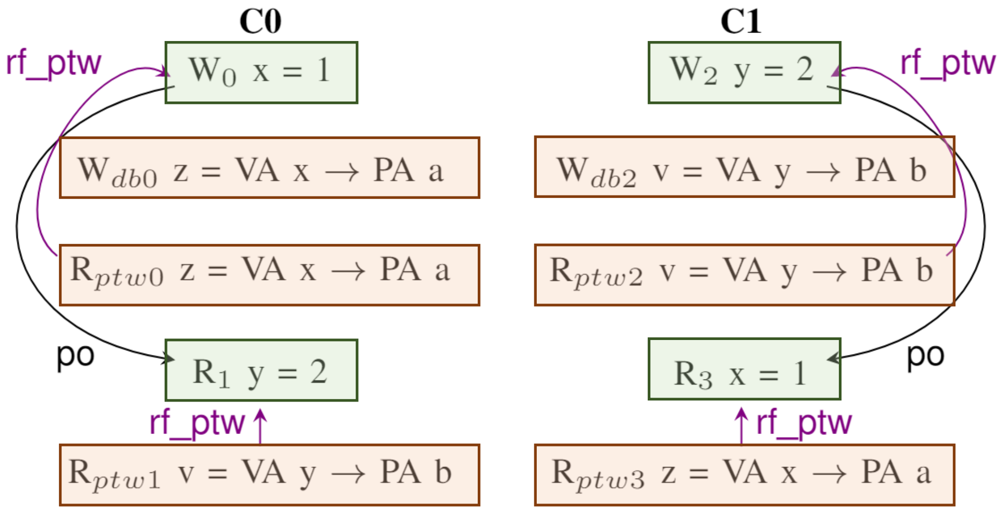
Ghost instructions

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


C0
C1
rf_ptw
W_0 x = 1
W_2 y = 2
rf_ptw
W_{db0} z = VA x → PA a
W_{db2} v = VA y → PA b
R_{ptw0} z = VA x → PA a
R_{ptw2} v = VA y → PA b
po
R_1 y = 2
R_3 x = 1
po
rf_ptw
rf_ptw
R_{ptw1} v = VA y → PA b
R_{ptw3} z = VA x → PA a

MTM Vocabulary: Hardware-level events

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

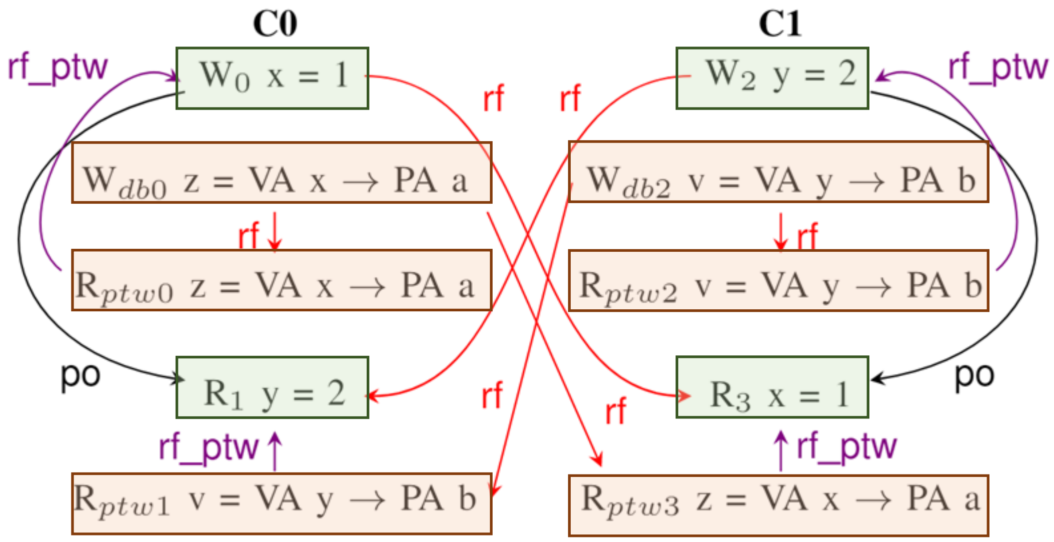
Ghost instructions

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

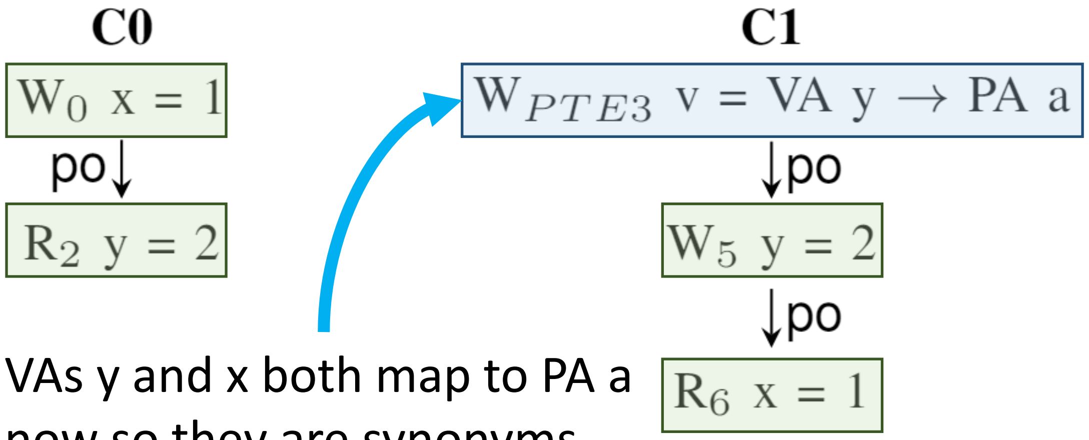

C0
C1
W_0 x = 1
W_{db0} z = VA x → PA a
R_{ptw0} z = VA x → PA a
R_1 y = 2
R_{ptw1} v = VA y → PA b
W_2 y = 2
W_{db2} v = VA y → PA b
R_{ptw2} v = VA y → PA b
R_3 x = 1
R_{ptw3} z = VA x → PA a
rf_ptw
rf_ptw
rf_ptw
rf_ptw
rf
rf
rf
rf
rf
rf
po
po

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



VAs y and x both map to PA a now so they are synonyms.

