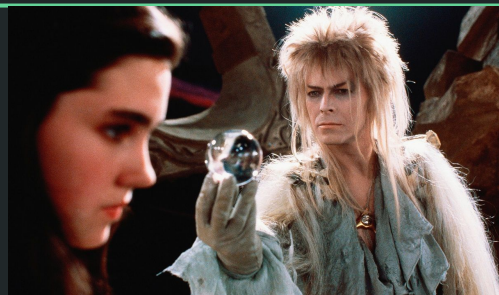




# The **80s** never died: Automata theory for reversing modern CPUs



RootedCON - March 2020



[github.com/cgvwzq](https://github.com/cgvwzq)

# About me

I'm Pepe Vila (a.k.a. cgvwzq)

PhD student at the IMDEA Software Institute

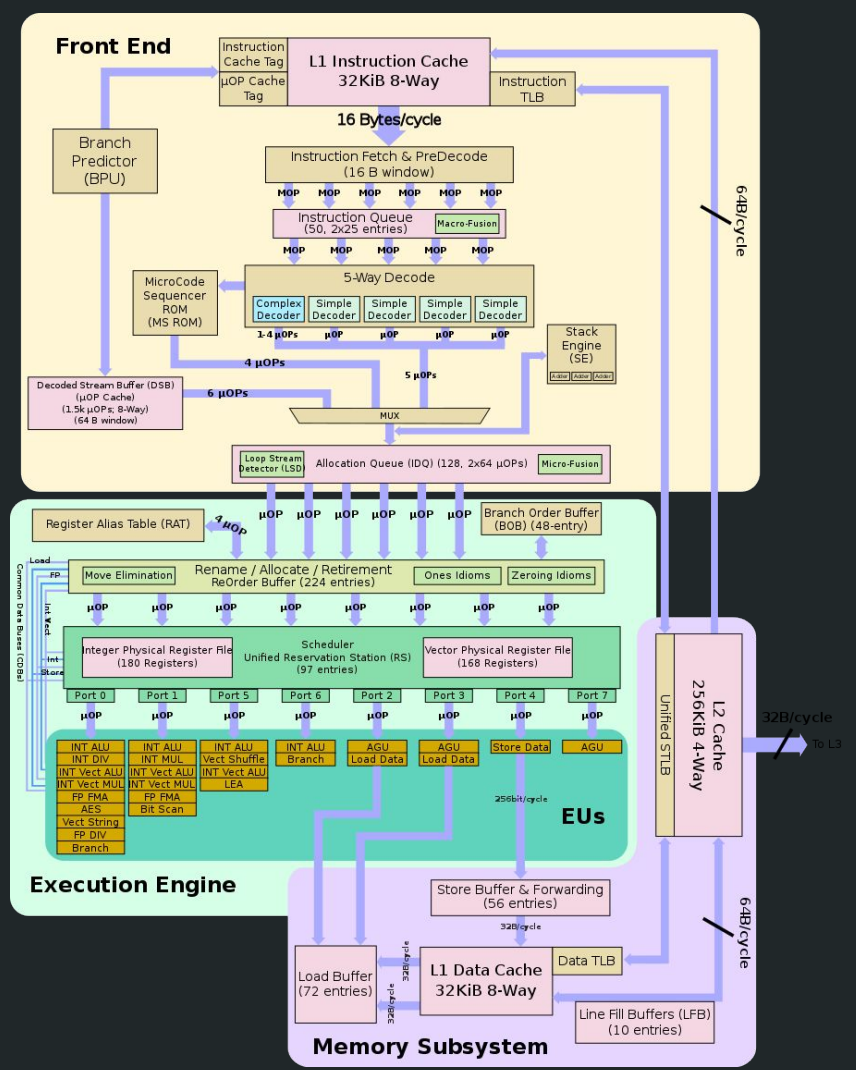
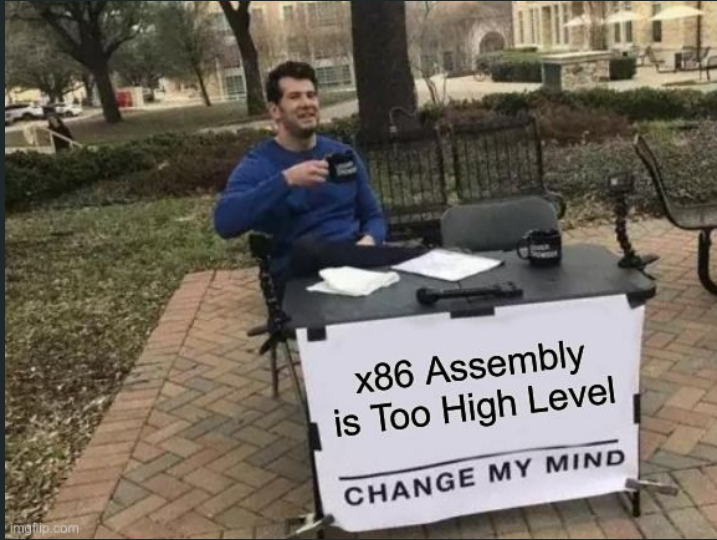
Worked as security consultant and pentester

Intern at Facebook and Microsoft Research

I used to mess with browsers and JavaScript...

