The 80s never died: Automata theory for reversing modern CPUs

RootedCON - March 2020

vwzq.net  cgvkwq  github.com/cgvkwq
About me

I’m Pepe Vila (a.k.a. cgvvwzq)
PhD student at the IMDEA Software Institute
Worked as security consultant and pentester
Intern at Facebook and Microsoft Research

I used to mess with browsers and JavaScript...
...but fell into the side channel’s rabbit hole
x86 Assembly is Too High Level

CHANGE MY MIND
Motivation

Remember last year’s “Cache and syphilis”?  

dafuq is this pattern :S
Motivation

Knowing the cache replacement policy useful for finding eviction sets,

but also for optimal eviction strategies in rowhammer,
or high bandwidth covert channels
A primer on Hardware Caches

(data from Kaby Lake i7-8550U CPU)
A primer on Hardware Caches

- Memory partitioned in memory blocks (64 bytes = $2^6$)
- Cache partitioned in equally sized cache sets ($1024 = 2^{10} = 256KB / (64 \times 4)$)
- Cache sets have capacity for N cache lines (also known as ways or associativity)
A primer on Hardware Caches

- Memory partitioned in **memory blocks** (64 bytes = $2^6$)
- Cache partitioned in equally sized **cache sets** ($1024 = 2^{10} = 256\text{KB} / (64 \times 4)$)
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)
A primer on Hardware Caches

- Memory partitioned in memory blocks (64 bytes = $2^6$)
- Cache partitioned in equally sized cache sets (1024 = $2^{10} = 256$KB / (64 * 4))
- Cache sets have capacity for N cache lines (also known as ways or associativity)
A primer on Hardware Caches

- Memory partitioned in **memory blocks** (64 bytes = $2^6$)
- Cache partitioned in equally sized **cache sets** ($1024 = 2^{10} = 256KB / (64 * 4)$)
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)
A primer on Hardware Caches

- Memory partitioned in memory blocks (64 bytes = $2^6$)
- Cache partitioned in equally sized cache sets (1024 = $2^{10} = 256$KB / (64 * associativity))
- Cache sets have capacity for N cache lines (also known as ways or associativity)
A primer on Hardware Caches

- Memory partitioned in **memory blocks** (64 bytes = 2⁶)
- Cache partitioned in equally sized **cache sets** (1024 = 2¹⁰ = 256KB / (64 * associativity))
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)
A primer on Hardware Caches

- Memory partitioned in memory blocks (64 bytes = 2^6)
- Cache partitioned in equally sized cache sets (1024 = 2^{10} = 256KB / (64 * associativity))
- Cache sets have capacity for N cache lines (also known as ways or associativity)
A primer on Hardware Caches

- Memory partitioned in **memory blocks** (64 bytes = $2^6$)
- Cache partitioned in equally sized **cache sets** ($1024 = 2^{10} = 256$KB / (64 * associativity))
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)

![Diagram of memory, CPU, and cache](image)

- MISS
- 256KBs Cache
- Set 0
- Set 1
- Tag
- Data
- Associativity
- Tag
- Set
- Offset
- 10
- 6
- block 0
- block 1
- block 2
- ...
A primer on Hardware Caches

- Memory partitioned in **memory blocks** (64 bytes = $2^6$)
- Cache partitioned in equally sized **cache sets** ($1024 = 2^{10} = 256\text{KB} / (64 \times \text{associativity})$)
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)

![Diagram showing memory and cache organization with tags, sets, and associativity](chart.png)
A primer on Hardware Caches

- Cache set partition exploits programs’ **spatial locality**
- Replacement policy decides which blocks to evict exploiting programs’ **temporal locality**
- What does a replacement policy look like?
  - First Input First Output (FIFO), Least Recently Used (LRU), Pseudo-LRU, etc.
  - These examples keep track of the order or **ages** of blocks, and evict oldest one
- More complex policies nowadays, but same idea: maintain some metadata or **control state**
Caches as Mealy machines

- Natural **abstraction** for an individual cache set
- **Input** alphabet = set of memory blocks, e.g. \{a, b, c\} mapping to the same cache set
- **Output** alphabet = \{H, M\} (hit or miss) for the observable result of accessing a given block
- Every **state** represents the content of the cache set plus its control state (or metadata)

Example: 2-way FIFO cache with 3 blocks \{a, b, c\}
Previous work
## Previous work