MTM Vocabulary: System-level events

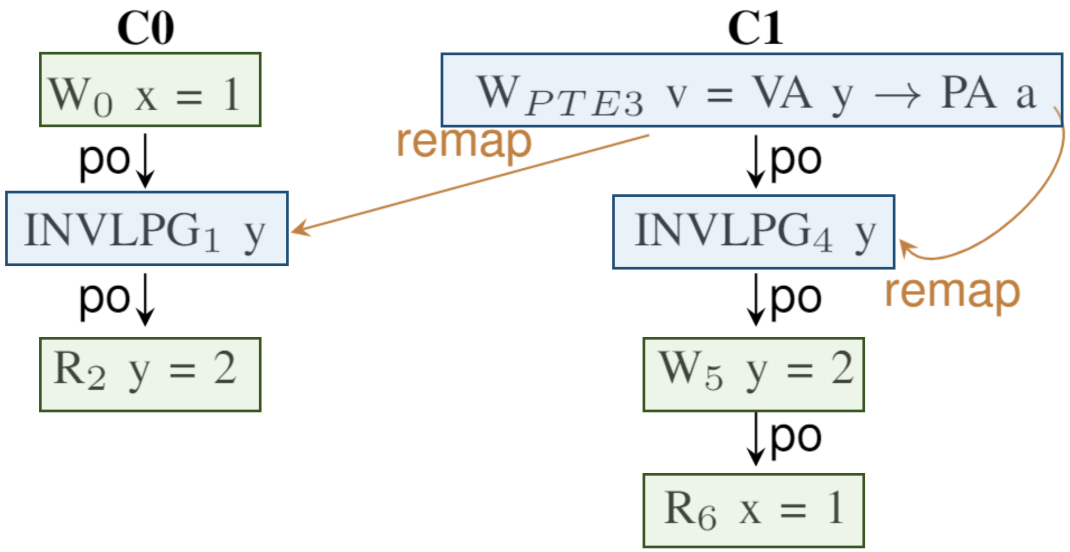
TransForm supports **address remappings via PTE Writes** and **TLB entry invalidations**

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



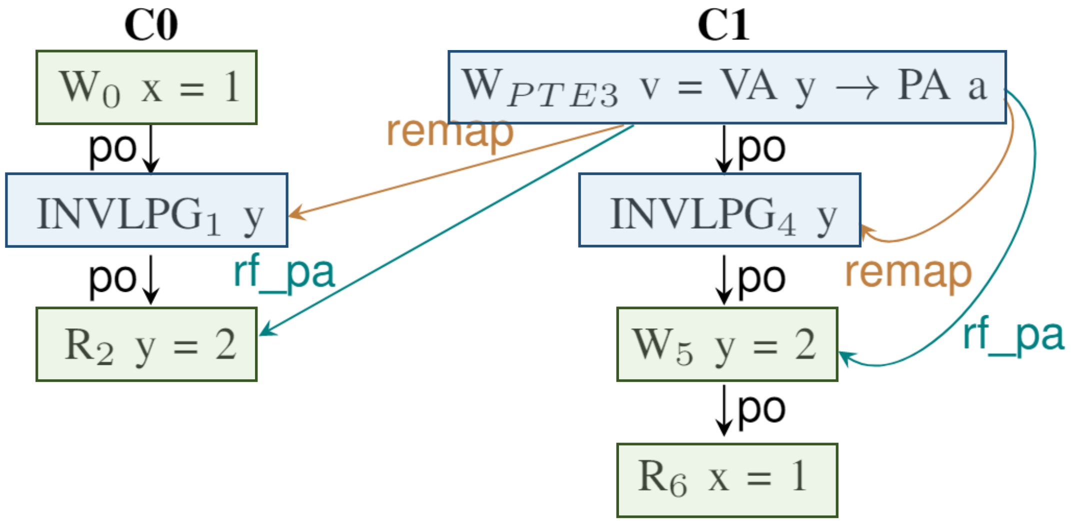
MTM Vocabulary: System-level events

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

co_pa and **fr_va** follow similarly

MTM Vocabulary: System-level events

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

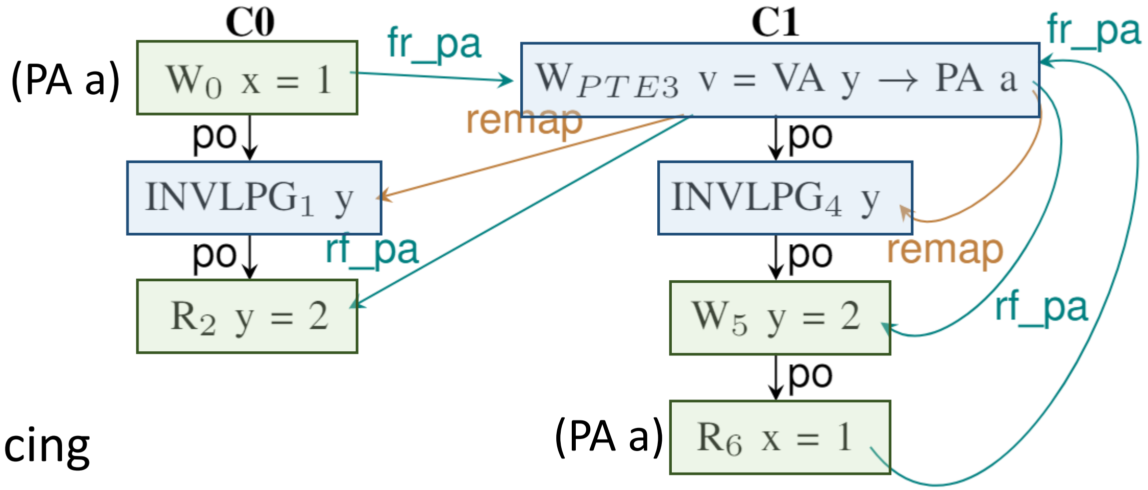
Support instructions

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

INVLPG: invalidates TLB entry (named after x86 instruction)

remap – relates PTE Writes to invoked INVLPGs

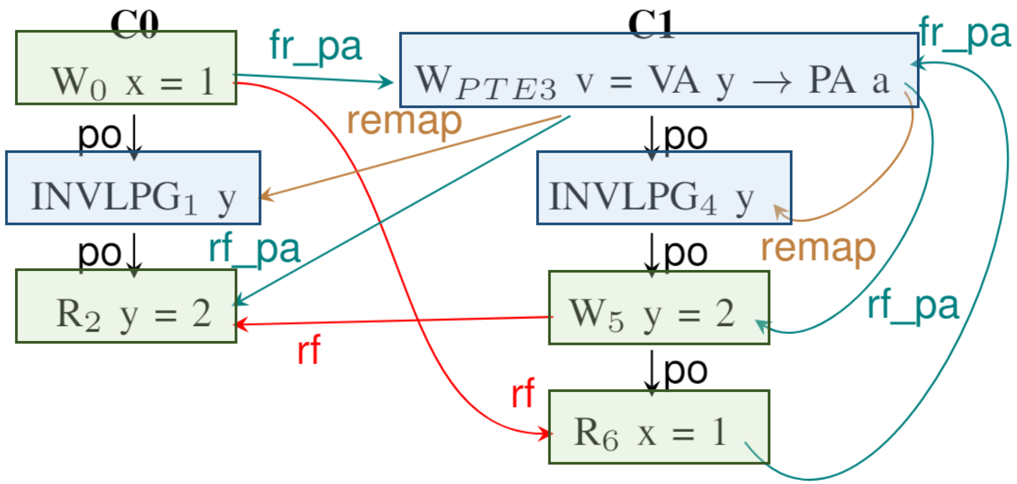
rf_pa – relates PTE Write for VA v → PA p to user-facing MemoryEvents accessing PA p via VA v
fr_pa – relates user-facing MemoryEvents accessing PA p via VA v to PTE Writes for VA v' → PA p
co_pa and **fr_va** follow similarly



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.

