CacheQuery: Learning Replacement Policies from Hardware Caches

Pepe Vila
IMDEA Software Institute
pepe.vila@imdea.org

Pierre Ganty
IMDEA Software Institute
pierre.ganty@imdea.org

Marco Guarnieri
IMDEA Software Institute
marco.guarnieri@imdea.org

Boris Köpf
Microsoft Research
boris.koepf@microsoft.com

Abstract

We show how to infer deterministic cache replacement policies using off-the-shelf automata learning and program synthesis techniques. For this, we construct and chain two abstractions that expose the cache replacement policy of any set in the cache hierarchy as a membership oracle to the learning algorithm, based on timing measurements on a silicon CPU. Our experiments demonstrate an advantage in scope and scalability over prior art and uncover 2 previously undocumented cache replacement policies.

This paper uses colours to provide a clearer notation. For a better experience, please print or view this paper in colours.

1 Introduction

Understanding the timing behavior of modern CPUs is crucial for optimizing code and for ensuring timing-related security and safety properties. Examples of such properties are bounds on programs' worst-case execution time [38] or on the amount of information leaked via timing [8, 12]. Unfortunately, the timing behavior of modern high-performance processors depends on subtle and poorly documented details of their microarchitecture, which has triggered laborious efforts to reverse-engineer microarchitectural details [4, 18, 26, 29].

Cache replacement policies have received specific attention, because they control the content of the memory hierarchy and hence heavily influence execution time [1, 2, 7, 32, 39]. However, only few authors have approached the problem of inferring replacement policies in a principled way.

- Rueda [32] uses off-the-shelf techniques for learning register automata to infer cache replacement policies. The approach learns replacement policies with small state-spaces from noiseless simulator traces, but has not been successfully applied to actual hardware.
- Abel and Reineke [1] present an approach that infers so-called permutation-based replacement policies, which include LRU, FIFO, and PLRU [14]. The approach has been used to infer policies from performance counter measurements on hardware. However, permutation-based policies are restrictive in that they do not include important examples such as MRU [25], SRRIP [20], or the ones that are implemented in the lower-level caches of recent Intel CPUs.

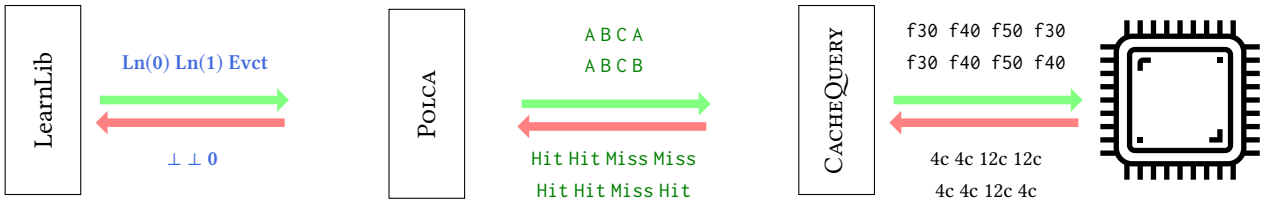
Furthermore, both approaches share a common drawback: the inferred policy representations are not easily interpretable by humans.

Approach. In this paper we propose an approach for learning cache replacement policies that goes beyond the state-of-the-art in that it (1) can learn arbitrary deterministic replacement policies (2) from real-time (or performance counter) measurements on silicon CPUs. Moreover, we show how to (3) apply program synthesis to yield human-readable interpretations of the inferred policies.

Our approach relies on two contributions that enable us to leverage off-the-shelf automata learning tools [19, 34] for attacking the problem:

- A tool, called CacheQuery, that provides an abstract interface to any *individual cache set* within the cache hierarchy[1] of a silicon CPU. With CacheQuery, users can specify a cache set (say: set 63 in the L2 cache) and a pattern of memory accesses (say: `A B C A B C`), and they receive as output a sequence (say: `Miss Miss Miss Hit Hit Hit`) representing the hits and misses produced when performing a sequence of memory loads to addresses that are mapped into the specified cache set and that follow the specified pattern. CacheQuery liberates the user from dealing with intricate details such as the virtual-to-physical memory mapping, cache slicing, set indexing, interferences from other levels of the cache hierarchy, and measurement noise, and thus enables civilized interactions with an individual cache set. See Figure 1c.
- An algorithm, called Polca, that provides an abstract interface to the *cache replacement policy* based on an interface to a cache set, such as CacheQuery. Specifically, Polca translates inputs to the replacement policy (which refer to the cache *lines*) into inputs to the cache set (which refer to the *memory blocks* that are stored in it). To achieve this, Polca itself keeps track of the current cache content, e.g., by issuing queries to the cache interface to determine which block has been evicted in a miss. Polca exploits the

[1] For a primer on hardware caches, see § 2.1.



(a) LearnLib issues membership queries to the system under learning (SUL). The salient feature of our approach is that the language of the SUL refers to cache lines and *not* to the cache content. When the learning loop terminates, our tool returns an automaton describing the cache replacement policy under learning.

(b) Polca translates a sequence of requests for cache lines Ln(i) or evictions Evct into sequences of abstract memory blocks. For this, the algorithm keeps track of the current cache state (here: blocks A/B in lines 0/1). Evct spawns multiple sequences that first produce a cache miss (here: C), followed by accesses to all previously contained blocks to infer which line was evicted (here: 0).

(c) CacheQuery receives as input sequences of abstract blocks (e.g., A B C A) and translates them into distinct concrete memory blocks (e.g., A->f30) that all map into the same cache set. It loads the corresponding memory blocks, counts the corresponding clock cycles, and returns for each load whether it was a cache hit (e.g., 4c->Hit) or a miss.

Figure 1. Leveraging Polca and CacheQuery to learn a toy replacement policy of a 2-way set associative CPU cache using the LearnLib [34] framework.

data-independence symmetry of the replacement policy that would otherwise have to be inferred by the learning algorithm, and is key to making automata learning work in this domain. See Figure 1b.

We use Polca as a so-called *membership oracle* for the replacement policy to be learned from an abstract cache set (which can be implemented by CacheQuery or by a software-simulated cache). The oracle provides an interface to libraries such LearnLib [34], which enables us to leverage the state-of-the-art in automata learning for inferring replacement policies of silicon CPUs. We give formal relative completeness guarantees for the learned policies, based on the correctness of Polca and LearnLib. The full learning pipeline is depicted in Figure 1.

Finally, we show how to use program synthesis to automatically derive a higher-level representation of the learned replacement policy. The main component of our synthesis step is a *program template* for replacement policies, which we base on concepts used to describe replacement policies in the microarchitecture community [20]. By combining the policy template with a set of constraints derived from the learned automaton, we can rely on off-the-shelf synthesis solvers to synthesize high-level policy representations [33].

Evaluation. We evaluate our approach in 3 case studies:

1. We evaluate the scalability of learning replacement policies with Polca. To this end, we learn a comprehensive set of deterministic replacement policies (including FIFO, LRU, PLRU [14], MRU [25], LIP [30], and different variants of SRRIP [20]) from the noiseless hit-miss traces produced by a software-simulated cache. Our experiments demonstrate that Polca enables LearnLib to infer policies with state-spaces of more than 2 orders of magnitude larger than what was reported for direct applications of LearnLib to simulator traces [32].
2. We evaluate the effectiveness of learning with Polca and CacheQuery on the L1, L2, L3 caches of three recent Intel CPUs.[2] Our experiments show that we can effectively learn cache replacement policies up to associativity 8 from timing measurements on modern CPUs. While some of the policies were known, we do uncover 2 policies that have not yet been documented in the literature.
3. We evaluate our template-based synthesis approach by synthesizing programs for 8 out of 9 different policies (obtained from both the simulators and silicon CPUs), for a fixed associativity 4. This allows us to provide high-level descriptions for the 2 previously undocumented policies.

Summary of Contributions. In summary, we present a practical end-to-end solution for inferring deterministic cache replacement policies using off-the-shelf techniques for automata learning and program synthesis. The enabling contribution is a chain of two abstractions that exposes a membership oracle to the cache replacement policy, based on timing measurements on a silicon CPU.

Our tools, CacheQuery and Polca, are available, together with the learned models and synthesized programs, at

https://github.com/cgvwzq/cachequery/,

and

https://github.com/cgvwzq/polca/.

[2]For L3 caches with adaptive policies, we focus on the deterministic leader cache sets, as non-deterministic components are out of scope.

2 Modeling Caches and Policies

In this section we present our model of hardware caches. A key feature, inspired by [10], is that we distinguish between the replacement policy (§ 2.2), which determines the cache lines to be replaced, and the cache itself (§ 2.3), which stores and retrieves memory blocks (according to a replacement policy). We start by introducing necessary background.

2.1 A Primer on Hardware Caches

Caches are fast but small memories that bridge the latency gap between the CPU and the main memory. To profit from spatial locality and to reduce management overhead, the main memory is logically partitioned into a set of *blocks*.