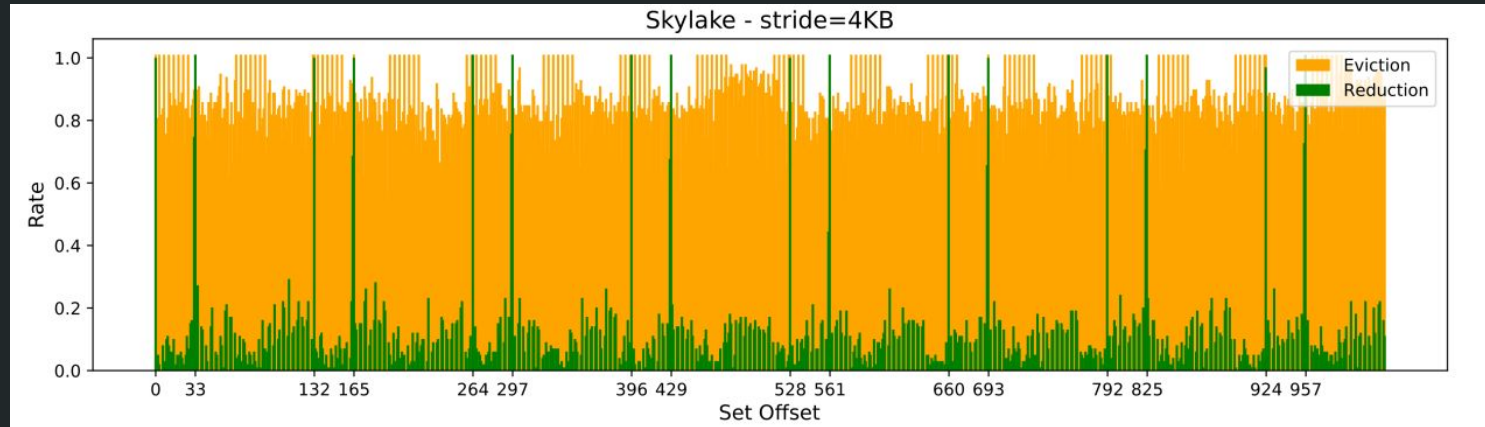
...but fell into the side channel's rabbit hole





# Motivation

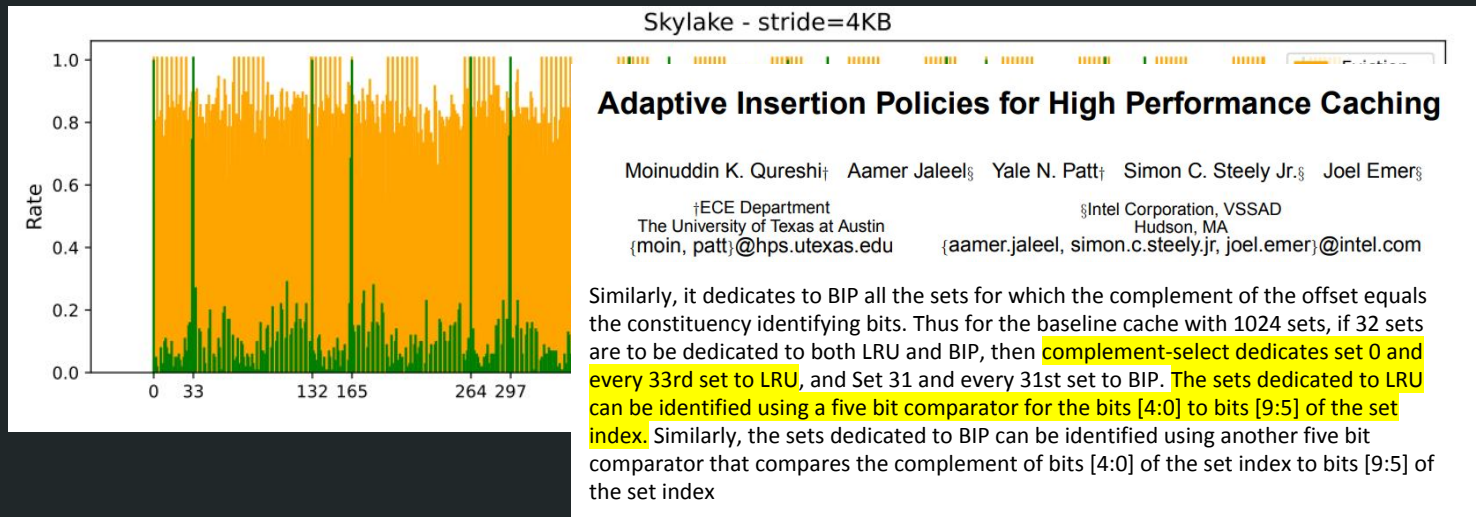
Remember last year's "Cache and syphilis"?