<table>
<thead>
<tr>
<th></th>
<th>Others</th>
<th>Abel &amp; Reineke</th>
<th>Rueda’s MS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Automatic</strong></td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Supported class of policies</strong></td>
<td>Individual</td>
<td>Permutation-based</td>
<td>Deterministic</td>
</tr>
<tr>
<td><strong>On real hardware</strong></td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td><strong>Human readable</strong></td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td><strong>Correctness</strong></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
return i;
Our approach

Program synthesis

Template

Explanation

int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
    return i;

int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
    return i;

ITS SCIENCE B**TCH
## Previous work vs. our approach

<table>
<thead>
<tr>
<th></th>
<th>Others</th>
<th>Abel &amp; Reineke</th>
<th>Rueda’s MS</th>
<th>Our</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Automatic</strong></td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Supported class of policies</strong></td>
<td>Individual</td>
<td>Permutation-based</td>
<td>Deterministic</td>
<td>Deterministic</td>
</tr>
<tr>
<td><strong>On real hardware</strong></td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Human readable</strong></td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Correctness</strong></td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
Program synthesis

Automata learning

Policy abstraction

CacheQuery

Template

int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
return i;
CacheQuery: a hardware interface

- Frees the user from **low-level details** like set mapping, timing, cache filtering, code generation, and system’s interferences.

- Accepts sequences of **blocks** decorated with an **optional tag**: ? indicates access should be profiled, ! indicates that block should be invalidated, no tag means access.

- Support for **macros**:
  - @ expansion, _ wildcard, power operator, etc.
  - E.g. For assoc=4: @ x _? expands to:
    - (a b c d) x [a b c d]?, which expands to
    - {a b c d x a?, a b c d x b?, a b c d x c?, a b c d x d?}
    - and returns {M, H, H, H}
CacheQuery: demo

- Disable system’s noise
- REPL interactive session
- Target specific level and set
- Ask arbitrary queries
Polca: a cache automaton abstraction

int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
return i;
Polca: a cache automaton abstraction

- Why not learn directly from the cache?
  - **Redundancy** → Replacement policy is agnostic of the specific content
  - Policy’s logic should depend only on the **control state** (metadata)
  - Cache’s **content management** increases automata complexity and learning cost

- We abstract the replacement policy from the cache content management!
Polca: a cache automaton abstraction

Abstract automaton
Replacement policy

Concrete automaton
Cache management

Input: \{h(0), h(1), ..., h(n-1), m()\}
Output: \{_, 0, 1, ..., n-1\}

Input: \{A, B, C, ....\}
Output: \{H, M\}
Polca: a cache automaton abstraction

- Example of **concrete cache automaton** for 2-ways LRU with fixed input alphabet \{a, b, c\} and output \{H, M\}

- Example of corresponding **abstract policy automaton**, using input alphabet \{h(0), h(1), m()\} and output \{_, 0, 1\}

- 12 vs. 2 states $\Rightarrow$ much **easier to learn**!

- Reduction of (associativity+1)! in most cases
LearnLib: an automata learning framework

int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
    return i;
Automata learning

- Dana Angluin’s L\* algorithm:
  
  “Learning regular sets from queries and counterexamples” (1987)

- Student-Teacher protocol. Student asks 2 types of questions to the teacher:

  - **membership** - Is a word ‘w’ in the target language ‘U’? Yes / No
    - interaction with SUL (System Under Learning)

  - **equivalence** - Does the automaton accept language ‘U’? Yes / counterexample
    - needs access to a specification or oracle

- Find the **minimal automaton** for U with **polynomial cost** in the number of states of the automaton and the length of longest counterexample
L* by example

- Teacher knows language $U = \{aa, bb\}$ (alphabet $\Sigma = \{a, b\}$)
- Student asks if ‘$\varepsilon$’, ‘a’, and ‘b’ are in $U$ and obtains the following Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set of strings $S$, represents the states $S \cdot \Sigma$

- Table entries: $(s,e) = 1$ iff $uv \in U$ - summarizes all membership queries
- From an observation table we can directly construct an automaton if table is
  - closed - $\forall t \in S.\Sigma \ \exists s \in S \ \text{row}(t) = \text{row}(s)$
  - consistent - $\forall s_1, s_2$ s.t. $\text{row}(s_1) = \text{row}(s_2) \Rightarrow \forall a \in \Sigma \ \text{row}(s_1.a) = \text{row}(s_2.a)$
L* by example

Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
</tbody>
</table>

(source: https://www.csa.iisc.ac.in/~deepakd/atc-2015/L_Star_Alg.pdf)
L* by example

Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
</tbody>
</table>

It is closed and consistent
Hypothesis: empty language!