Each block is cached as a whole in a cache *line* of the same size. When accessing a block, the cache logic determines whether the block is stored in the cache (a cache hit) or not (a cache miss). For this purpose, caches are partitioned into equally sized cache *sets*. The capacity of a set-associative cache set is called *associativity* (or *ways*) and represents the number of lines per set. Because the cache is smaller than the main memory, upon a cache miss, a *replacement policy* must decide which memory block to *evict* in order to make room for the more recent block.

In this work, we consider n-way set associative caches, i.e., caches where all cache sets consist of n lines, and we focus on individual cache sets. For brevity's sake, in the following we refer to a cache set of an n-way set-associative cache simply as an n-way cache.

2.2 Replacement Policy Model

We model the replacement policy of a cache set as a deterministic, finite-state Mealy machine that accepts inputs of the form $\mathbf{Ln(i)}$, for accessing the i-th cache line, and $\mathbf{Evct}$, for requesting a cache line to be freed (cf. Table 1). Given an input, the policy updates its control state and outputs the index of the line to be freed (or $\bot$ otherwise).

	Policy	Cache
Input	$\{\mathbf{Ln(0)}, \ldots, \mathbf{Ln(n-1)}\} \cup \{\mathbf{Evct}\}$	$\mathtt{Blocks}$
Output	$\{\bot\} \cup \{\mathbf{0}, \ldots, \mathbf{n-1}\}$	$\{\mathtt{Hit}, \mathtt{Miss}\}$

Table 1. Policy and cache alphabets (associativity n)

Definition 2.1. A *replacement policy* of associativity $n \in \mathbb{N}$ is a Mealy machine $\langle \mathbf{CS}, \mathbf{cs_0}, \mathbf{IP}, \mathbf{OP}, \delta, \lambda \rangle$ consisting of:

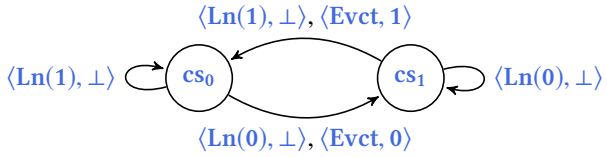
- a finite set of *control states* $\mathbf{CS}$;
- an *initial* control state $\mathbf{cs_0} \in \mathbf{CS}$;
- the set of inputs $\mathbf{IP} = \{\mathbf{Ln(0)}, \ldots, \mathbf{Ln(n-1)}\} \cup \{\mathbf{Evct}\}$;
- the set of outputs $\mathbf{OP} = \{\bot\} \cup \{\mathbf{0}, \ldots, \mathbf{n-1}\}$;
- a transition function $\delta : \mathbf{CS} \times \mathbf{IP} \to \mathbf{CS}$; and
- an output function $\lambda : \mathbf{CS} \times \mathbf{IP} \to \mathbf{OP}$.

We require that, (a) λ returns a value in $\{\mathbf{0}, \ldots, \mathbf{n-1}\}$ when given the input $\mathbf{Evct}$; and (b) λ returns $\bot$ when given an input in $\{\mathbf{Ln(0)}, \ldots, \mathbf{Ln(n-1)}\}$.

We write $\mathbf{cs} \xrightarrow{\langle \mathbf{i}, \mathbf{o} \rangle} \mathbf{cs'}$ when $\delta(\mathbf{cs}, \mathbf{i}) = \mathbf{cs'}$ and $\lambda(\mathbf{cs}, \mathbf{i}) = \mathbf{o}$.

We now introduce the trace semantics of policies, where traces are sequences of input/output pairs. We use standard sequence notation: S^* is the set of finite sequences over S, ε is the empty sequence, and $s_1 \cdot s_2$ denotes sequence concatenation. The *trace semantics* of a policy $\mathbf{P}$ is the set $[\![\mathbf{P}]\!] \subseteq (\mathbf{IP} \times \mathbf{OP})^*$ of all sequences $\langle \mathbf{i_1}, \mathbf{o_1} \rangle \cdot \langle \mathbf{i_2}, \mathbf{o_2} \rangle \cdot \ldots \cdot \langle \mathbf{i_m}, \mathbf{o_m} \rangle$ for which there are control states $\mathbf{cs_1}, \ldots, \mathbf{cs_m}$ such that $\mathbf{cs_0} \xrightarrow{\langle \mathbf{i_1}, \mathbf{o_1} \rangle} \mathbf{cs_1} \xrightarrow{\langle \mathbf{i_2}, \mathbf{o_2} \rangle} \ldots \xrightarrow{\langle \mathbf{i_m}, \mathbf{o_m} \rangle} \mathbf{cs_m}$.

Example 2.2. Consider a *Least Recently Used* (LRU) replacement policy, where the least recently used block is the one to be evicted. The LRU policy with associativity 2 can be formalized with the following Mealy Machine:



There are two control states $\mathbf{cs_0}$ and $\mathbf{cs_1}$, where $\mathbf{cs_i}$ indicates that line $\mathbf{i}$ contains the least recently used block.

2.3 Cache Model

We model a cache of associativity n as a Labeled Transition System (LTS) that accepts as input elements $\mathtt{b}$ from a potentially infinite set of memory blocks $\mathtt{Blocks}$, and that produces as output a $\mathtt{Hit}$ when block $\mathtt{b}$ is in the cache, and a $\mathtt{Miss}$ otherwise (cf. Table 1).

Each state of the cache is a pair $\langle \mathtt{cc}, \mathbf{cs} \rangle$ consisting of the cache content $\mathtt{cc} \in \mathtt{CC}^n$, which is an n-tuple of memory blocks without repetitions, and the control state $\mathbf{cs}$ of a replacement policy $\mathbf{P}$. Formally:

Definition 2.3. An *n-way cache* induced by a policy $\mathbf{P} = \langle \mathbf{CS}, \mathbf{cs_0}, \mathbf{IP}, \mathbf{OP}, \delta, \lambda \rangle$ is an LTS $\mathtt{C}(\mathbf{P}, \mathtt{cc_0}, n) = \langle \mathtt{S}, \mathtt{s_0}, \mathtt{IC}, \mathtt{OC}, \Longrightarrow \rangle$ consisting of:

- a set of *cache states* $\mathtt{S} = \mathtt{CC}^n \times \mathbf{CS}$;
- an *initial* cache state $\mathtt{s_0} = \langle \mathtt{cc_0}, \mathbf{cs_0} \rangle \in \mathtt{S}$;
- a set of inputs $\mathtt{IC} = \mathtt{Blocks}$;
- a set of outputs $\mathtt{OC} = \{\mathtt{Hit}, \mathtt{Miss}\}$;
- a transition relation $\Longrightarrow \subseteq \mathtt{S} \times \mathtt{IC} \times \mathtt{OC} \times \mathtt{S}$ that is induced by the policy $\mathbf{P}$ following Figure 2.

We explain the cache's transition relation from Figure 2:

The rule Hit captures what happens upon access to a block that is cached: The rule (1) determines that $\mathtt{b}$ is stored in $\mathtt{cc}$'s $\mathbf{i}$-th line, and (2) updates the control state by executing the policy with input $\mathbf{Ln(i)}$.

The rule Miss captures what happens upon access to a block that is not cached: The rule (1) checks that the block $\mathtt{b}$ is not in the cache, (2) determines the line $\mathbf{i}$ of the block to evict by executing the policy with input $\mathbf{Evct}$, and (3) inserts $\mathtt{b}$ in the $\mathbf{i}$-th line and updates the cache state.

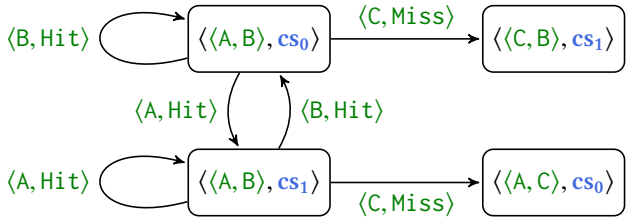
$$\frac{\mathtt{cc}[\mathtt{i}] = \mathtt{b} \qquad \mathtt{cs} \xrightarrow{\langle \mathtt{Ln(i)}, \bot \rangle} \mathtt{cs}'}{\langle \mathtt{cc}, \mathtt{cs} \rangle \xRightarrow{\langle \mathtt{b}, \mathtt{Hit} \rangle} \langle \mathtt{cc}, \mathtt{cs}' \rangle} \textsc{Hit}$$

$$\frac{\forall \mathtt{i}.\ \mathtt{cc}[\mathtt{i}] \neq \mathtt{b} \qquad \mathtt{cs} \xrightarrow{\langle \mathtt{Evct}, \mathtt{i} \rangle} \mathtt{cs}'}{\langle \mathtt{cc}, \mathtt{cs} \rangle \xRightarrow{\langle \mathtt{b}, \mathtt{Miss} \rangle} \langle \mathtt{cc}[\mathtt{i} \mapsto \mathtt{b}], \mathtt{cs}' \rangle} \textsc{Miss}$$

Figure 2. Transition relation for a cache $\Longrightarrow$ given that of a replacement policy $\longrightarrow$. Here, $\mathtt{cc}[\mathtt{i}]$ denotes the block stored in $\mathtt{cc}$'s $\mathtt{i}$-th line and $\mathtt{cc}[\mathtt{i} \mapsto \mathtt{b}]$ the cache content obtained by replacing the block in the $\mathtt{i}$-th line with $\mathtt{b}$

Similarly to a policy's trace semantics, we introduce a cache's trace semantics. The *trace semantics* of a cache $\mathtt{C}$ is the set $[\![\mathtt{C}]\!] \subseteq (\mathtt{IC} \times \mathtt{OC})^*$ of all sequences $\langle \mathtt{i}_1, \mathtt{o}_1 \rangle \cdot \langle \mathtt{i}_2, \mathtt{o}_2 \rangle \cdot \ldots \cdot \langle \mathtt{i}_m, \mathtt{o}_m \rangle$ for which there are cache states $\mathtt{s}_1, \ldots, \mathtt{s}_m$ such that $\mathtt{s}_0 \xRightarrow{\langle \mathtt{i}_1, \mathtt{o}_1 \rangle} \mathtt{s}_1 \xRightarrow{\langle \mathtt{i}_2, \mathtt{o}_2 \rangle} \ldots \xRightarrow{\langle \mathtt{i}_m, \mathtt{o}_m \rangle} \mathtt{s}_m$.