dafuq is this pattern :S

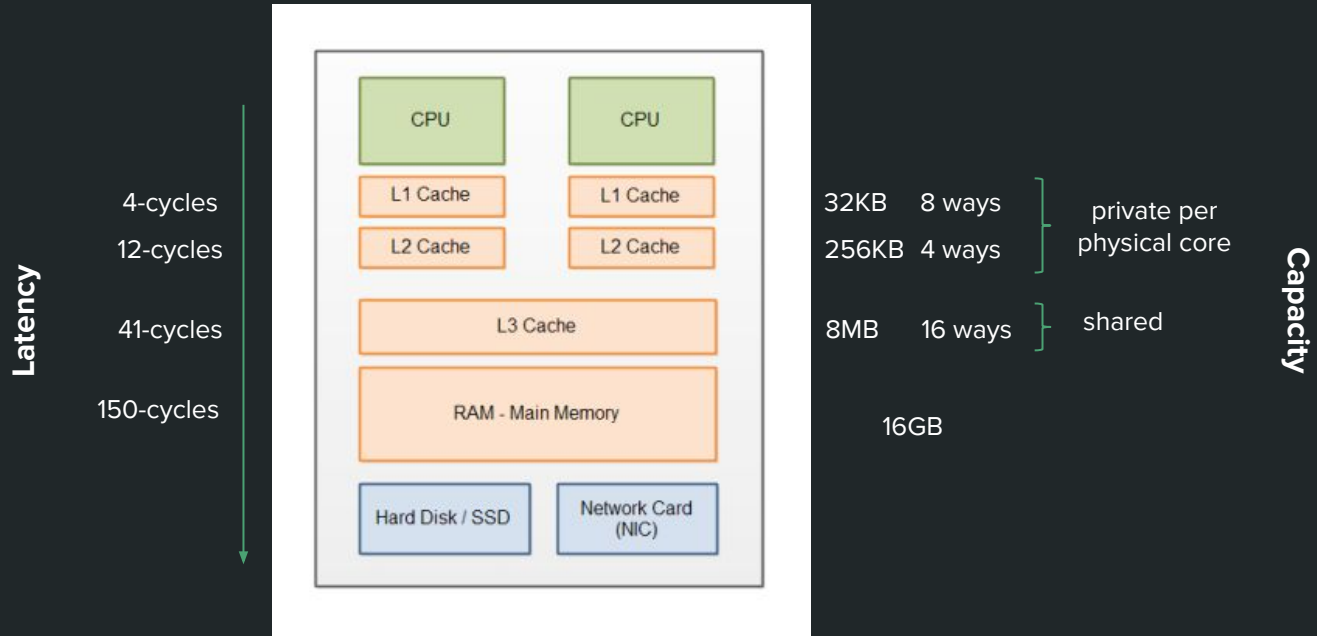
# Motivation

Knowing the cache replacement policy useful for finding eviction sets,



but also for optimal eviction strategies in rowhammer,  
or high bandwidth covert channels