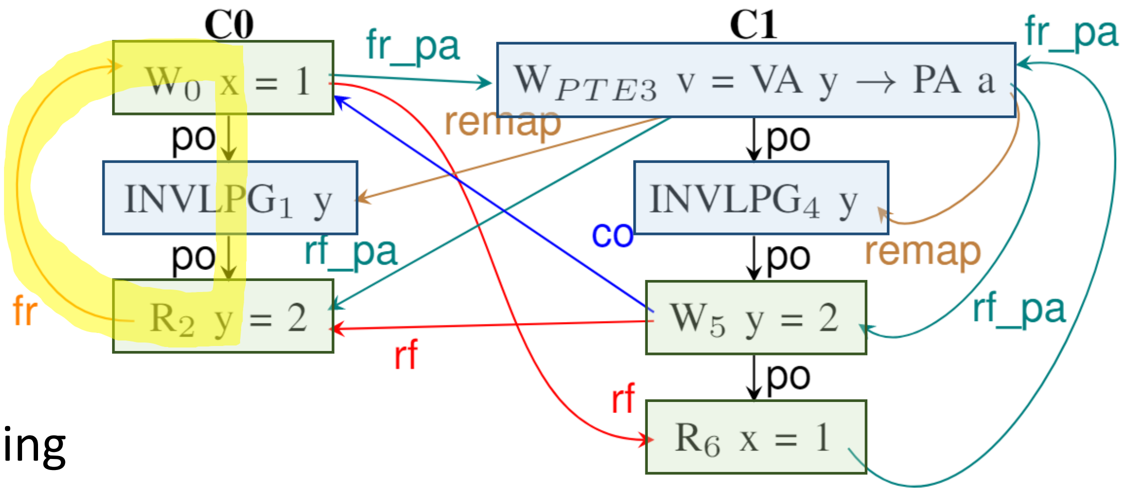


rf_pa – relates PTE Write for VA v → PA p to user-facing MemoryEvents accessing PA p via VA v
fr_pa – relates user-facing MemoryEvents accessing PA p via VA v to PTE Writes for VA v' → PA p
co_pa and **fr_va** follow similarly

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.



Forbidden!

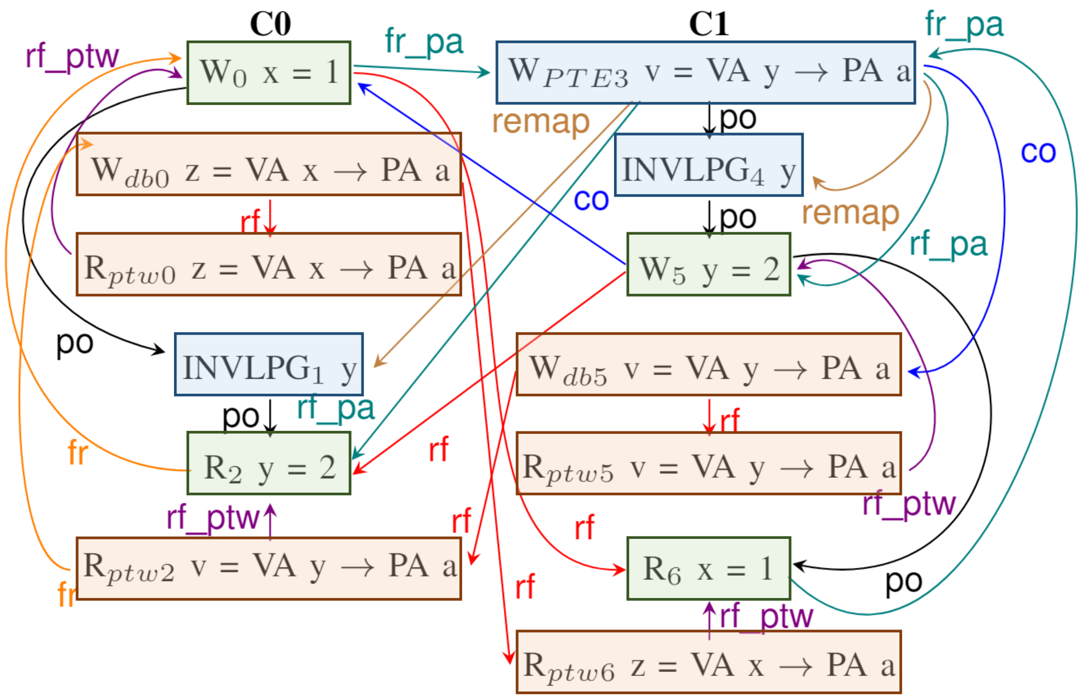
rf_pa – relates PTE Write for VA v → PA p to user-facing MemoryEvents accessing PA p via VA v

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: Putting it all together

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.



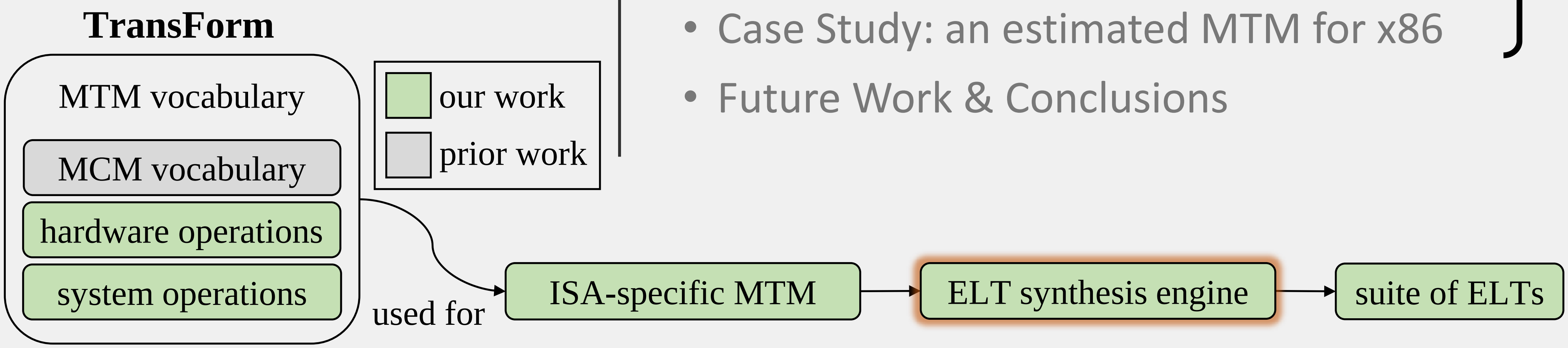
C0
C1
W_0 x = 1
W_{PTE3} v = VA y → PA a
W_{db0} z = VA x → PA a
R_{ptw0} z = VA x → PA a
$INVLPG_1$ y
R_2 y = 2
R_{ptw2} v = VA y → PA a
$INVLPG_4$ y
W_5 y = 2
W_{db5} v = VA y → PA a
R_{ptw5} v = VA y → PA a
R_6 x = 1
R_{ptw6} z = VA x → PA a
rf_ptw
fr_pa
fr_pa
remap
po
co
po
remap
co
rf_pa
rf
po
po
rf_pa
rf
fr
rf_ptw
rf
rf
rf_ptw
fr
po
rf
rf_ptw