Example 2.4. The LTS of the cache induced by the LRU policy from Example 2.2 is as follows (we depict only part of the infinite LTS for 3 abstract blocks $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{C}$):



Consider the cache state $\langle \langle \mathtt{A}, \mathtt{B} \rangle, \mathtt{cs}_0 \rangle$. Accessing the block $\mathtt{B}$ produces a $\mathtt{Hit}$ since $\mathtt{B}$ is stored in line 1. Hence, we modify neither the cache content nor the control state because $\mathtt{cs}_0 \xrightarrow{\langle \mathtt{Ln}(1), \bot \rangle} \mathtt{cs}_0$ according to the policy. Accessing the block $\mathtt{A}$, which is stored in line 0, also produces a $\mathtt{Hit}$. This time, however, we update the control state since the least recently used cache line is now 1, i.e., $\mathtt{cs}_0 \xrightarrow{\langle \mathtt{Ln}(0), \bot \rangle} \mathtt{cs}_1$. In contrast, accessing the block $\mathtt{C}$, which is not in the cache, leads to a $\mathtt{Miss}$. The replacement policy determines that the block $\mathtt{C}$ has to be stored in line 0, i.e., $\mathtt{cs}_0 \xrightarrow{\langle \mathtt{Evct}, 0 \rangle} \mathtt{cs}_1$, and the new cache state is $\langle \langle \mathtt{C}, \mathtt{B} \rangle, \mathtt{cs}_1 \rangle$.

3 Polca: Learning Replacement Policies

In this section, we present our policy learning approach. We begin with some background on automata learning (§ 3.1). Next, we describe the two main components of our learning approach, i.e., membership (§ 3.2) and equivalence queries (§ 3.3) for replacement policies. We conclude by describing our prototype implementation of Polca on top of LearnLib (§ 3.4).

3.1 A Primer on Automata Learning

The prevalent approach to learning automata follows the student-teacher paradigm established by Angluin [5] and extended to Mealy machines by Niese [28]. There, the student's goal is to learn an unknown Mealy machine M by asking queries to the teacher.

There are two types of queries:

1. *membership queries*, where the student asks whether a given trace belongs to the machine M, and
2. *equivalence queries*, where the student asks whether a hypothesized Mealy machine H is (trace) equivalent to M.

Initially, the student knows only the input and output alphabets. By making a finite number of queries as prescribed by the learning algorithm, the student eventually learns M.

We next show how to answer membership and equivalence queries for a replacement policy (§ 2.2) based on interactions with the cache (§ 2.3), which enables us to leverage standard libraries for automata learning for inferring cache replacement policies.

3.2 Membership Queries

We now present Polca (see Algorithm 1), our algorithm for constructing a membership oracle for an unknown policy $\mathtt{P}$'s trace semantics (i.e., an oracle for deciding, given a trace $\mathtt{t}$, whether $\mathtt{t} \in [\![\mathtt{P}]\!]$), given a cache $\mathtt{C}$ induced by $\mathtt{P}$. For that, the algorithm translates a trace $\mathtt{t}$ of policy inputs and outputs into a series of *queries* (i.e., sequences of memory blocks) to the underlying cache $\mathtt{C}$ and, by observing $\mathtt{C}$'s behavior, determines whether $\mathtt{t} \in [\![\mathtt{P}]\!]$.

Polca receives as input an initial cache content $\mathtt{cc}_0$, a policy trace $\mathtt{t} \in (\mathtt{IP} \times \mathtt{OP})^*$, and the cache $\mathtt{C}$'s trace semantics $[\![\mathtt{C}]\!]$; and it outputs *true* if $\mathtt{t}$ belongs to the $\mathtt{P}$'s trace semantics and *false* otherwise.

Polca relies on the following helper functions:

- *probeCache* which, given a query $\mathtt{q}$ (i.e., a sequence of memory blocks) and the cache trace semantics $[\![\mathtt{C}]\!]$, accesses all blocks in $\mathtt{q}$ and returns whether the last block produces $\mathtt{Hit}$ or $\mathtt{Miss}$.
- *mapInput* which, given a policy input $\mathtt{ip}$ and a cache content $\mathtt{cc}$, maps $\mathtt{ip}$ to a memory block $\mathtt{b}$. If $\mathtt{ip}$ is $\mathtt{Ln(i)}$, the function returns $\mathtt{cc}[i]$. Otherwise (i.e., $\mathtt{ip}$ is $\mathtt{Evct}$), it returns a block $\mathtt{b}$ not in $\mathtt{cc}$.
- *mapOutput* which, given a cache output $\mathtt{oc}$, a query $\mathtt{q}$, and a cache content $\mathtt{cc}$, maps $\mathtt{oc}$ to the line containing the block that is evicted. If $\mathtt{oc}$ is $\mathtt{Hit}$, the function returns $\bot$. Otherwise (i.e., $\mathtt{oc}$ is $\mathtt{Miss}$), it returns the line $\mathtt{i}$ where the evicted block was stored.
- *findEvicted* which, given a query $\mathtt{q}$, a cache content $\mathtt{cc}$, and the cache trace semantics $[\![\mathtt{C}]\!]$, determines which line has been evicted by the last memory block in $\mathtt{q}$. For that, the function probes the cache with queries $\mathtt{q} \cdot \mathtt{cc}[1], \ldots, \mathtt{q} \cdot \mathtt{cc}[n]$ and determines which block resulted in $\mathtt{Miss}$, i.e., the line that has been evicted by the last block in $\mathtt{q}$.

Algorithm 1 POLCA: A membership oracle for policies

```
1:  function Polca(cc_0, t, [[C]])
2:    cc ← cc_0
3:    for all i = 1, ..., |t| do
4:      let ⟨ip, op⟩ be t[i]
5:      ic[i] ← mapInput(ip, cc)
6:      oc ← probeCache(ic[1 ... i], [[C]])
7:      op' ← mapOutput(oc, ic[1 ... i], cc)
8:      if op' ≠ ⊥ then                    ▷ Update cache content
9:        cc ← cc[op' ↦ ic[i]]
10:     if op ≠ op' then
11:       return false
12:   return true

13: function probeCache(q, [[C]])
14:   k ← |q|
15:   let o be such that ⟨q[1], o[1]⟩ · ... · ⟨q[k], o[k]⟩ ∈ [[C]]
16:   return o[k]

17: function mapInput(ip, cc)
18:   if ip ∈ {Ln(0), ..., Ln(n − 1)} then
19:     let i be such that ip = Ln(i)
20:     return cc[i]
21:   else                                  ▷ ip = Evct
22:     let b ∈ Blocks be such that cc[i] = b for no i
23:     return b

24: function mapOutput(oc, q, [[C]])
25:   if oc = Hit then
26:     return ⊥
27:   else                                  ▷ oc = Miss
28:     return findEvicted(q, cc, [[C]])

29: function findEvicted(q, cc, [[C]])
30:   for all i = 1, ..., n do
31:     if probeCache(q · cc[i], [[C]]) = Miss then
32:       return i
```

We are now ready to describe POLCA in detail. For each pair $\langle \mathtt{ip}, \mathtt{op} \rangle \in \mathtt{t}$, the algorithm maps the input policy symbol $\mathtt{ip}$ to a memory block $\mathtt{b}$ (*mapInput* call at line 5). Then, the algorithm probes the cache under learning to determine the result $\mathtt{oc}$ of accessing $\mathtt{b}$. Next, the cache output $\mathtt{oc}$ is mapped to an output policy symbol $\mathtt{op}'$ (*mapOutput* call at line 7). If $\mathtt{op}$ does not match the computed $\mathtt{op}'$, the algorithm returns *false*; else, the algorithm moves to the next pair of input/output policy symbols, and returns *true* when the sequence is entirely processed.

We remark that the algorithm needs to keep track of the blocks in the cache ($\mathtt{cc}$ variable). For that, it updates the $\mathtt{cc}$ variable after every cache miss (line 9).

The algorithm also keeps track of the sequence of blocks processed so far while processing the trace (through the $\mathtt{ic}$ variable). This is used (line 6) to set the cache $\mathtt{C}$ into the correct state before accessing the new block $\mathtt{ic}[i]$. The sequence is also used to find which cache line was last evicted (*findEvicted* function).