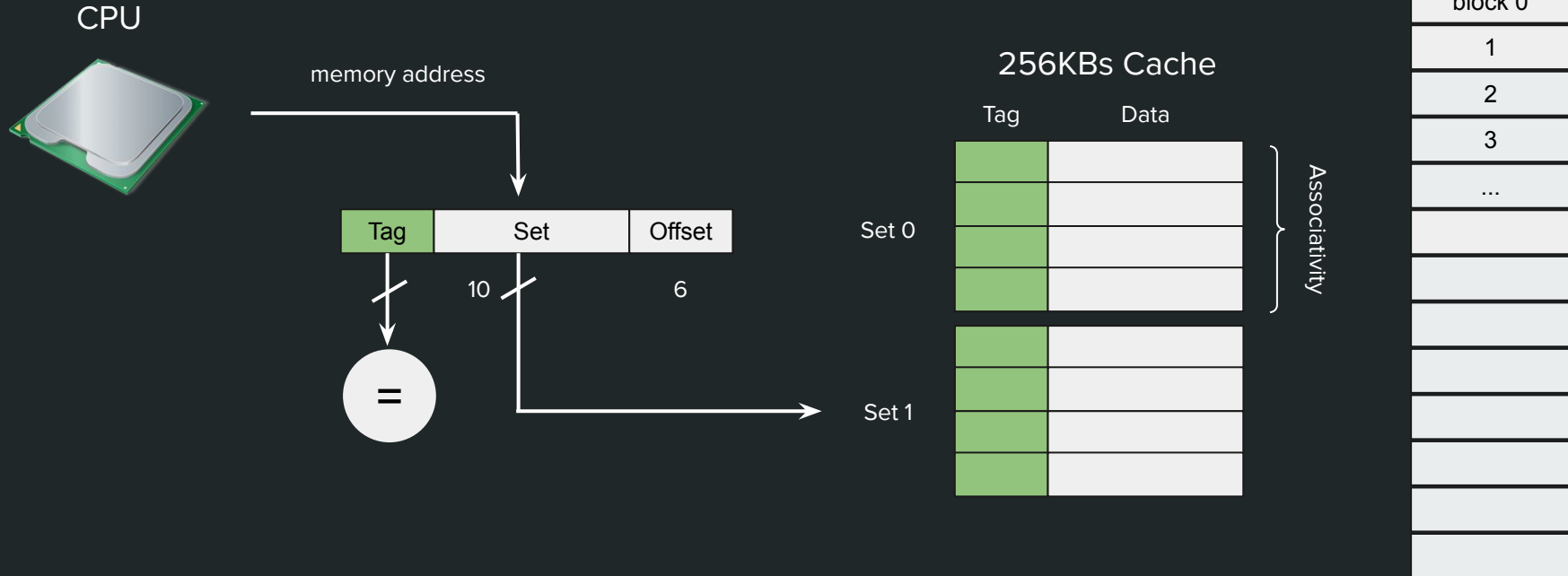
# A primer on Hardware Caches



(data from Kaby Lake i7-8550U CPU)