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

Prior Work

My Work



TransForm

MTM vocabulary

MCM vocabulary

hardware operations

system operations

used for

ISA-specific MTM

ELT synthesis engine

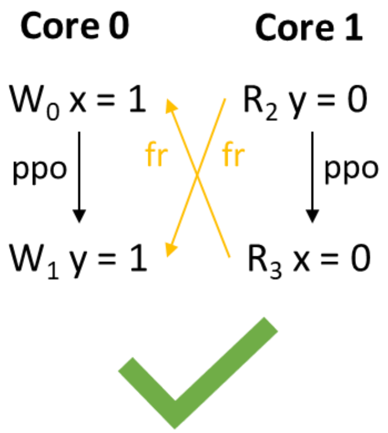
suite of ELTs

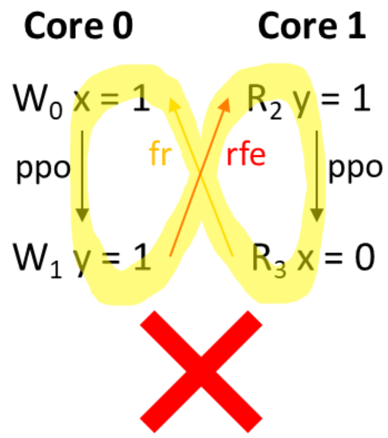
🟩 our work

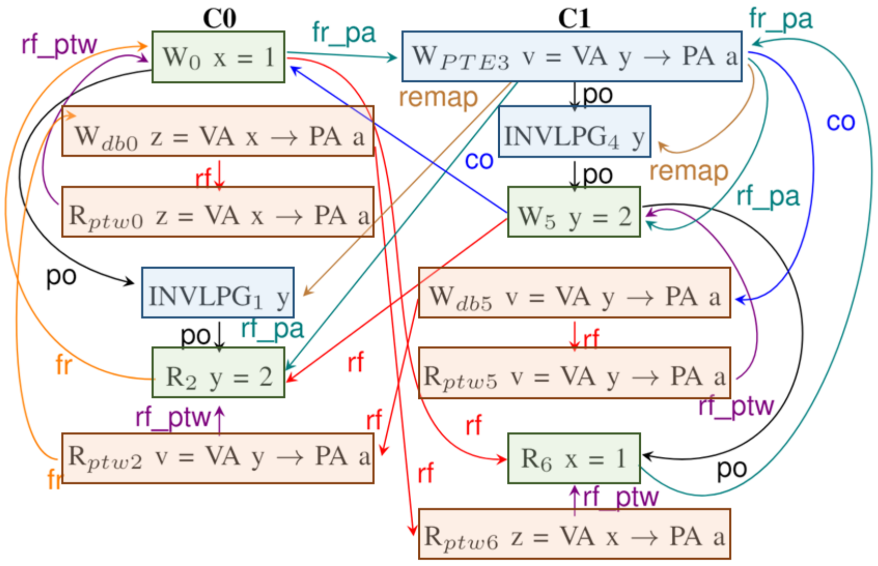
⬜ prior work

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

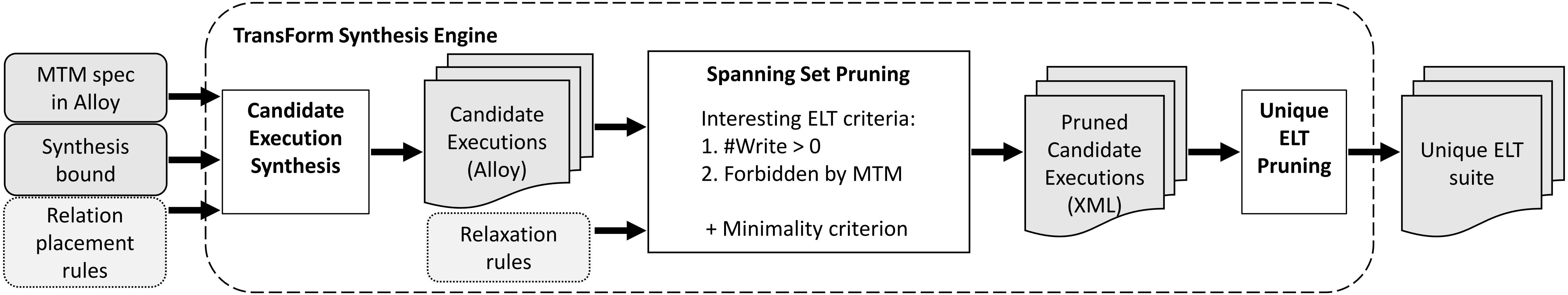
- **Litmus tests**: small diagnostic programs for validating MCM behaviors
    - Executions and their outcomes can be deemed **permitted** or **forbidden** by MCM specification
- **Enhanced litmus tests (ELTs)**: litmus tests enhanced with system- and hardware-level events that facilitate address translation