Correctness. Theorem 3.1 states that, given a cache $\mathtt{C}$ with unknown policy $\mathtt{P}$, POLCA provides a sound, complete and terminating oracle for $\mathtt{P}$'s trace semantics.

Theorem 3.1. *Let $\mathtt{C}$ be a cache of associativity n with initial content $\mathtt{cc}_0 \in \mathtt{CC}^n$ and policy $\mathtt{P}$. Then, $[\![\mathtt{P}]\!] = \{\mathtt{t} \in (\mathtt{IP} \times \mathtt{OP})^* \mid \textsc{Polca}(\mathtt{cc}_0, \mathtt{t}, [\![\mathtt{C}]\!]) = \mathit{true}\}$.*

As a corollary to Theorem 3.1, we obtain a language-theoretic relationship between policies and caches. Proposition 3.2 states that, once we fix the initial cache content $\mathtt{cc}_0$ and associativity n, a replacement policy $\mathtt{P}$ uniquely determines the corresponding cache $\mathtt{C}(\mathtt{P}, \mathtt{cc}_0, n)$.

Proposition 3.2. *Given two policies $\mathtt{P}$ and $\mathtt{P}'$ of associativity $n \in \mathbb{N}$ and an initial cache content $\mathtt{cc}_0 \in \mathtt{CC}^n$, then $[\![\mathtt{P}]\!] = [\![\mathtt{P}']\!]$ iff $[\![\mathtt{C}(\mathtt{P}, \mathtt{cc}_0, n)]\!] = [\![\mathtt{C}(\mathtt{P}', \mathtt{cc}_0, n)]\!]$.*

Concretely, Proposition 3.2 provides a theoretical justification for learning only the policy, since knowing the policy is equivalent to knowing the cache behavior.

3.3 Equivalence Queries

Equivalence queries between two Mealy machines M and M' are commonly implemented using *conformance testing*, which relies on a test suite (TS) of membership queries: If there is a membership query in TS on which M and M' disagree, the machines are clearly not equivalent. However, there are Mealy machines M for which there is no finite test suite TS that demonstrates non-equivalence for all M' with $[\![M]\!] \neq [\![M']\!]$. That is, the approximation of equivalence queries using finite membership tests is not complete [27].

In our approach, we hence aim for a weaker notion of completeness due to [27]. Namely, for a parameter m we say that a test suite TS is *m-complete* for a hypothesized policy $\mathtt{H}$, if there is no policy $\mathtt{P}$ with less than m control states and $[\![\mathtt{H}]\!] \neq [\![\mathtt{P}]\!]$, such that both $\mathtt{H}$ and $\mathtt{P}$ agree on the TS. With this, one can obtain the following guarantees for the equivalence test.

Theorem 3.3. *Let $\mathtt{P}$ be an unknown replacement policy, $\mathtt{H}$ a hypothesized policy, and TS an m-complete test suite for $\mathtt{H}$ for some $m \in \mathbb{N}$. If $\mathtt{P}$ and $\mathtt{H}$ agree on all queries of TS then either $[\![\mathtt{H}]\!] = [\![\mathtt{P}]\!]$ or $\mathtt{P}$ has more than m states.*

We rely on existing algorithms, e.g. the Wp-Method [22], for computing m-complete test suites for our hypothesized policy $\mathtt{H}$.

3.4 Tool Implementation

We implement Polca in a Java prototype on top of the LearnLib automata framework v0.14.0 [34]. This allows us to leverage state-of-the-art automata learning algorithms.

To access the cache trace semantics $[\![\mathtt{C}]\!]$, our tool interacts either with a software-simulated cache or with CacheQuery when targeting real hardware.

Concretely:

- for the membership oracle we implement Polca, as described in § 3.2.
- for the equivalence oracle we rely on the Wp-Method [22] for computing test suites for conformance testing, as described in § 3.3. Ideally, one would use a m-complete TS for $\mathtt{H}$, for m as large as possible. Unfortunately the cost of computing m-complete TS grows exponentially with m. To achieve a good trade-off between completeness guarantees and complexity we rely on $(|\mathtt{H}| + k)$-complete TS, for a small constant k which we call the *depth* of the suite.

Our main learning loop uses the k-deep conformance test for finding counterexamples. If the test fails, i.e., a counterexample for the current hypothesis is found, we refine the hypothesis and learning continues. Otherwise, the learning terminates and we output the current hypothesis. At the end, we get the following overall guarantees.

Corollary 3.4. *If our learning approach with Polca, applied to a cache* $\mathtt{C}(\mathtt{P}, \mathtt{cc}_0, n)$*, returns a policy* $\mathtt{P}'$*, then* $[\![\mathtt{P}]\!] = [\![\mathtt{P}']\!]$*, or* $\mathtt{P}$ *has more than* $|\mathtt{P}'| + k$ *states.*

To avoid cumbersome notation, from this point on we use Polca to refer to both our algorithm and to our prototype tool integrating the algorithm with the learning loop.

4 CacheQuery: An Interface to Hardware Memory Caches

In this section we present CacheQuery, a tool for querying silicon CPU caches that frees the user from low-level details like slicing, mapping, virtual-to-physical translation, and timing.

We first describe MemBlockLang, the domain-specific language for specifying inputs to CacheQuery; then we describe CacheQuery's architecture and discuss some of the implementation challenges.

4.1 Domain Specific Language

We design *MemBlockLang* (MBL), a language that facilitates the writing of queries to caches.

A *query* is a sequence of one or more *memory operations*. Each memory operation is specified as a block from a finite, ordered set of blocks `Blocks`, and it is decorated with an optional *tag* from {?, !}. The tag '?' indicates that the access to the block should be profiled [15] to determine whether it results in a cache hit or miss; the tag '!' indicates that the block should be invalidated (e.g., via `clflush`); and no tag means that the block should just be accessed.

MBL features several macros that facilitate writing common query sequences:

- An *expansion* macro '`@`' that produces a sequence of associativity many different blocks in increasing order, starting from the first element. For example, for associativity 8, `@` expands to the sequence of blocks `A B C D E F G H`.
- A *wildcard* macro '`_`' produces associativity many different queries, each one consisting of a different block. As for '`@`', blocks are chosen in alphabetical order. For example, for associativity 8, `_` expands to the set of queries {`A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`}.
- A *concatenation* macro $s_1 \circ s_2$ that concatenates each query in s_1's expansion with each query in s_2's expansion. For instance, `(A B C D)` $\circ$ `(E F)` expands to the query `A B C D E F`.
- An *extension* macro $s_1\,[s_2]$ that takes as input queries s_1 and s_2 and then creates $|s_2|$ many copies of s_1 and extends each of them with a different element of s_2. For example, `(A B C D)[E F]` expands to the set of queries {`A B C D E`, `A B C D F`}.
- A power operator $(s)n$ that repeats a query macro s for n times. For example, `(A B C)3` expands to the query `A B C A B C A B C`.
- A tag over (s_1) or $[s_1]$ applies to every block in s_1. For example, `(A B)?` expands to `A? B?`.

MBL expressions can be given a formal semantics in terms of sets of queries (cf. Appendix A). For the purpose of presentation we omit such a formalization and focus on examples.

Example 4.1. For associativity 4, the query '`@ X _?`' expands to '`(A B C D)` $\circ$ `X` $\circ$ `[A B C D]?`' or, equivalently, to the set of queries {`A B C D X A?`, `A B C D X B?`, `A B C D X C?`, `A B C D X D?`}. This query performs an initial insertion (i.e., fills the cache with blocks `A B C D`), accesses a block `X` not in the cache, and probes all blocks `A`, `B`, `C`, and `D` to determine which one has been replaced after the cache miss caused by `X`. In fact, this query implements the function *findEvicted* in Algorithm 1.

4.2 Architecture

CacheQuery is split into two parts, described next. The backend is implemented in C as a Linux Kernel Module, while the frontend is implemented in Python 3.

Frontend. CacheQuery's frontend expands MBL expressions into sets of queries. The frontend provides two different execution modes: interactive and batch.

- The *interactive* mode provides a REPL shell for executing queries, modifying configuration options, and dynamically choosing the target cache level and set. We use this mode as an interface for the learning algorithm
- The *batch* mode allows to run groups of predefined queries against different cache sets. This becomes useful for

running batteries of tests, which, for instance, allows us to identify fixed leader sets (cf. Appendix B).

Furthermore, the frontend uses LevelDB[3] to cache query responses. This improves performance by avoiding emitting repeated queries to the backend.

Backend. CacheQuery's backend translates queries into sequences of memory accesses, generates the appropriate machine code with the corresponding profiling, executes the code in a low-noise environment, and finally returns traces of hits and misses. We remark that profiling happens at an individual memory access granularity, and supports performance counters, time stamp counter, or counting core cycles, as demanded by the user.

The backend is implemented as a Loadable Kernel Module (LKM). This allows us to use APIs that provide fine-grained control over operating system details like virtual to physical address translation, interrupts, or preemption.