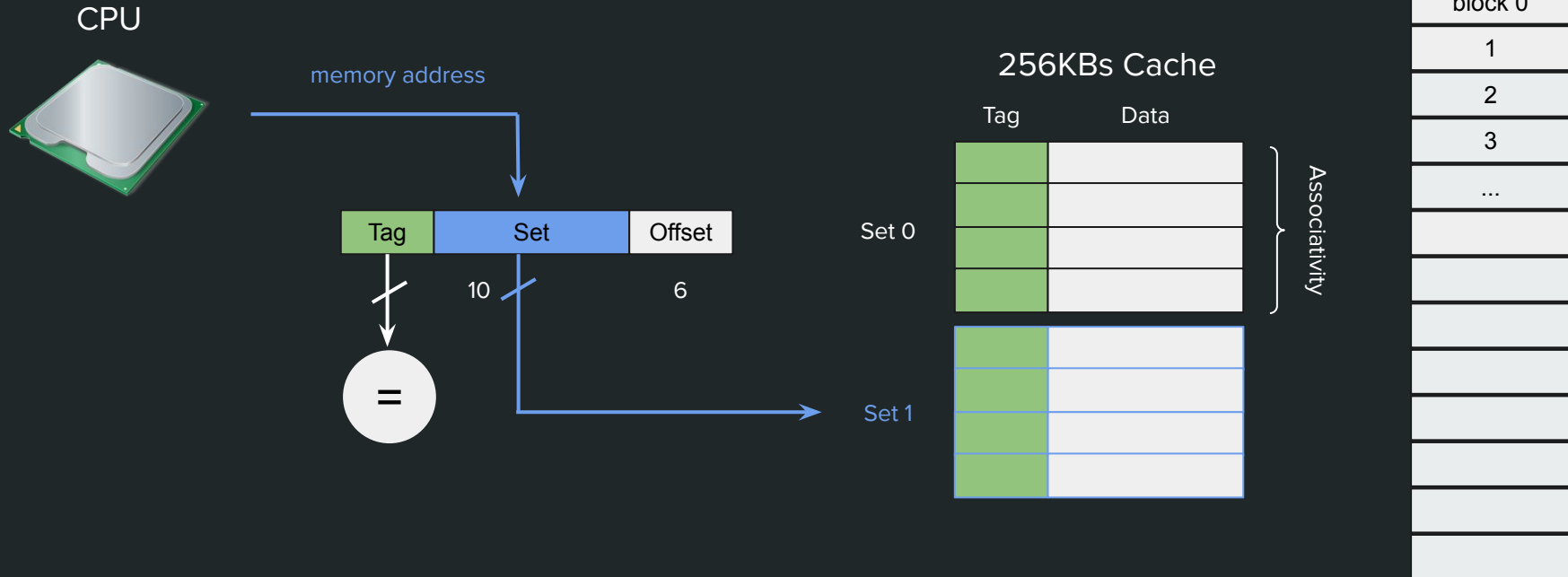


# A primer on Hardware Caches



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

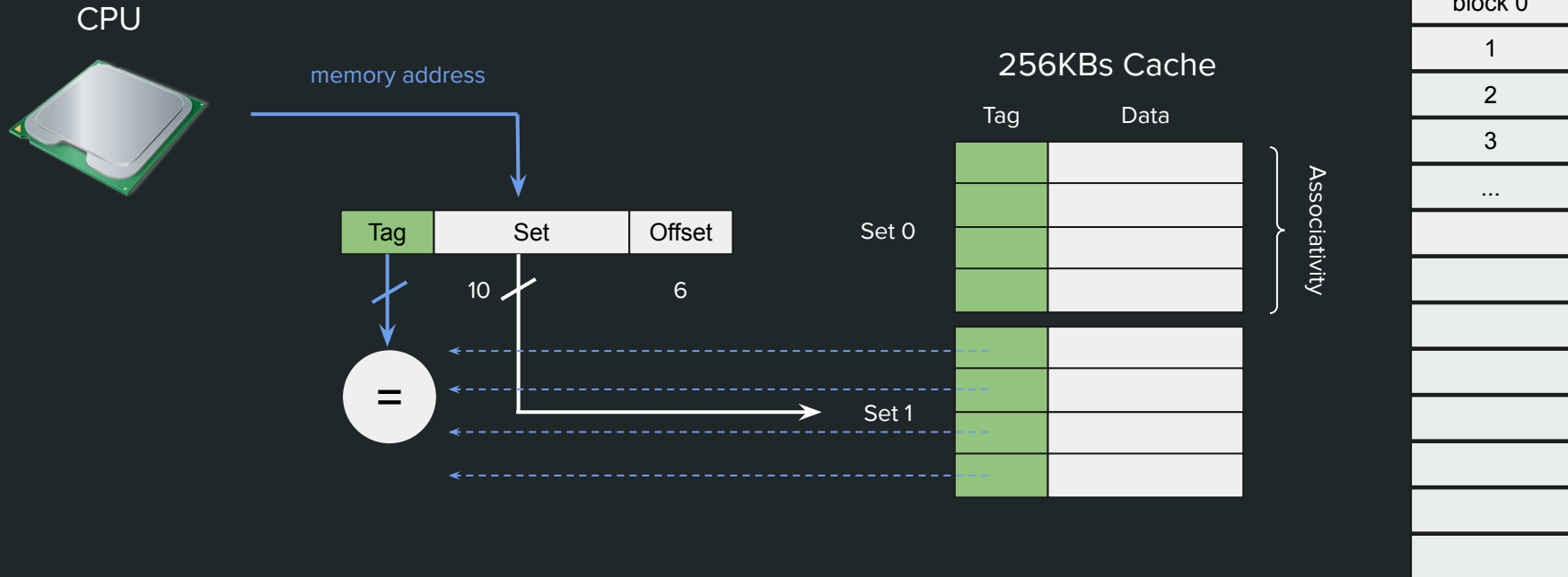
# A primer on Hardware Caches



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

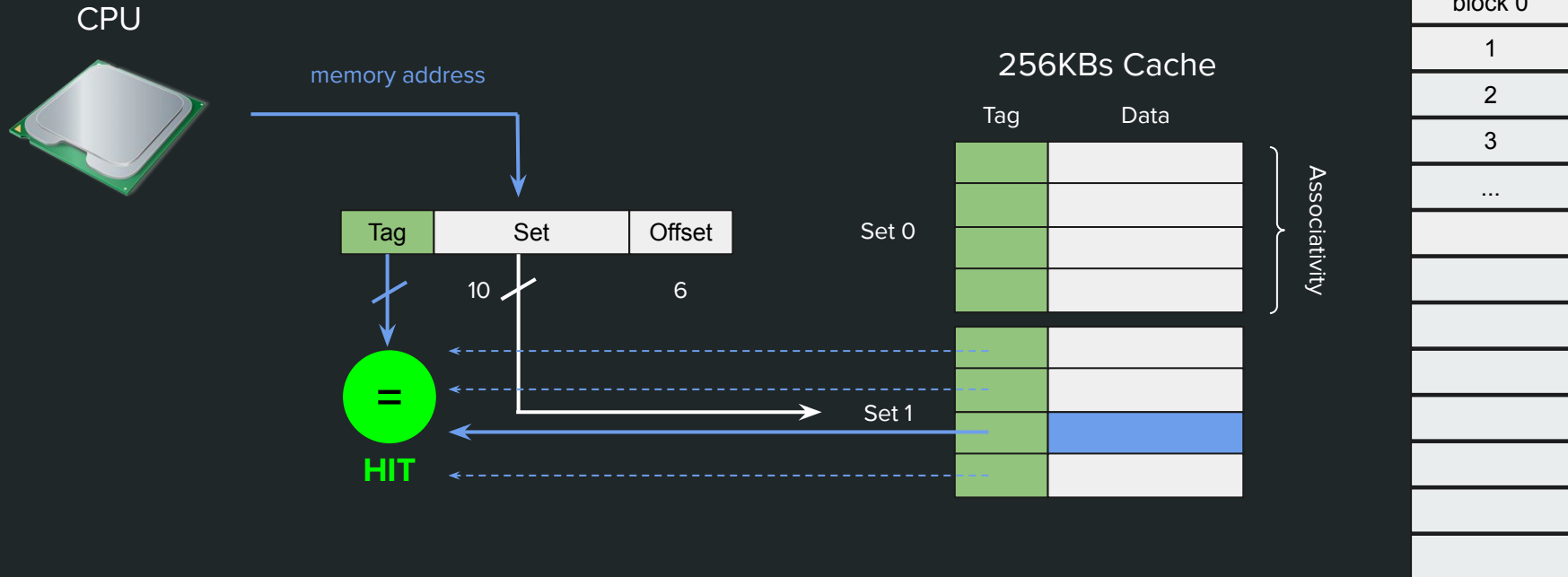