Teacher says NO and returns: `ce = aa`

We need to **extend S with ‘ce’ and all its prefixes**
L* by example

Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>aa</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>?</td>
</tr>
<tr>
<td>aaa</td>
<td>?</td>
</tr>
<tr>
<td>aab</td>
<td>?</td>
</tr>
</tbody>
</table>

perform a more membership queries

(source: https://www.csa.iisc.ac.in/~deepakd/atc-2015/L_Star_Algo.pdf)
**L* by example**

Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>aa</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>0</td>
</tr>
<tr>
<td>aaa</td>
<td>0</td>
</tr>
<tr>
<td>aab</td>
<td>0</td>
</tr>
</tbody>
</table>

New table is closed, but not consistent

Row($\varepsilon$) = row(a), but row($\varepsilon$.a) $\neq$ row(a.a)

To fix it, we need to add the difference to the table by increasing column 37

(source: https://www.csa.iisc.ac.in/~deepakd/atc-2015/L_Star_Alg.pdf)
L* by example

Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>aa</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aaa</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aab</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

now it is closed and consistent

we make a new hypothesis, but teacher says NO: ce = bb

(source: https://www.csa.iisc.ac.in/~deepakd/atc-2015/L_Star_Alg.pdf)
L* by example

Observation Table:

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>aa</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bb</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aaa</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aab</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bba</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bbb</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table is closed and consistent, let's see if hypothesis is correct not?

nope ce = babb

(source: https://www.csa.iisc.ac.in/~deepakd/atc-2015/L_Star_Algo.pdf)
**L* by example**

- With one more step, we finally find the automaton accepting $U = \{aa, bb\}$

![Automaton Diagram]

- The algorithm ensures that on every hypothesis the **automaton is minimal**.

- Teacher can give arbitrarily long counterexamples.
LearnLib handles all the learning

- **LearnLib** is an open source Java framework for automata learning developed at the TU Dortmund University - [https://learnlib.de/](https://learnlib.de/)

- Angluin’s L* algorithm has been extended to **Mealy machines**:
  - Membership queries replaced by **output queries**
  - Equivalence queries **approximated by test sequences** for conformance testing
  - **Reset sequence** is bootstrapping problem, we solve it with Flush+Refill

**WP-method**: test sequence selection - given an upper bound on the number of states of the System Under Learning (SUL), guarantees equivalence
Sketch: synthesizing programs as explanations

**Template**

- **Program synthesis**
- **Automata Learning**
- **Polica**
- **CacheQuery**

```cpp
int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
if(state[i] == 3)
return i;
```
Automata models are great, but if we want to understand what is really happening...

This is only LRU with associativity 4, a fairly simple policy.
Domain knowledge or high-level view of a replacement policy:

- Each block has an associated age
- **Promotion** rule decides how the ages are updated upon a hit
- **Replacement** rule decides which block is evicted upon a miss
- **Insertion** rule decides the age of a new block
- **Normalization** rule describes how to normalize ages after/before a hit or miss (e.g. in MRU reset used bit when all are set)
Sketch: synthesizing programs as explanations

With that domain knowledge, we “sketch” a template of how replacement policies looks like:

```plaintext
hit (state, line) :: States×Lines → States
state = promote(state, line)
state = normalize(state, line)
return state

miss (state) :: States → States×Lines
Lines idx = -1
state = normalize(state, idx)
idx = evict(state)
state[idx] = insert(state, idx)
state = normalize(state, idx)
return ⟨state, idx⟩
```
Sketch: synthesizing programs as explanations

With that domain knowledge, we “sketch” a template of how replacement policies looks like:

**hit** (state, line) :: States×Lines → States
state = promote(state, line)
state = normalize(state, line)
return state

**miss** (state) :: States → States×Lines
Lines idx = -1
state = normalize(state, idx)
idx = evict(state)
state[idx] = insert(state, idx)
state = normalize(state, idx)
return ⟨state, idx⟩