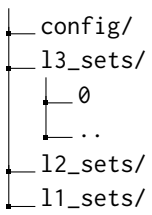


Figure 3. LKM's virtual file system used by CacheQuery

On load, the LKM allocates several pools of memory (one per cache level) and maps each address into its corresponding cache set. The backend provides a virtual system file interface, described in Figure 3, and all the communication is handled through read and write operations over virtual files. Specifically, writing query sequences into the cache set virtual file triggers the address selection and code generation, whereas reading from the cache set virtual file executes the generated code, and returns the sequence of hits and misses.

4.3 Implementation Challenges

In the following we discuss some of the challenges in implementing CacheQuery.

Set Mapping. The first challenge is identifying which memory addresses are mapped into a specific cache set, i.e. which addresses are *congruent* or part of the same *eviction set*. For this, we need to know the number of cache sets, if these sets are virtually or physically indexed, and how the mapping is performed, i.e., which bits of the address are used for the mapping. For most architectures, this information is publicly available [18, 26]. Otherwise, it is possible to infer it[1] or to dynamically compute eviction sets [37]. CacheQuery is completely parametric on the set mapping details. Furthermore, this information needs only to be determined once per micro-architecture.

Cache Filtering. When running queries against a low-level cache, say L3, one needs to make sure that the corresponding memory accesses do not hit higher-level caches such as L1. To this end, CacheQuery automatically evicts every accessed block from higher-level caches. For instance, after accessing a block b in L3, CacheQuery automatically accesses non-interfering eviction sets (i.e. addresses that are congruent with b in L2 and L1, but *not* congruent in L3) to ensure b's eviction from L2 and L1.

Code Generation. MBL expressions are first expanded into sets of queries and then dynamically translated into native code for execution. CacheQuery's code allocation ensures that instructions do not interfere—whenever possible—with the target cache set. This allows to target different sets dynamically. Furthermore, to support arbitrary queries, CacheQuery uses immediate load operations serialized with fences, rather than the more usual pointer chasing technique [35].

Interferences. To minimize noise during the execution of memory accesses CacheQuery can temporary disable hardware prefetchers, hyper-threading, frequency scaling, and other cores.

4.4 Limitations

Currently, CacheQuery only supports Intel CPUs. While several parts of our implementation are architecture-agnostic, adding support for other architectures, such as AMD and ARM, will require some effort.

Likewise, CacheQuery currently runs on top of a fully-fledged Linux kernel. While facilitating development, this adds unnecessary complexity and non-determinism. Using a custom unikernel could provide a better suited environment for our experiments.

5 Explaining policies

In this section, we present our approach for synthesizing explanations of replacement policies in the form of high-level programs, starting from our automata models..

Policy explanations. We explain replacement policies in terms of four simpler rules: (a) a *promotion rule* describing how the control state is updated whenever there is a cache hit, (b) an *eviction rule* describing how to select the cache line to evict, (c) an *insertion rule* describing how the control state is updated whenever there is a cache miss, and (d) a *normalization rule* describing how to normalize the control state before or after a hit or a miss[4]. We borrow these terms from policy proposals from the hardware community [20].

[3]LevelDB is a fast string key-value storage library: https://github.com/google/leveldb.

[4]Normalization is used in some policies to preserve control state invariants. For example, MRU updates the control state after a hit if all lines have age 0.

Explanation template. The main component of our synthesis approach is a program-level template for explanations, which is defined in terms of promotion, eviction, insertion, and normalization rules:

```
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⟩
```

The template models control states as arrays mapping cache lines to their so-called ages. The concrete age values (of type `Nat`) are left as holes to be instantiated during the synthesis. Additionally, the template consists of two functions:

The function `hit` describes how the control state is updated whenever there is a cache hit. The function takes as input a control state `state` and a cache line `line`, updates the control state using the promotion rule, normalizes, and returns the new state.

In contrast, the function `miss` modifies the control state in case of a cache miss. The function takes as input a control state `state`, normalizes it, detects the cache line `idx` to evict using the eviction rule, updates the age of the evicted line using the insertion rule, and finally normalizes again the ages.

Generators. Our template specifies several *generators* for the rules. Generators are programs with holes that can be instantiated during synthesis. Each of the holes can be instantiated with expressions generated from specific grammars, which constraint the synthesis' search space. To illustrate, this is a generator for the promotion rule:

```
promote(state,pos)::States×Lines→States
    States final = state
    if(??{boolExpr(state[pos])}) // Update line
        final[pos] = ??{natExpr(state[pos])}
    for(i in Lines) // Update rest
        if(i ≠ line ∧
            ??{boolExpr(state[pos],state[i])})
            final[i] = ??{natExpr(state[i])}
    return final
```

The generator takes as input a control state and a cache line and returns the updated control state. The updated state is derived by first conditionally modifying the age of the accessed line and later iterating over the remaining cache line and conditionally updating them.

All conditions and update expressions are encoded as holes that refer to template variables. For instance, the hole `??{`*boolExpr*`(state[pos],state[i])}` can be instantiated with a conjunction of equalities and inequalities that refer to natural numbers and to `state[pos]` and `state[i]`, whereas `??{`*natExpr*`(state[i])}` can be instantiated with an arbitrary sequence of additions and subtractions that refer to natural numbers and to `state[i]`.

In general, our grammar generators can refer to constants, line indices (like `pos` and `i` in the `promote` example), and ages (like `state[pos]` in the `promote` example). We also implement a simplified version of our generators that (1) fix the `normalize` rule to the identity function, and (2) restrict the grammar to only refer to constants and ages.

Constraints. Given a policy $\mathbf{P}$, we construct a formula $\varphi_{\mathbf{P}}$ encoding $\mathbf{P}$'s transition relation $\longrightarrow$ in terms of our template's `hit` and `miss` functions. In our encoding, we associate $\mathbf{P}$'s control states with logical variables. Concretely, we map each control state $\mathbf{cs_i}$ in $\mathbf{P}$ to a corresponding variable $\mathtt{cs}_i$. The constraint $\varphi_{\mathbf{P}}$ is defined as follows, where $\mathbf{cs_1}, \ldots, \mathbf{cs_m}$ are all P's control states:

$$\exists \mathtt{cs}_1, \ldots, \mathtt{cs}_m. \bigwedge_{1 \le i,j \le |P| \wedge i \neq j} \mathtt{cs}_i \neq \mathtt{cs}_j \wedge$$
$$\bigwedge_{\delta(\mathbf{cs_k}, \mathbf{Ln(i)}) = \mathbf{cs_l}} \mathtt{hit}(\mathtt{cs}_k, i) = \mathtt{cs}_l \wedge$$
$$\bigwedge_{\substack{\delta(\mathbf{cs_k}, \mathbf{Evct}) = \mathbf{cs_l} \wedge \\ \lambda(\mathbf{cs_k}, \mathbf{Evct}) = \mathbf{i}}} \mathtt{miss}(\mathtt{cs}_k) = \langle \mathtt{cs}_l, i \rangle$$

The existential quantification and the first conjunct ensure that there are m concrete control states (one per control state in $\mathbf{P}$). The second and third conjuncts ensure that the `hit` and `miss` functions behave as specified by $\mathbf{P}$.

Synthesis. To synthesize an explanation for a learned policy $\mathbf{P}$, we query a syntax-guided synthesis solver for an instance of our template that satisfies the constraint $\varphi_{\mathbf{P}}$. The solver, then, either returns a program *Prg* that instantiates the holes in the template in a way that satisfy $\varphi_{\mathbf{P}}$, or terminates without finding a model (in case our template cannot represent $\mathbf{P}$).

Example 5.1. Let $\mathbf{P}$ be the LRU policy with associativity 2 given in Example 2.2. The constraint $\varphi_{\mathbf{P}}$ is as follows:

$$\exists\, \mathtt{cs}_0, \mathtt{cs}_1.\ \mathtt{cs}_0 \neq \mathtt{cs}_1 \wedge \mathtt{hit}(\mathtt{cs}_0, 1) = \mathtt{cs}_0 \wedge$$
$$\mathtt{hit}(\mathtt{cs}_0, 0) = \mathtt{cs}_1 \wedge \mathtt{hit}(\mathtt{cs}_1, 0) = \mathtt{cs}_1 \wedge \mathtt{hit}(\mathtt{cs}_1, 1) = \mathtt{cs}_0 \wedge$$
$$\mathtt{miss}(\mathtt{cs}_0) = \langle \mathtt{cs}_1, 0 \rangle \wedge \mathtt{miss}(\mathtt{cs}_1) = \langle \mathtt{cs}_0, 1 \rangle\ .$$

Whenever the solver synthesizes a program that satisfy the given constraints, we can lift the correctness guarantee of our approach also to the synthesized program *Prg*. Indeed, the solver's soundness, the template's determinism, and the constraint $\varphi_{\mathbf{P}}$ ensure that *Prg* behaves exactly as the learned policy $\mathbf{P}$ on the concrete control states.

6 Case Study: Learning from Software-Simulated Caches

This section reports on a case study where we use Polca to learn well-known replacement policies from software-simulated caches implementing such policies.

This case study's goals is to evaluate Polca's efficiency and scalability across different classes of replacement policies, without the overhead introduced by interacting with real hardware. This case study also provides a basis for comparing Polca with prior approaches [1, 32].