# A primer on Hardware Caches



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

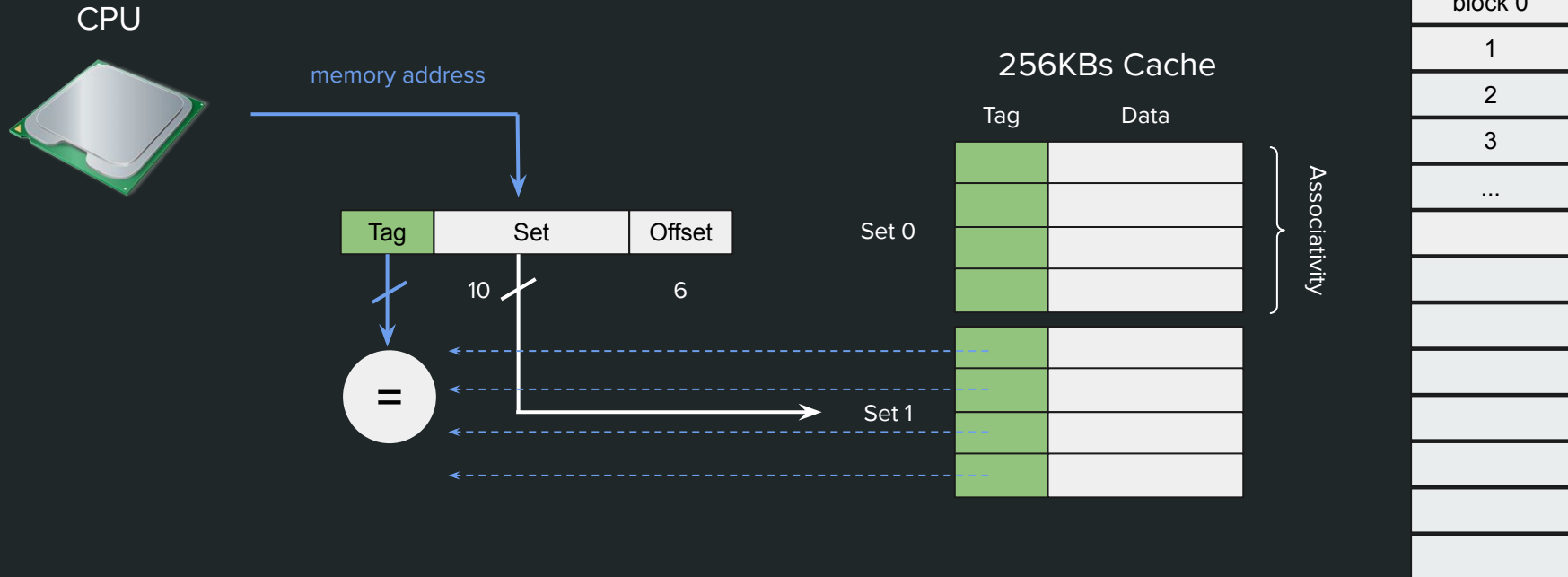
# A primer on Hardware Caches



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

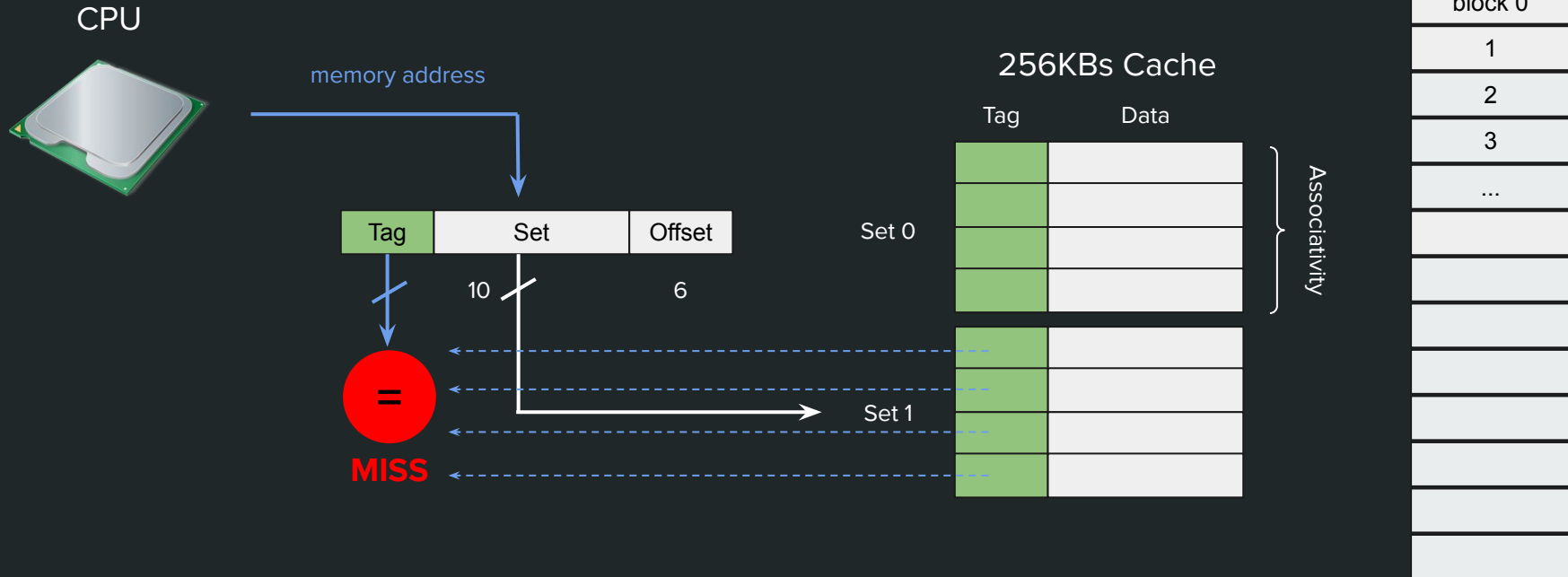