Specify the grammar of the functions. For instance:

**promote** (state, pos) :: States×Lines → States
States final = state
if (??{boolExpr(state[pos])})
    final[pos] = ??{natExpr(state[pos])}
for(i in Lines)
    if(i != pos ∧ ??{boolExpr(state[pos], state[i])})
        final[i] = ??{natExpr(state[i])}
return final
Sketch: synthesizing programs as explanations

With that domain knowledge, we “sketch” a template of how replacement policies looks like:

```plaintext
hit (state, line) :: States×Lines → States
state = promote(state, line)
state = normalize(state, line)
return state

miss (state) :: States → States×Lines
Lines idx = -1
state = normalize(state, idx)
idx = evict(state)
state[idx] = insert(state, idx)
state = normalize(state, idx)
return ⟨state, idx⟩
```

Specify the grammar of the functions. For instance:

```plaintext
promote (state, pos) :: States×Lines → States
States final = state
if (?{boolExpr(state[pos])})
  final[pos] = ?{natExpr(state[pos])}
for(i in Lines)
  if(i != pos ∧ ?{boolExpr(state[pos], state[i])})
    final[i] = ?{natExpr(state[i])}
return final
```

And encode the automaton’s output and transition functions as constraints.
Case Studies

- Learning from software simulated caches
- Learning from hardware
- Synthesizing Explanations
Case Study: Learning from Software-Simulated Caches

- Support for a **broader class of policies** than previous work
- Scale up to **larger associativities** than previous work
- Number of states still grows exponentially with associativity :(  

<table>
<thead>
<tr>
<th>Policy</th>
<th>Assoc.</th>
<th># States</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>2</td>
<td>2</td>
<td>0h 0m 0.14s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>0h 0m 0.38s</td>
</tr>
<tr>
<td>LRU</td>
<td>2</td>
<td>2</td>
<td>0h 0m 0.10s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24</td>
<td>0h 0m 0.22s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>720</td>
<td>0h 0m 32.70s</td>
</tr>
<tr>
<td>PLRU</td>
<td>2</td>
<td>2</td>
<td>0.10 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>0.22 s</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>128</td>
<td>1.46 s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>32768</td>
<td>34 h 18 m 25 s</td>
</tr>
<tr>
<td>MRU</td>
<td>2</td>
<td>2</td>
<td>0h 0m 0.10 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14</td>
<td>0h 0m 0.16 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>62</td>
<td>0h 0m 0.61 s</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>254</td>
<td>0h 0m 8.82 s</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1022</td>
<td>0h 5 m 58 s</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>4094</td>
<td>3h 59 m 20 s</td>
</tr>
<tr>
<td>LIP</td>
<td>2</td>
<td>2</td>
<td>0h 0m 0.10 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24</td>
<td>0h 0m 0.26 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>720</td>
<td>0h 0m 31.97 s</td>
</tr>
<tr>
<td>SRRIP-HP</td>
<td>2</td>
<td>12</td>
<td>0h 0m 0.16 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>178</td>
<td>0h 0m 1.46 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2762</td>
<td>0h 9 m 38 s</td>
</tr>
<tr>
<td>SRRIP-FP</td>
<td>2</td>
<td>16</td>
<td>0h 0m 0.19 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>256</td>
<td>0h 0m 7.27 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4096</td>
<td>2h 30 m 51 s</td>
</tr>
</tbody>
</table>

Table 2. Learning policies from software-simulated caches (with 36 hours timeout). We omit FIFO’s intermediate results.
## Case Study: Learning from Hardware

<table>
<thead>
<tr>
<th>CPU</th>
<th>Cache level</th>
<th>Assoc.</th>
<th>Slices</th>
<th>Sets per slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-4790 (Haswell)</td>
<td>L1</td>
<td>8</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>8</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>16</td>
<td>4</td>
<td>2048</td>
</tr>
<tr>
<td>i5-6500 (Skylake)</td>
<td>L1</td>
<td>8</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>4</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>12</td>
<td>8</td>
<td>1024</td>
</tr>
<tr>
<td>i7-8850U (Kaby Lake)</td>
<td>L1</td>
<td>8</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>4</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>16</td>
<td>8</td>
<td>1024</td>
</tr>
</tbody>
</table>
Case Study: Learning from Hardware

Challenges:

● Not all sets implement the same policy (set-duelling) ➔ we identify leader sets

● Not all leader sets are deterministic (probabilistic and adaptive policies) ➔ :(  

● L3 has too large associativities ➔ we use Intel’s CAT to virtually reduce associativity

● Reset sequences not 100% reliable ➔ required some manual adjustment
Case Study: Learning from Hardware

<table>
<thead>
<tr>
<th>CPU</th>
<th>Level</th>
<th>Assoc.</th>
<th>Sets</th>
<th>States</th>
<th>Policy</th>
<th>Reset Seq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-4790 (Haswell)</td>
<td>L1</td>
<td>8</td>
<td>0 – 63</td>
<td>128</td>
<td>PLRU</td>
<td>@ @</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>8</td>
<td>0 – 511</td>
<td>128</td>
<td>PLRU</td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>16</td>
<td>512 – 575 (only for slice 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>768 – 831 (only for slice 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i5-6500 (Skylake)</td>
<td>L1</td>
<td>8</td>
<td>0 – 63</td>
<td>128</td>
<td>PLRU</td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>4</td>
<td>0 – 1023</td>
<td>160</td>
<td>New1</td>
<td>D C B A @</td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>4†</td>
<td>0 33 132 165 264 297 396 429 528 561 660 693 792 825 924 957</td>
<td>175</td>
<td>New2</td>
<td>@</td>
</tr>
<tr>
<td>i7-8550U (Kaby Lake)</td>
<td>L1</td>
<td>8</td>
<td>0 – 63</td>
<td>128</td>
<td>PLRU</td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>4</td>
<td>0 – 1023</td>
<td>160</td>
<td>New1</td>
<td>D C B A @</td>
</tr>
<tr>
<td></td>
<td>L3</td>
<td>4†</td>
<td>0 33 132 165 264 297 396 429 528 561 660 693 792 825 924 957</td>
<td>175</td>
<td>New2</td>
<td>@</td>
</tr>
</tbody>
</table>

Table 4. Results of learning policies from hardware caches. † indicates that the associativity has been virtually reduced using CAT. The ‘Sets’ column specifies the analyzed cache sets (unless otherwise specified, the findings apply to all slices).
## Case Study: Synthesizing Explanations

<table>
<thead>
<tr>
<th>Policy</th>
<th>States</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>4</td>
<td>18ms</td>
</tr>
<tr>
<td>LRU</td>
<td>24</td>
<td>81ms</td>
</tr>
<tr>
<td>PLRU</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>LIP</td>
<td>24</td>
<td>4s</td>
</tr>
<tr>
<td>MRU</td>
<td>14</td>
<td>40s</td>
</tr>
<tr>
<td>SRRIP-HP</td>
<td>178</td>
<td>105h</td>
</tr>
<tr>
<td>SRRIP-FP</td>
<td>256</td>
<td>48h</td>
</tr>
<tr>
<td>New1</td>
<td>160</td>
<td>9h</td>
</tr>
<tr>
<td>New2</td>
<td>175</td>
<td>26h</td>
</tr>
</tbody>
</table>
Case Study: Synthesizing Explanations

Description of Skylake/Kaby Lake L2’s (New1):

Initial insertion on a flushed cache set:

```c
int[4] s0 = {3,3,3,0};

    int[4] final = state;
    // Promotion
    final[pos] = 0;
    // Is there a block with age 3?
    bit found = 0;
    for(int j = 0; j < 4; j = j + 1)
        if(!found)
            for(int i = 0; i < 4; i = i + 1)
                if(!found && final[i] == 3)
                    found = 1;
    // If not, increase all blocks except promoted one
    if(!found)
        for(int i = 0; i < 4; i = i + 1)
            if(i != pos)
                final[i] = final[i] + 1;
    return final;
```

```c
    int[4] final = state;
    int replace = missIdx(state);
    // Insertion
    final[replace] = 1;
    // Is there a block with age 3?
    bit found = 0;
    for(int j = 0; j < 4; j = j + 1)
        if(!found)
            for(int i = 0; i < 4; i = i + 1)
                if(!found && final[i] == 3)
                    found = 1;
    // If not, increase all blocks except inserted one
    if(!found)
        for(int i = 0; i < 4; i = i + 1)
            if(replace != i)
                final[i] = final[i] + 1;
    return final;
```

```c
    int[4] final = state;
    int replace = missIdx(state);
    // Insertion
    final[replace] = 1;
    // Is there a block with age 3?
    bit found = 0;
    for(int j = 0; j < 4; j = j + 1)
        if(!found)
            for(int i = 0; i < 4; i = i + 1)
                if(!found && final[i] == 3)
                    found = 1;
    // If not, increase all blocks except inserted one
    if(!found)
        for(int i = 0; i < 4; i = i + 1)
            if(replace != i)
                final[i] = final[i] + 1;
    return final;
```

```c
    int[4] final = state;
    int replace = missIdx(state);
    // Insertion
    final[replace] = 1;
    // Is there a block with age 3?
    bit found = 0;
    for(int j = 0; j < 4; j = j + 1)
        if(!found)
            for(int i = 0; i < 4; i = i + 1)
                if(!found && final[i] == 3)
                    found = 1;
    // If not, increase all blocks except inserted one
    if(!found)
        for(int i = 0; i < 4; i = i + 1)
            if(replace != i)
                final[i] = final[i] + 1;
    return final;
```
Case Study: Synthesizing Explanations