Setup. We implemented software-simulated caches (parametric in the cache's associativity) for 7 commonly used replacement policies: First In First Out (FIFO), Least Recently Used (LRU), Pseudo-LRU (PLRU) [14], Most Recently Used (MRU) [25], LRU Insertion Policy (LIP) [30], and HP and FP variants of Static Re-reference Interval Prediction (SRRIP) [20] with 4 ages. We simulate the policies with associativity ranging from 2 to 16.[5] For each policy and associativity, we use Polca to learn the policy with a timeout of 36 hours. We record the time needed to learn the automaton and the learned automaton's number of states. In our experiments, we set the test suite depth k to 1 (cf. § 3.4), which proves sufficient for discovering counterexamples.

Results. Table 2 reports the time taken by Polca to learn the policies and the number of states of the resulting automata. We highlight the following:

- Except for FIFO, the learning time grows roughly exponentially with associativity. Polca learns FIFO and PLRU up to associativity 16.
- Prior approaches for permutation-based policies [1] can learn only FIFO, LRU, and PLRU, from our experimental setup. In contrast, Polca learns policies such like MRU, LIP, SRRIP-HP, and SRRIP-FP (up to associativities 12, 6, 6, and 6 respectively).
- Prior general purpose approaches [32] learn MRU only up to associativity 5 and timeout after 72 hours for larger associativities. In contrast, Polca learns MRU up to associativity 12 and takes 600 milliseconds for associativity 6.

Alternative approaches exist that leverage different heuristics, like random walks, for a deeper counterexample exploration. These approaches generally enable faster hypothesis refinement, and hence better performance. However, we opted for a default and deterministic setup, and leave a more thorough performance evaluation for future work.

Platform. We run all experiments on a Linux virtual machine (kernel 4.9.0-8-amd64) with Debian 9.0, Java OpenJDK v1.8.0_222, running on a Xeon Gold 6154 CPU (with 72 virtual cores), and 64 GB of RAM. We execute the experiments in parallel using a single virtual core for each policy.

[5] Some policies constraint the possible associativities. For instance, PLRU policies are well-defined only for associativities that are powers of 2.

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

Table 2. Learning policies from software-simulated caches (with 36 hours timeout). We omit FIFO's intermediate results.

7 Case Study: Learning from Hardware

This section reports on a case study where we use Polca and CacheQuery to learn policies from real hardware. This case study's goals are (1) to determine whether Polca can learn policies directly from hardware using CacheQuery as an interface, and (2) to understand the additional challenges involved with learning policies from hardware.

CPU	Cache level	Assoc.	Slices	Sets per slice
i7-4790 (Haswell)	L1	8	1	64
	L2	8	1	512
	L3	16	4	2048
i5-6500 (Skylake)	L1	8	1	64
	L2	4	1	1024
	L3	12	8	1024
i7-8550U (Kaby Lake)	L1	8	1	64
	L2	4	1	1024
	L3	16	8	1024

Table 3. Processors' specifications [17, 18, 26].

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

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).

7.1 Setup

We analyze the L1, L2, and L3 caches of the Intel i7-4790 (Haswell), i5-6500 (Skylake), and i7-8550U (Kaby Lake) processors (cf. Table 3 for the specifications). For each processor and cache level, we use Polca to learn the policy while using CacheQuery as an interface to the hardware cache.

In our experiments, we make the following simplifications:

- For some policies, Polca does not scale to the large associativities used in L3 caches. To overcome this, we use Intel's CAT technology [16] to virtually reduce L3 associativity to 4 for the i5-6500 (Skylake) and i7-8550U (Kaby Lake) processors. CAT is not supported by i7-4790 (Haswell).
- Modern L3 caches often implement adaptive replacement policies [20, 30, 31], where separate groups of *leader* cache sets implement distinct replacement policies and the remaining *follower* sets switch between these policies dynamically. We *only* learn leader sets' policies. Appendix B in [6] details how we identify leader sets with CacheQuery.
- Our membership oracle Polca (Algorithm 1) accesses the cache's trace semantics, where all traces are executed starting from a fixed initial state. In general, a cache can be reset to a fixed initial state by invalidating the entire cache content (with `clflush` or `wbinvd` instructions) and accessing associativity-many different blocks, i.e., executing the '`@`' MBL query. However, this is not always the case. For instance, resetting i7-4740's L1 cache requires accessing associativity-many blocks *twice* since the insertion order in an invalidated cache seems arbitrary. To address this, we manually identified[6] reset sequences for each cache (cf. Table 4).

Platform. We run CacheQuery on three different machines equipped with the three processors. Additionally, we run Polca on the same platform described in § 6. The communication between Polca and CacheQuery happens over SSH in a local network.

7.2 Results

Learned policies. Table 4 summarizes the learned policies. We highlight the following findings:

- For all processors' L1 caches and for Haswell's L2 cache, Polca learns the same policy, that is, a tree-based PLRU policy. We identified this policy by checking its equivalence to a manually implemented PLRU automaton. This result confirms common folklore that Intel processors implement PLRU in their L1 policy.
- For Skylake's and Kaby Lake's L2 caches, Polca learns a previously undocumented policy. This policy is indicated as *New1* in Table 4, and we further discuss it in § 8.
- For Skylake's and Kaby Lake's L3 caches, Polca learns a previously undocumented policy for the leader sets. This policy is indicated as *New2* in Table 4, and we further discuss it in § 8. Additionally, we confirm the mapping of Skylake's leader sets [37], and discover that Kaby Lake follows the mapping.
- For Haswell's L3 cache, Polca cannot learn the replacement policy. This is due to (1) i7-4790 not supporting CAT, and (2) one of the leader sets showing a non-deterministic behavior.

Cost of learning from hardware. Learning policies from hardware caches comes with a significant overhead when compared with learning from software-simulated caches. This is due to (1) communication overhead between Polca and CacheQuery, and (2) CacheQuery overhead for code generation and profiling. We separately analyze the impact of (1) and (2).

For (1), we compare the time needed to learn a PLRU policy with associativity 8 from a software-simulated cache and from CacheQuery where every MBL query hits the LevelDB cache (i.e., the results of the MBL queries on the

[6] We believe that reset sequences could be learned automatically by extending our framework with an explicit `invalid` command. We leave this as future work.

real hardware have been precomputed). Learning from the software-simulated cache takes 1.46 s (cf. Table 2), while learning from CacheQuery takes 2247 s, resulting in a 1500x overhead.

For (2), we measure the time taken to execute a single MBL query '`@ M _?`' across cache levels. The averaged query execution time (across 100 executions on the i5-6500 Skylake processor) is 16 ms on L1, 11 ms on L2, and 20 ms on L3. We remark that learning the PLRU policy with associativity 8 requires more than 50'000 MBL queries.

8 Case Study: Synthesizing Explanations

This section reports on a case study where we use the synthesis approach from § 5 to derive policy explanations for the automata learned in § 6–7. This case study's goals are (1) to evaluate if our approach can explain the replacement policies learned in §§ 6–7, and (2) to determine whether the synthesized explanations can help in understanding previously undocumented policies.

8.1 Setup

We encoded our template (and all rules generators) from § 5 in Sketch [33]. We use Sketch to synthesize explanations for all the policies from § 6 (i.e., FIFO, LRU, PLRU, MRU, LIP, SRRIP-HP, and SRRIP-IP) and for the undocumented policies *New1* and *New2* from § 7. In our experiments, we fix the associativity to 4.

For all policies, we synthesize explanations using Sketch[7], and record the time needed to synthesize an explanation that satisfies our constraints. During synthesis, we bound both the size of natural numbers and the recursion depth of our grammar generators. For associativity 4, we choose a size bound of 4 and recursion depth bound of 2. We explore the synthesis space incrementally until we find a solution or the space is exhausted. For each policy, we first try synthesizing an explanation using the simplified template from § 6 (which we refer to as *Simple* template), and if we cannot synthesize a solution, then we try using the more general template from § 6 (which we refer to as *Extended* template).

Platform. We run the experiments on the same platform as in § 6 and use Sketch v1.7.5, with a single thread.

8.2 Results

Table 5 summarizes the results of our synthesis approach. We highlight the following findings:

1. Our approach successfully explains the FIFO, LRU, and LIP policies using the *Simple* template in less than 5 s.

2. Our approach synthesize explanations for the MRU, SRRIP-HP, SRRIP-FP, *New1*, and *New2* policies using the *Extended* template. The synthesis time varies (from ~40 s for MRU to ~4.5 days for SRRIP-HP), but it is roughly correlated with the number of states.

3. Our approach cannot synthesize an explanation for the PLRU policy using the provided templates. The reason for this is two-fold. First, PLRU control state is global whereas our template encodes a local control state with one age per cache line. Second, encoding PLRU requires bit-wise operators, which are currently not supported by our template's grammar generators.