# A primer on Hardware Caches



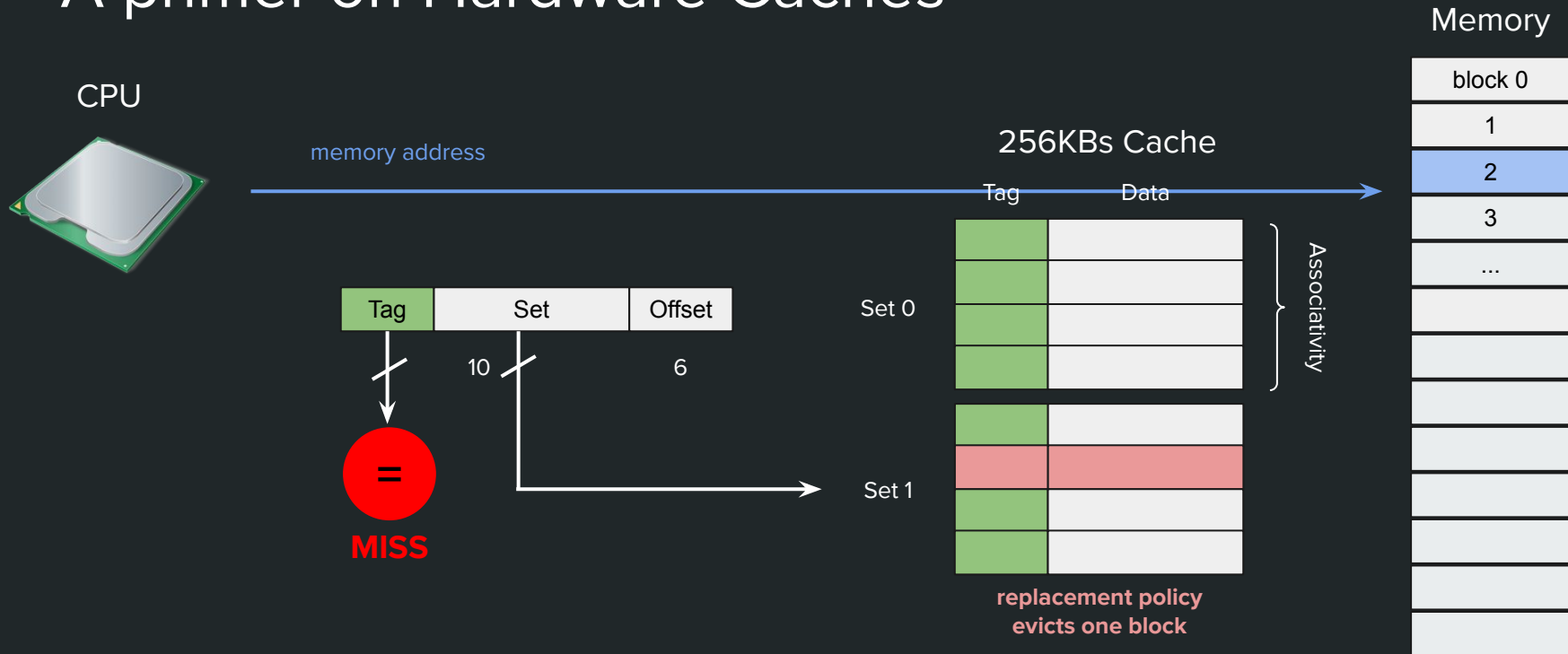
- Memory partitioned in **memory blocks** (64 bytes =  $2^6$ )
- Cache partitioned in equally sized **cache sets** ( $1024 = 2^{10} = 256\text{KB} / (64 * \text{associativity})$ )
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)