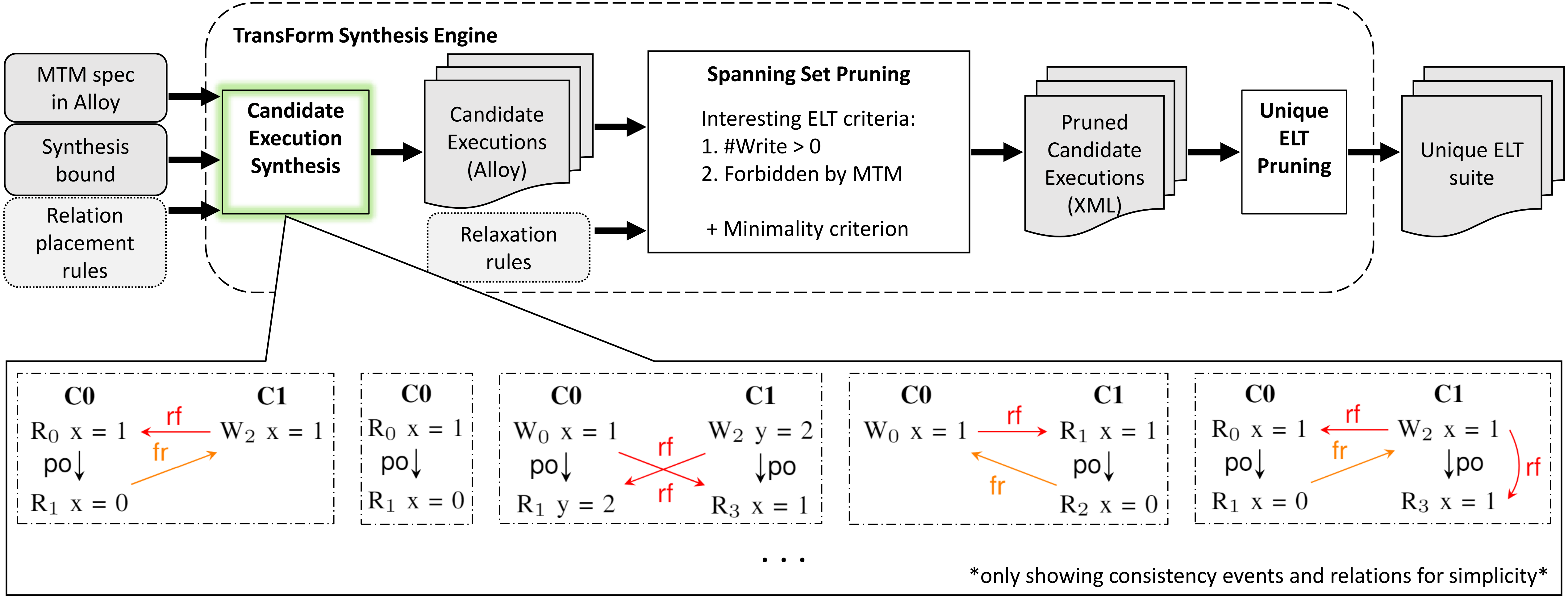

C0
C1
W_0 x = 1
W_{db0} z = VA x → PA a
R_{ptw0} z = VA x → PA a
INVLPG$_1$ y
R_2 y = 2
R_{ptw2} v = VA y → PA a
W_{PTE3} v = VA y → PA a
INVLPG$_4$ y
W_5 y = 2
W_{db5} v = VA y → PA a
R_{ptw5} v = VA y → PA a
R_6 x = 1
R_{ptw6} z = VA x → PA a
rf_ptw
fr_pa
remap
po
co
rf
rf_pa
fr
co
rf_ptw
fr_pa
remap
rf_pa
rf
rf_ptw
po
rf
rf
fr

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

C0 **C1**
R_0 x = 1 ← W_2 x = 1
R_1 x = 0

C0
R_0 x = 1
R_1 x = 0

C0 **C1**
W_0 x = 1 W_2 y = 2
R_1 y = 2 R_3 x = 1

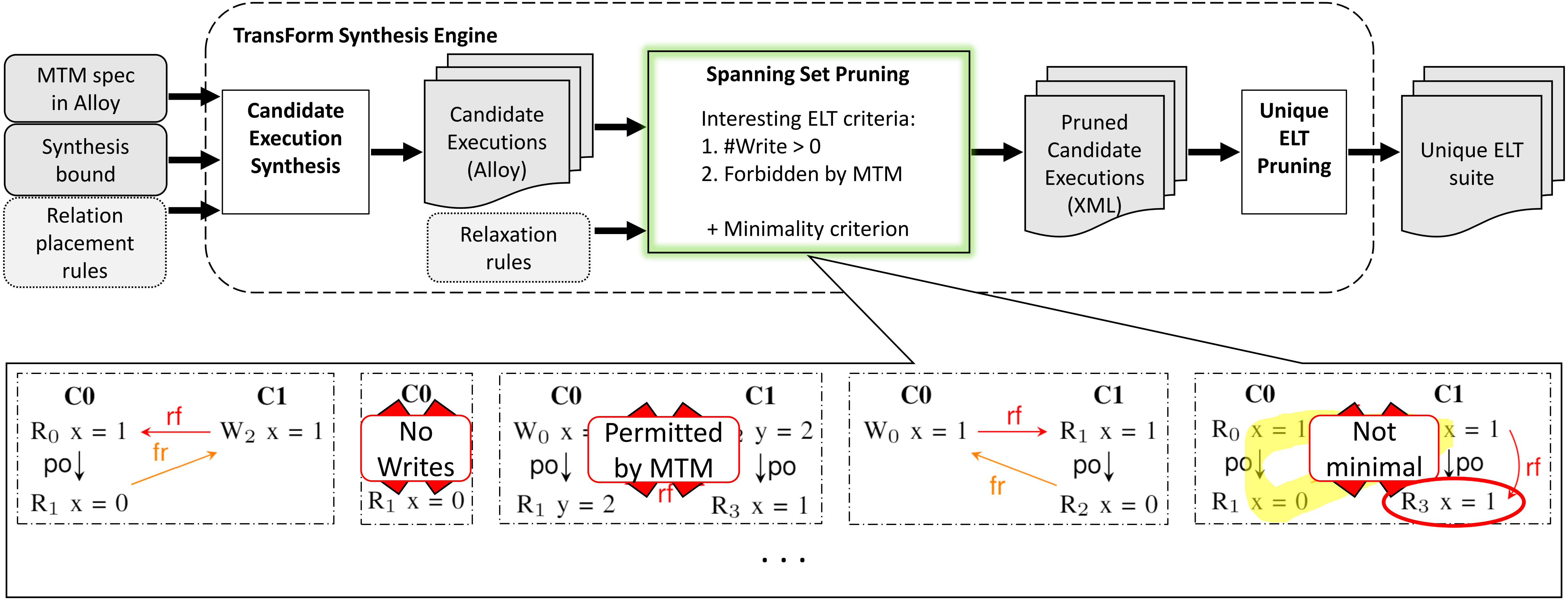
C0 **C1**
W_0 x = 1 → R_1 x = 1
R_2 x = 0

C0 **C1**
R_0 x = 1 ← W_2 x = 1
R_1 x = 0 R_3 x = 1

. . .

only showing consistency events and relations for simplicity

Candidate executions are pruned for interesting ELT behaviors and checked for minimality