Description of Skylake/Kaby Lake L3’s (New2):

Initial insertion on a flushed cache set:

```c
int[4] s0 = {3,3,3,3};
```

```c
int[4] final = state;
// Promotion
if (final[pos] > 1)
    final[pos] = 1;
else
    final[pos] = 0;
// Is there a block with age 3?
if(!found)
    for(int i = 0; i < 4; i = i + 1)
        if(!found && final[i] == 3)
            found = 1;
// If not, increase all blocks
if(!found)
    for(int i = 0; i < 4; i = i + 1)
        final[i] = final[i] + 1;
return final;
```

```c
int[4] final = state;
int replace = missIdx(state);
// Insertion
final[replace] = 1;
// Is there a block with age 3?
if(!found)
    for(int j = 0; j < 4; j = j + 1)
        if(!found)
            for(int i = 0; i < 4; i = i + 1)
                if(!found && final[i] == 3)
                    found = 1;
// If not, increase all blocks
if(!found)
    for(int i = 0; i < 4; i = i + 1)
        final[i] = final[i] + 1;
return final;
```

// Replace first block with age 3 starting from the left
int missIdx (int[4] state)
for(int i = 0; i < 4; i = i + 1)
    if(state[i] == 3)
        return i;
```
SO IN SUMMARY
Conclusions

- End-to-end solution for learning deterministic hardware replacement policies
- We are able to automatically infer human-readable descriptions
- We uncover 2 previously undocumented policies used in recent Intel processors
- All our contributions are independent and ready to use in alternative workflows

https://github.com/cgvwzq/cachequery
https://github.com/cgvwzq/polca
Thank you for listening! Questions?

https://github.com/cgvwzq/cachequery
https://github.com/cgvwzq/polca
Adaptive Insertion Policies for High Performance Caching

Intel Ivy Bridge Cache Replacement Policy
http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

Measurement-based Modeling of the Cache Replacement Policy

Learning Cache Replacement Policies using Register Automata
Extra material
Extra: Adaptive Policies and Leader Sets

- We use thrashing sequences (e.g. @ M @?) on a per cache set basis to identify leader sets:
  - Haswell i7-4790:
    - sets 512 – 575 in slice 0 fixed policy susceptible to thrashing.
    - sets 768 – 831 in slice 0 fixed thrash resistant policy (seems not deterministic).
    - rest of sets follow the policy producing less misses.
  - Skylake i5-6500 and Kaby Lake i7-8550U:
    - sets whose indexes satisfy \(((\text{set} \& 0x3e0) \gg 5) \oplus (\text{set} \& 0x1f)) = 0x0\) \(\land (\text{set} \& 0x2) = 0x0\) fixed policy susceptible to thrashing (group 1)
    - rest of sets seem to use an adaptive policy
    - but sets whose indexes satisfy \(((\text{set} \& 0x3e0) \gg 5) \oplus (\text{set} \& 0x1f)) = 0x1f\) \(\land (\text{set} \& 0x2) = 0x2\) change differently (group 2), still WIP for this

group 1: 0 33 132 165 264 297 396 429 528 561 660 693 792 825 924 957

group 2: 31 62 155 186 279 310 403 434 527 558 651 682 775 806 899 930