Policy	States	Template	Execution Time
FIFO	4	*Simple*	0 h 0 m 0.18 s
LRU	24	*Simple*	0 h 0 m 0.81 s
PLRU	8	—	—
LIP	24	*Simple*	0 h 0 m 4.36 s
MRU	14	*Extended*	0 h 0 m 39.80 s
SRRIP-HP	178	*Extended*	105 h 28 m 30 s
SRRIP-FP	256	*Extended*	48 h 30 m 25 s
New1	160	*Extended*	9 h 36 m 9 s
New2	175	*Extended*	26 h 4 m 22 s

Table 5. Synthesizing explanations for policies (of associativity 4). In the *Simple* template, `normalize` is fixed to the identity function and the grammar for expressions is simpler. In contrast, the *Extended* template supports the `normalize` rule and has a more expressive expression grammar.

Explaining New1 and New2. Sketch successfully synthesize explanations for the previously undocumented policies *New1* and *New2* from § 7. Below we provide a high-level description of the policies. We include the complete synthesized programs in Appendix C in [6].

The *New1* policy is defined by:

- The initial control state is $\{3, 3, 3, 0\}$.
- **Promote:** Set the accessed line's age to 0.
- **Evict:** Select the first line, starting from left, whose age is 3.
- **Insert:** Set the evicted line's age to 1.
- **Normalize:** After a hit or a miss, while there is no line with age 3, increase the age of all lines by 1 except for the just accessed/evicted line.

The *New2* policy is defined by:

- The initial control state is $\{3, 3, 3, 3\}$.
- **Promote:** If the accessed line has age 1 set it to 0, otherwise set it to 1.
- **Evict:** Select the first line, starting from left, whose age is 3.
- **Insert:** Set the evicted line's age to 1.
- **Normalize:** After a hit or miss, while there is no line with age 3, increase all lines by 1.

In contrast to the automata models, our high-level representation allow us to compare the previously undocumented

[7] Sketch uses a random seed to explore the search space. Hence, Sketch might synthesize different explanations that satisfy the constraints. In our experiments, we fix the seed to `--slv-seed 1337`.

policies with known ones. Concretely, both *New1* and *New2* are variants of the SRRIP-HP policy, defined in [20]. The main difference appears in the normalization rule, where SRRIP-HP normalizes the ages (by increasing all ages by 1 while there is no line with age 3) only before a miss.

9 Related Work

Model Learning. For related work on automata learning techniques for black-box systems we refer the interested reader to Vaandrager's survey paper [36].

Reverse-engineering cache policies. Abel and Reineke [1] design an efficient algorithm for learning permutation base replacement policies, a class of policies that include LRU, PLRU, and FIFO. Moreover, they use an adhoc approach to reverse engineer two variants of PLRU that employ randomization [2].

Guillem Rueda's master thesis [32] studies how register automata learning can serve to learn a broader class of replacement policies, in comparison to permutation based policies, including MRU. This is an interesting and novel approach, however, their method does not scale in practice (cf. § 6).

Wong [39] first notices the use of adaptive policies in Intel's Ivy Bridge, and tries to identify the new implemented policies guided by recent papers [20, 31], without complete success.

In concurrent work, Abel and Reineke extend nanoBench [3, 4] to reverse engineer cache replacement policies. In contrast to our approach, they proceed by producing random sequences and comparing the results from hardware against a pool of ~300 software-simulated caches. While this approach is less general and the results lack correctness guarantees, in practice, it proves highly efficient and accurate. In fact, we are able to validate several of their findings.

Security. Rowhammer.js [11] tests thousands of eviction strategies, memory access patterns with high eviction rate, in order to identify efficient strategies to mount a rowhammer attack from a web browser. Recent attacks [12, 23, 40] show how detailed knowledge about the replacement policy state can leak information, in contrast to the common content based leak. Similarly, Briongos et al. [7] exploit a new cache side-channel attack leveraging changes in the policy state to bypass mechanisms based on monitoring of cache misses. For this, they attempt to explain the behavior of the replacement policy on several modern Intel CPUs. While their description is not completely accurate, it results sufficient for their attack.

Detailed models about the cache policy, as the ones we provide, can enable to systematically compute optimal eviction strategies, and to unveil new sophisticated cache attacks.

Custom kernels. Recent research projects have developed custom kernels and hypervisors for specialized tasks that require extremely high performance, precise measurements, or access to privileged modes. These environments provide complete control over the hardware improving testing and reproducibility. Some examples include angryOS [24], which has been used for reverse engineering microcode, or Sushi Roll [13], a highly deterministic kernel, initially designed for fuzzing, converted into a cycle-by-cycle CPU introspection tool.

Implementing interfaces as CacheQuery on a custom kernel can provide a better environment for high performance and predictability, ultimately enabling the use of learning methods for other undocumented micro-architectural components, like prefetchers, branch predictors, or data buffers.

10 Conclusions

We presented a practical end-to-end solution for learning hardware cache replacement policies. In our experiments we were successful in inferring human-readable descriptions of cache replacement policies used in recent Intel processors, including 2 previously undocumented policies.

Our approach relies on two contributions that enable us to tackle the problem using off-the-shelf techniques for automata learning and program synthesis: (1) CacheQuery, a tool that provides a clean interface for interacting with hardware memory caches, and (2) Polca, an algorithm that provides a direct interface to the replacement policy by abstracting from the cache content.

Both our contributions are independent and ready to use in alternative workflows, such as advanced learning approaches [9, 21] or manual analysis.

References

[1] Andreas Abel and Jan Reineke. 2013. Measurement-based modeling of the cache replacement policy. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 65–74. https://publications.cispa.saarland/596/

[2] Andreas Abel and Jan Reineke. 2014. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. 141–142. https://doi.org/10.1109/ISPASS.2014.6844475

[3] Andreas Abel and Jan Reineke. 2019. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems.

[4] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, 673–686. https://doi.org/10.1145/3297858.3304062

[5] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[6] Anonymized. 2019. Appendix in Supplementary Material.

[7] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. 2019. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. *CoRR* abs/1904.06278 (2019). http://arxiv.org/abs/1904.06278

[8] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side

channel detection and mitigation. In *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*. IEEE, 0.

[9] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2018. *Extending Automata Learning to Extended Finite State Machines*. Springer International Publishing, 149–177. https://doi.org/10.1007/978-3-319-96562-8_6

[10] Pablo CaÃśones, Boris Köpf, and Jan Reineke. 2019. On the Incomparability of Cache Algorithms in Terms of Timing Leakage. *Logical Methods in Computer Science* Volume 15, Issue 1 (2019). https://doi.org/10.23638/LMCS-15(1:21)2019

[11] Daniel Gruss and Clémentine Maurice and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*. Springer.

[12] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* 18, 1 (2015), 4:1–4:32.

[13] Brandom Falk. 2019. Sushi Roll: A CPU research kernel with minimal noise for cycle-by-cycle micro-architectural introspection. https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html

[14] Jim Handy. 1993. *The Cache Memory Book*. Academic Press Professional, Inc., San Diego, CA, USA.

[15] Intel. 2010. How to Benchmark Code Execution Times on IntelÂő IA-32 and IA-64 Instruction Set Architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[16] Intel. 2017. Are Noisy Neighbors in Your Data Center Keeping You Up at Night. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-rdt-infrastructure-paper.pdf

[17] Intel. 2018. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[18] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *In Proceedings of the 18th EUROMICRO Conference on Digital System Design*.

[19] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib. In *Computer Aided Verification*. Springer.

[20] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (2010), 60–71. https://doi.org/10.1145/1816038.1815971

[21] Ali Khalili and Armando Tacchella. 2014. Learning Nondeterministic Mealy Machines. In *The 12th International Conference on Grammatical Inference (Proceedings of Machine Learning Research)*, Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka (Eds.), Vol. 34. PMLR, Kyoto, Japan, 109–123.

[22] Fujiwara Bochmann Khendek, S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. 1991. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering* 17 (1991), 591–603.

[23] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. 974–987. https://doi.org/10.1109/MICRO.2018.00083

[24] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. 2017. Reverse Engineering x86 Processor Microcode. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, 1163–1180. http://dl.acm.org/citation.cfm?id=3241189.3241280

[25] Adam Malamy, Rajiv Patel N., and Norma M. Hayes. 1992. Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature. https://patents.google.com/patent/US5353425A/en US5353425A.

[26] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[27] Joshua Moerman. 2019. *Nominal Techniques and Black Box Testing for Automata Learning*. Ph.D. Dissertation. Radboud University.

[28] Oliver Niese. 2003. *An Integrated Approach to Testing Complex Systems*. Ph.D. Dissertation. Universität Dortmund.

[29] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[30] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, 381–391. https://doi.org/10.1145/1250662.1250709

[31] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. *SIGARCH Comput. Archit. News* 34, 2 (2006), 167–178. https://doi.org/10.1145/1150019.1136501

[32] Guillem Rueda. 2013. Learning Cache Replacement Policies using Register Automata. Master's thesis, Uppsala University, Department of Information Technology.

[33] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 4–13.

[34] Bernhard Steffen, Falk Howar, and Maik Merten. 2011. *Introduction to Active Automata Learning from a Practical Perspective*. Springer Berlin Heidelberg, 256–296. https://doi.org/10.1007/978-3-642-21455-4_8

[35] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (Jan. 2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y

[36] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (2017), 86–95. https://doi.org/10.1145/2967606

[37] Pepe Vila, Boris Köpf, and Jose Morales. 2019. Theory and Practice of Finding Eviction Sets. In *40th IEEE Symposium on Security and Privacy (S&P '19)*. IEEE, 695–710.