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

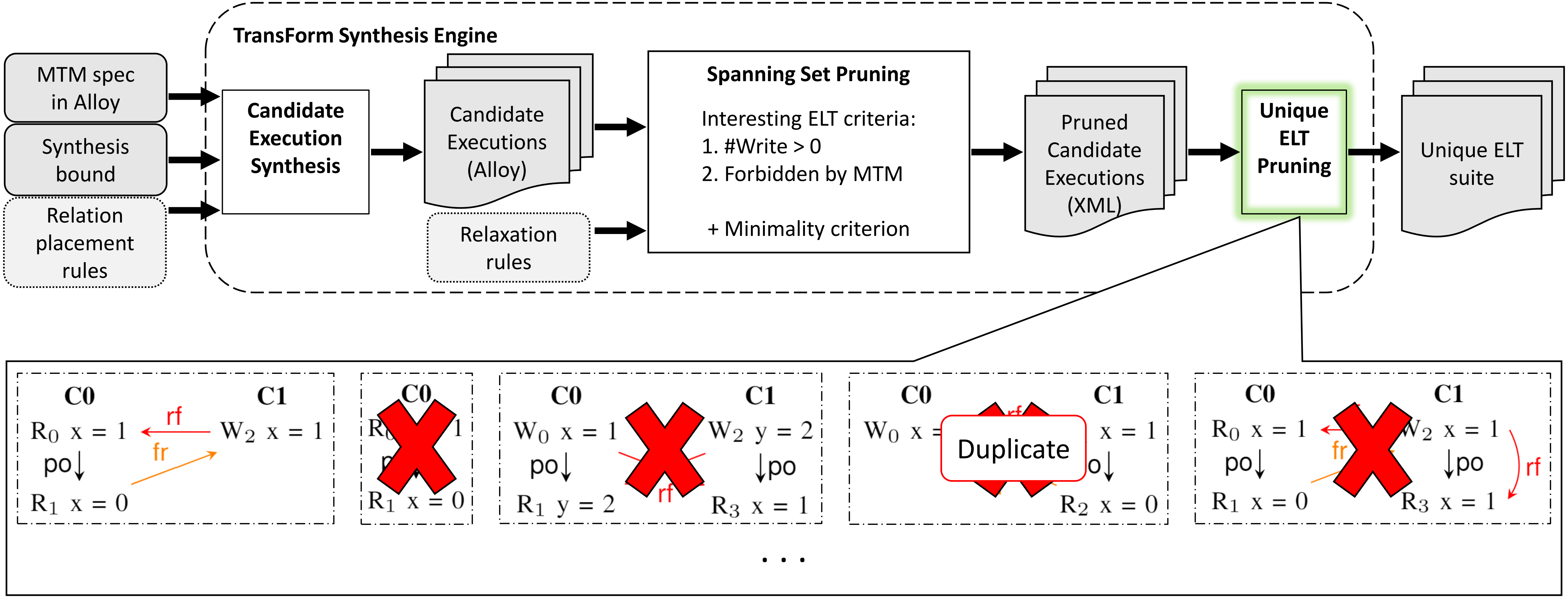
+ Minimality criterion

Pruned Candidate Executions (XML)

Unique ELT Pruning

Unique ELT suite

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



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

Outline

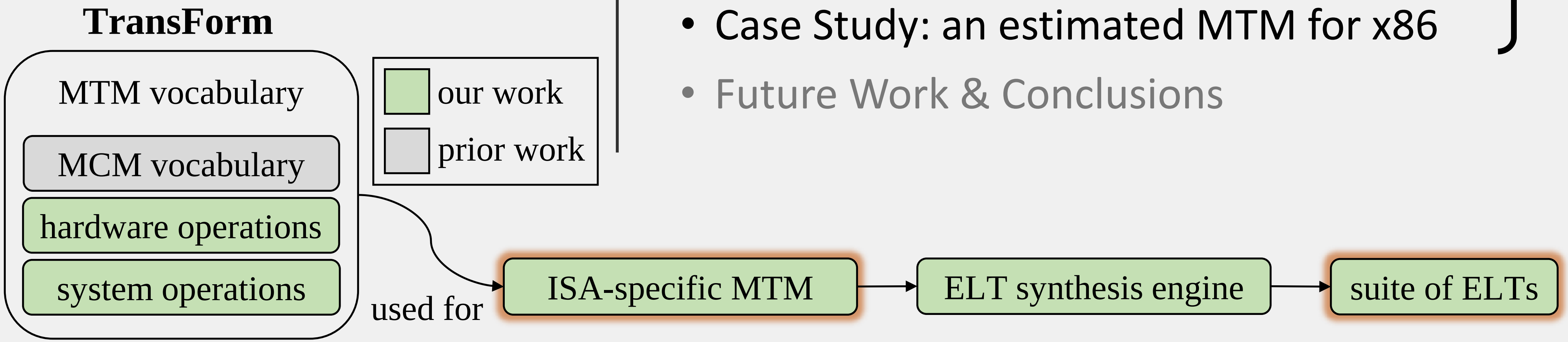
- Background on ISA-level MCM vocabulary
- Background on virtual memory systems

Prior Work

- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: an estimated MTM for x86

My Work

- Future Work & Conclusions



TransForm

MTM vocabulary

MCM vocabulary

hardware operations

system operations

🟩 our work

▨ prior work

used for

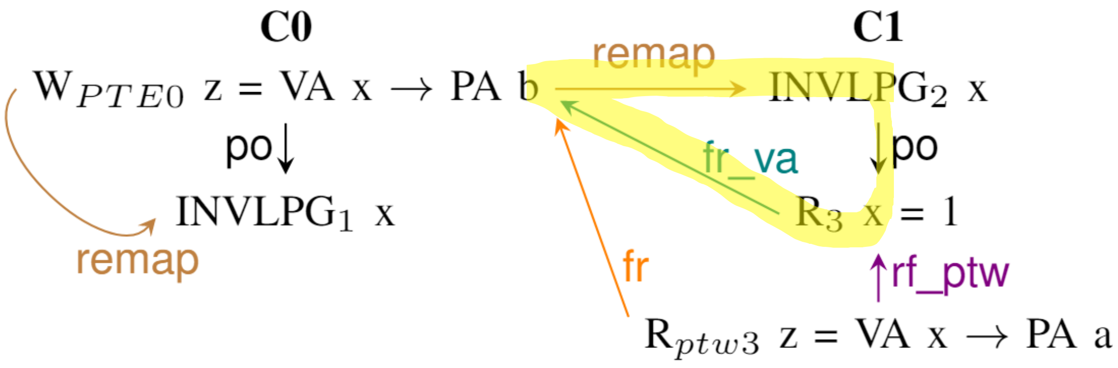
ISA-specific MTM → ELT synthesis engine → suite of ELTs

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

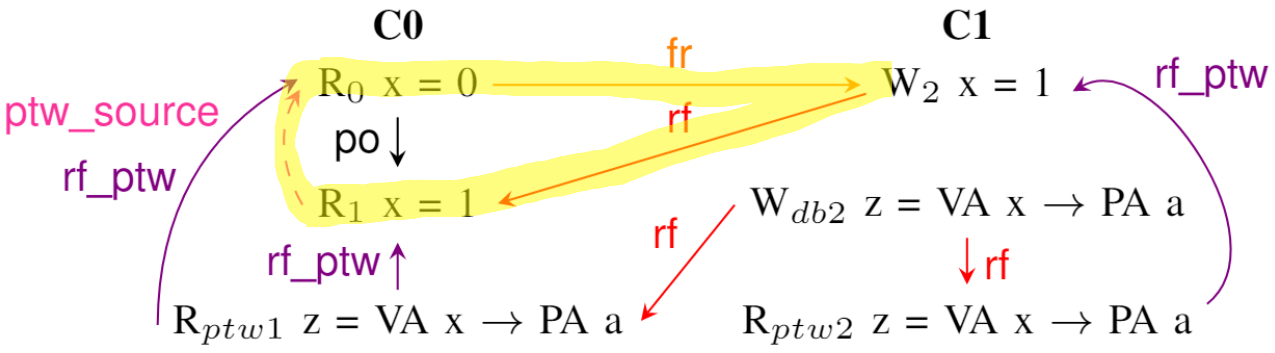
invlpg (required)

acylic[fr_va + remap + ^po]

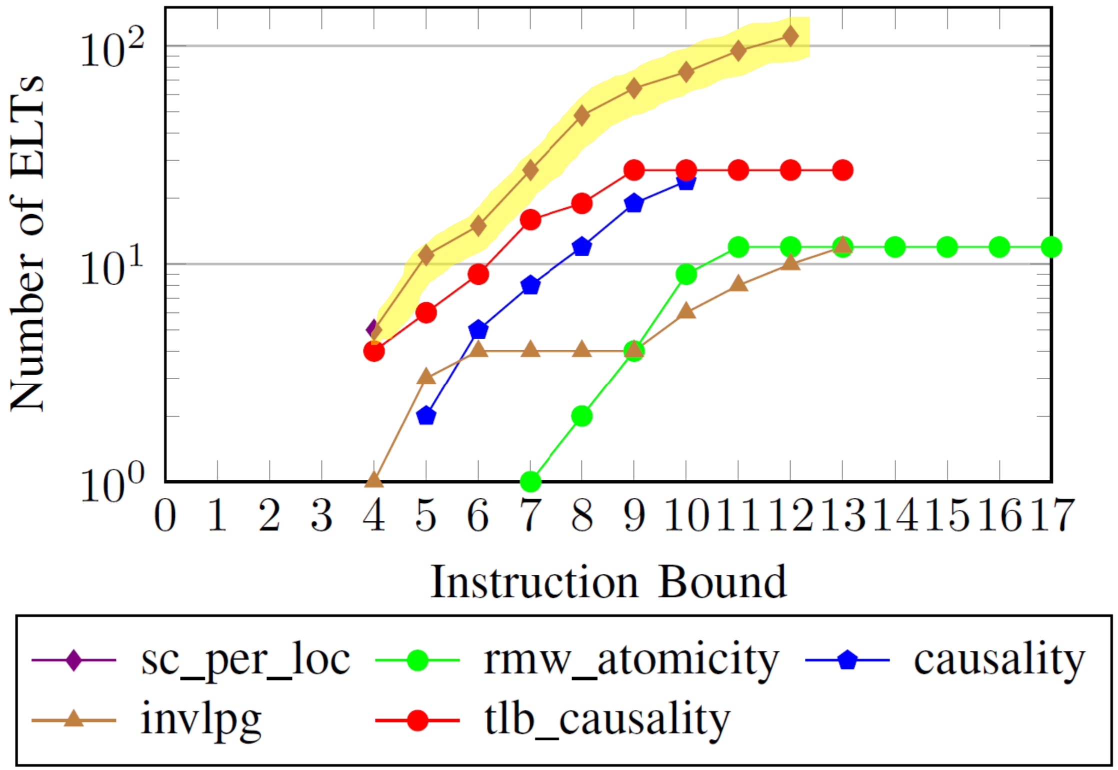


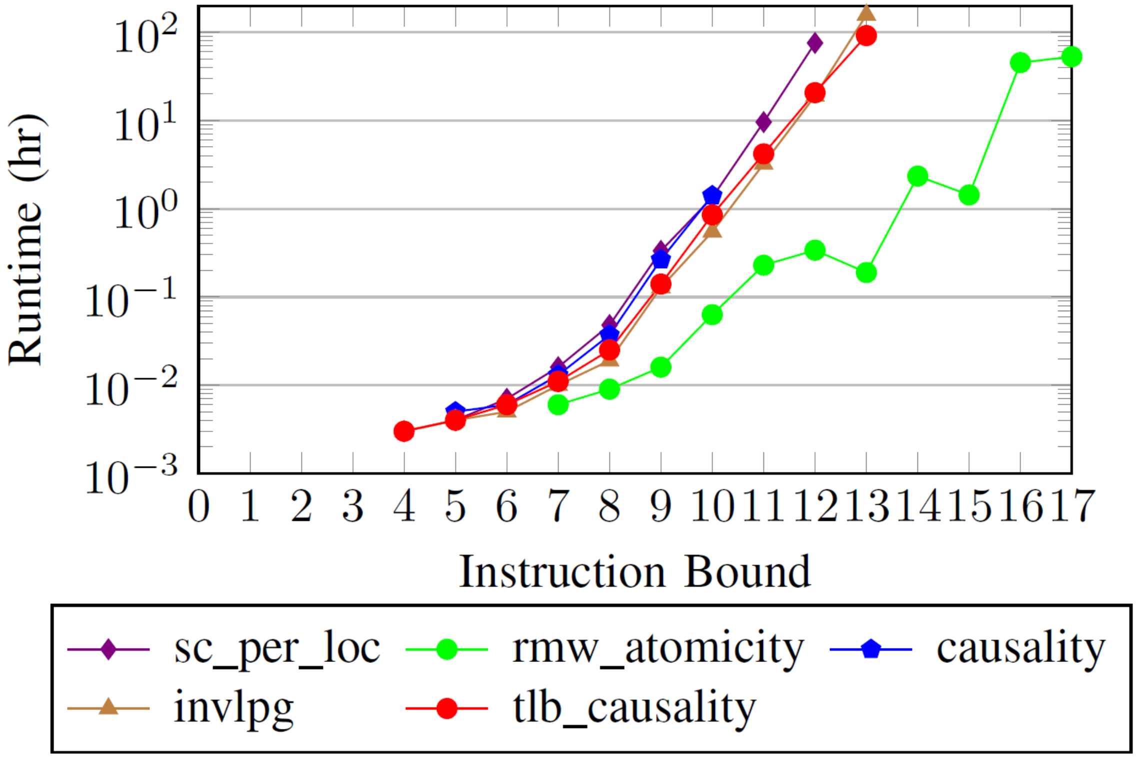
tlb_causality (auxiliary)

acyclic[ptw_source + com]


C0
C1
fr
rf_ptw
ptw_source
R$_0$ x = 0
W$_2$ x = 1
rf
rf_ptw
po
R$_1$ x = 1
W_{db2} z = VA x → PA a
rf
rf_ptw
rf
R_{ptw1} z = VA x → PA a
R_{ptw2} z = VA x → PA a

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

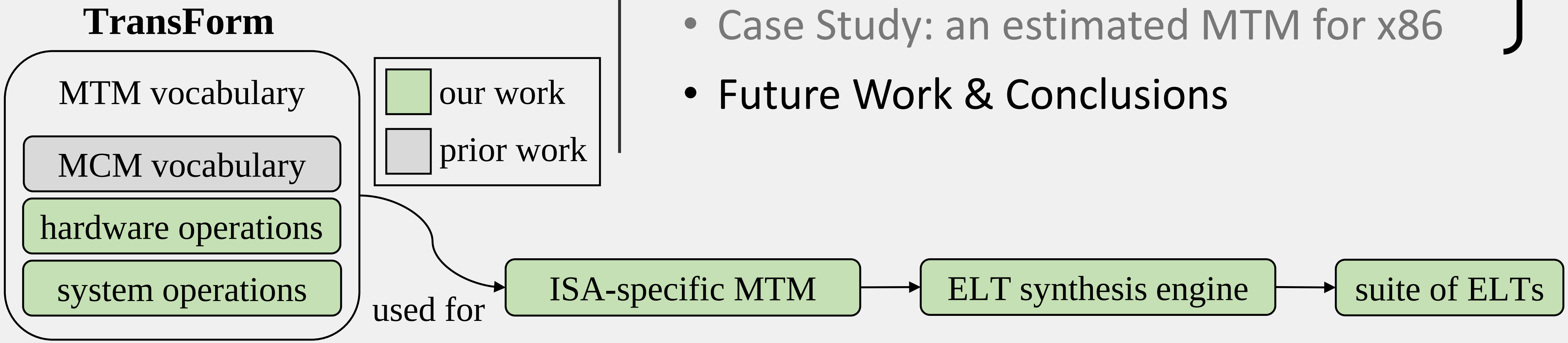
- Background on ISA-level MCM vocabulary
- Background on virtual memory systems