[38] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008), 36:1–36:53.

[39] Henry Wong. 2013. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

[40] Wenjie Xiong and Jakub Szefer. 2019. Leaking Information Through Cache LRU States. *CoRR* abs/1905.08348 (2019). http://arxiv.org/abs/1905.08348

Basic Types

(Blocks)	b	$\in$	$\{a_1, a_1, \ldots, a_m\}$
(Tags)	t	$\in$	$\{?, !\}$
(Numbers)	k	$\in$	$\mathbb{N}$

Syntax

(Queries)	q	$:=$	$\varepsilon \mid b \mid (b)t \mid q_1 \cdot q_2$
(Expressions)	s	$:=$	$(q) \mid \{q_1, \ldots, q_l\} \mid @ \mid \_ \mid (s)t \mid$ $s_1 \circ s_2 \mid (s_1)[s_2] \mid (s)k$

Figure 4. MBL syntax

A MemBlockLang syntax and semantics

Here, we present the syntax and semantics of the MBL language. In the following, we assume given an ordered set of blocks $\mathbb{B} = \{a_1, a_1, \ldots, a_m\}$ and a value $n \in \mathbb{N}$ representing the cache's associativity such that $n < m$.

Syntax. The syntax of MBL is given in Figure 4.

Semantics. The semantics of an MBL expression s consists of a set of queries $[\![s]\!]$ and it is defined as follows:

- $[\![q]\!] = \{q\}$ where q is a query.
- $[\![\{q_1, \ldots, q_l\}]\!] = \{q_1, \ldots, q_l\}$ where $\{q_1, \ldots, q_l\}$ is a set of queries.
- $[\![@]\!] = \{a_1 \cdot a_2 \cdot \ldots \cdot a_n\}$.
- $[\![\_]\!] = \{a_1, a_2, \ldots, a_n\}$.
- $[\![s?]\!] = \{a_1? \cdot a_2? \cdot \ldots \cdot a_k? \mid a_1 \cdot a_2 \cdot \ldots \cdot a_k \in [\![s]\!]\}$ where s is an MBL expression such that all queries in $[\![s]\!]$ do not have tags.
- $[\![s!]\!] = \{a_1! \cdot a_2! \cdot \ldots \cdot a_k! \mid a_1 \cdot a_2 \cdot \ldots \cdot a_k \in [\![s]\!]\}$ where s is an MBL expression such that all queries in $[\![s]\!]$ do not have tags.
- $[\![s_1 \circ s_2]\!] = \{s \cdot s' \mid s \in [\![s_1]\!] \wedge s' \in [\![s_2]\!]\}$ where s_1, s_2 are MBL expressions.
- $[\![s_1[s_2]]\!] = \{s \cdot a \mid s \in [\![s_1]\!] \wedge \exists s_2 \in [\![s_2]\!], i \in \mathbb{N}.\ s_2[i] = a\}$ where s_1, s_2 are MBL expressions.
- $[\![(s)k]\!] = \{s_1 \cdot s_2 \cdot \ldots \cdot s_k \mid \bigwedge_{1 \le i \le k} s_i \in [\![s]\!]\}$, where s is an MBL expression and $k \in \mathbb{N}$.
- $[\![(s)]\!] = [\![s]\!]$, where s is an MBL expression.

B Adaptive Policies and Leader Sets

Henry Wong [39] identifies an adaptive L3 cache on Intel's Ivy Bridge processors and describes heuristics for detecting the existence of leader sets.

In a similar way, we use CacheQuery to run several thrashing queries (i.e., access patterns with a working set that does not fit into the cache and degenerates performance) on a per set basis, obtaining the following results:

- Haswell i7-4790: sets $512 - 575$ in slice 0 implement a fixed policy susceptible to thrashing. Sets $768 - 831$ in slice 0 implement a fixed thrash resistant policy (that seems to be not deterministic). In contrast, the rest of the cache sets follow the policy producing less misses.
- Skylake i5-6500: cache sets whose indexes satisfy $((((\texttt{set}\ \&\ \texttt{0x3e0}) \gg 5) \oplus (\texttt{set}\ \&\ \texttt{0x1f})) = \texttt{0x0}) \wedge ((\texttt{set}\ \&\ \texttt{0x2}) = \texttt{0x0})$ implement a fixed policy susceptible to thrashing (i.e., policy *New2*). The rest seem to use an adaptive policy that behaves in a non-deterministic way.
- Kaby Lake i7-8550U: we observe the same behavior and set selection than on Skylake i5-6500.

On Haswell, we confirm previous results reported in [7]. However, we remark that leader sets are only present in slice 0. It is also worth mentioning that the ranges seem to be selected by comparing the index bits with some fixed constants: $((\texttt{set}\ \&\ \texttt{0x7c0}) \gg 6) = \texttt{0x8}$ and $((\texttt{set}\ \&\ \texttt{0x7c0}) \gg 6) = \texttt{0xc}$, respectively.

Previous work [37] identified sets with a fixed policy on Skylake, and argued that leader set influence did *not* cross slices. We complete this knowledge, including the description for Kaby Lake, and expose that adaptivity *has* effects across different slices, i.e. a single cache set leader producing lots of hits can affect all the follower sets in the cache.

We also report the following observations regarding the adaptive policy implemented in the rest of the L3 cache sets in Skylake and Kaby Lake:

- First, we observe another group of sets —those whose indexes satisfy ((((set & 0x3e0) ≫ 5) ⊕ (set & 0x1f)) = 0x1f) ∧ ((set & 0x2) = 0x2)— that change at a different rate than the majority.
- Second, it is possible to control the adaptive policies in 2 ways, by only interacting with the thrash-vulnerable fixed sets: (1) heavily thrashing the fixed sets makes the adaptive policy become more thrash resistant, i.e., `@ M a M?` always produces a miss; (2) continuously producing hits on the fixed sets makes them tend towards the *New2* policy.

We remark that this interaction needs to happen concurrently, which might indicate a small counter refereeing the set dueling mechanism. If the unknown adaptive policy is inspired by DRRIP [20], it could behave deterministically when completely saturated. However, we have not yet been able to learn it.

Interestingly, the set selection uncovered for Skylake and Kaby Lake processors is very similar to that in [30], which augments our confidence in the abovementioned explanation.

C Previously undocumented policies

Figure 5 shows a cleaned-up version, with minor variable renaming and layout adjustments, of the synthesized explanations describing the 2 previously undocumented cache replacement policies from § 7.

We include the complete code for all templates, constrains, and solutions, in our repository: https://github.com/cgvwzq/polca/.

```
// We only display initial state
int[4] s0 = {3,3,3,0};

int[4]hitState (int[4] state, int pos)
  int[4] final = state; bit found = 0;
  // Promotion
  final[pos] = 0;
  // Normalization
  for(int j = 0; j < 4; j = j + 1)
    // Check if there is block to replace
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        if(found == 0 && final[i] == 3)
            found = 1;
   // If not, increase all ages
   // except for promoted one
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        if(i != pos)
          final[i] = final[i] + 1;
  return final;

int missIdx (int[4] state)
  // Replace first block with age 3
  for(int i = 0; i < 4; i = i + 1)
    if(state[i] == 3)
      return i;

int[4] missState (int[4] state)
  int[4] final = state; bit found = 0;
  // Insertion
  int replace = missIdx(state);
  final[replace] = 1;
  // Normalization
  for(int j = 0; j < 4; j = j + 1)
    // Check if there is block to replace
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        if(found == 0 && final[i] == 3)
          found = 1;
   // If not, increase all ages
   // except for replaced one
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        if(replace != i)
          final[i] = final[i] + 1;
  return final;
```

(a) Sketch solution for *New1* undocumented policy.

```
// We only display initial state
int[4] s0 = {3,3,3,3};

int[4] hitState (int[4] state)
  int[4] final = state; bit found = 0;
  // Promotion
  if(final[pos] < 2 && final[pos] == 1)
    final[pos] = 0;
  else if (state[pos] > 1)
    final[pos] = 1;
  // Normalization
  for(int j = 0; j < 4; j = j + 1)
    // Check if there is block to replace
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        if(found == 0 && final[i] == 3)
          found = 1;
    // If not, increase all ages
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        final[i] = final[i] + 1;
  return final;

int missIdx (int[4] state)
  // Replace first block with age 3
  for(int i = 0; i < 4; i = i + 1)
    if(state[i] == 3)
      return i;

int[4]missState (int[4] state)
  int[4] final = state; bit found = 0;
  // Insertion
  int replace = missIdx(state);
  final[replace] = 1;
  // Normalization
  for(int j = 0; j < 4; j = j + 1)
    // Check if here is a block to replace
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        if(found == 0 && final[i] == 3)
          found = 1;
    // If not, increase all ages
    if(found == 0)
      for(int i = 0; i < 4; i = i + 1)
        final[i] = final[i] + 1;
  return final;
```

(b) Sketch solution for *New2* undocumented policy.

Figure 5. Synthesized high-level programs for previously undocumented replacement policies using the *Extended* template.