Prior Work

- Novel ISA-level MTM vocabulary
- Automating synthesis of ELTs
- Case Study: an estimated MTM for x86

My Work

- Future Work & Conclusions



TransForm

MTM vocabulary

MCM vocabulary

hardware operations

system operations

used for

ISA-specific MTM

ELT synthesis engine

suite of ELTs

🟩 our work

□ prior work

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

TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests

Naorin Hossain
Princeton University

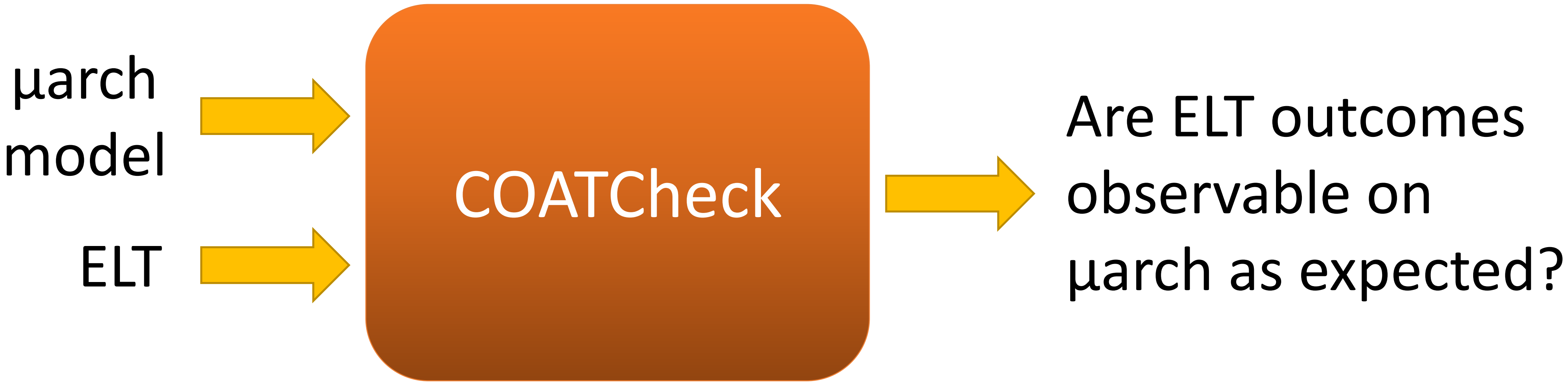
January 22, 2021

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



µarch model

ELT

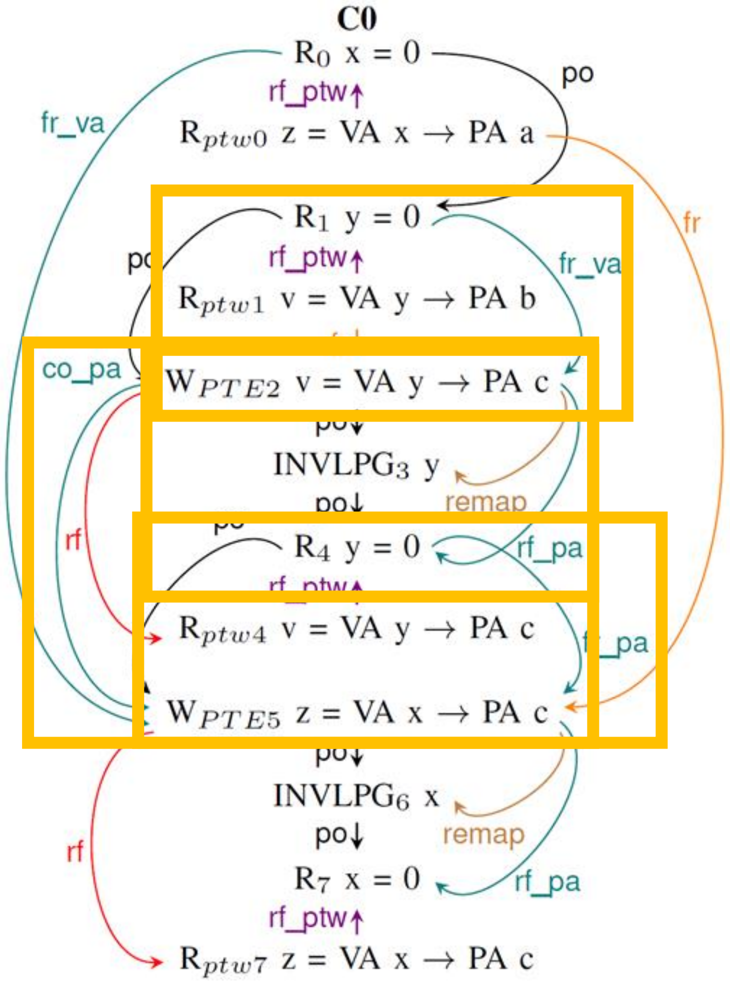
COATCheck

Are ELT outcomes observable on µarch as expected?

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

Communication relations can be extended to apply to virtual-to-physical address mappings

- **rf_pa**: maps a PTE write mapping VA v → PA p to instructions that access VA v → PA p.
- **fr_pa**: maps instructions that access VA v → PA p to PTE writes that map VA v’ → PA p.
- **co_pa**: maps PTE writes that change mappings for different VAs to the same PA.
- **fr_va**: maps instructions that access VA v → PA p to PTE writes that map VA v → PA p’.



Tests with instructions that do not contribute to forbidden outcomes violate minimality criterion

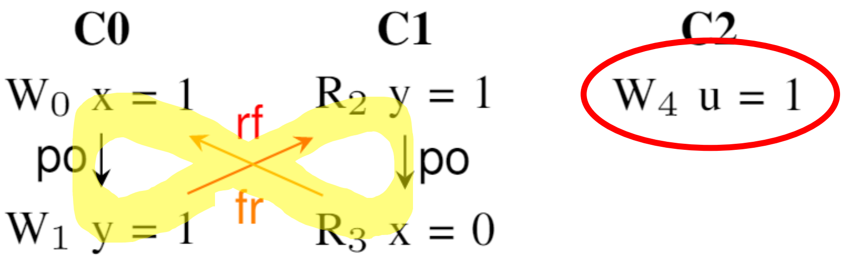
Minimality criterion: any **relaxation** on the test results in satisfying the transistency predicates

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


C0
C1
C2
W0 x = 1
po
W1 y = 1
rf
fr
R2 y = 1
po
R3 x = 0
W4 u = 1