# A primer on Hardware Caches



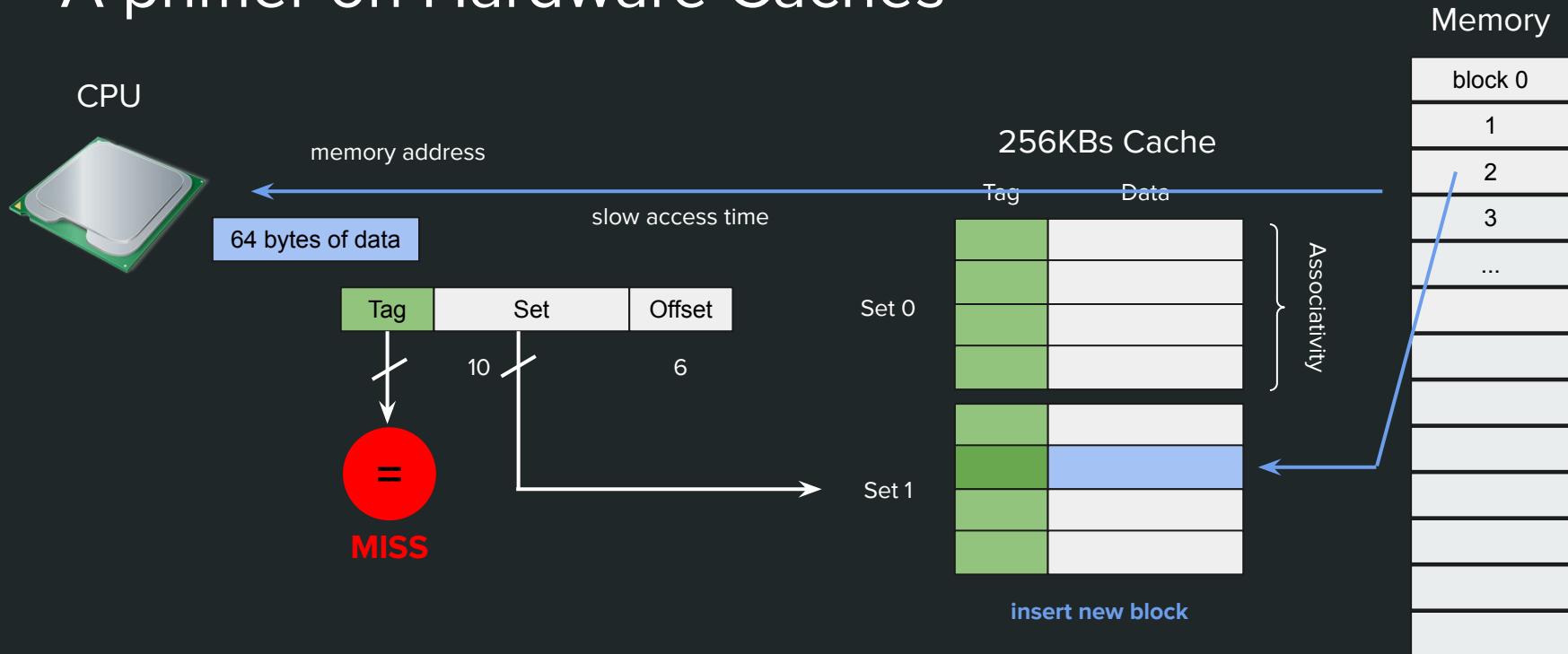
- Memory partitioned in **memory blocks** (64 bytes =  $2^6$ )
- Cache partitioned in equally sized **cache sets** ( $1024 = 2^{10} = 256\text{KB} / (64 * \text{associativity})$ )
- Cache sets have capacity for N **cache lines** (also known as ways or **associativity**)

# A primer on Hardware Caches



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

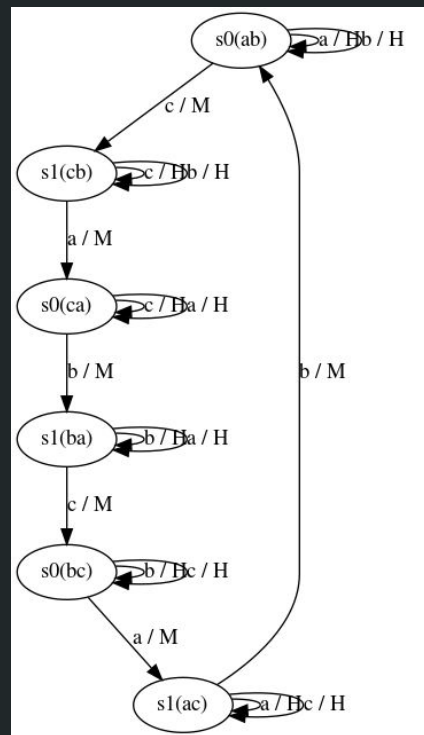


# A primer on Hardware Caches

- 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)



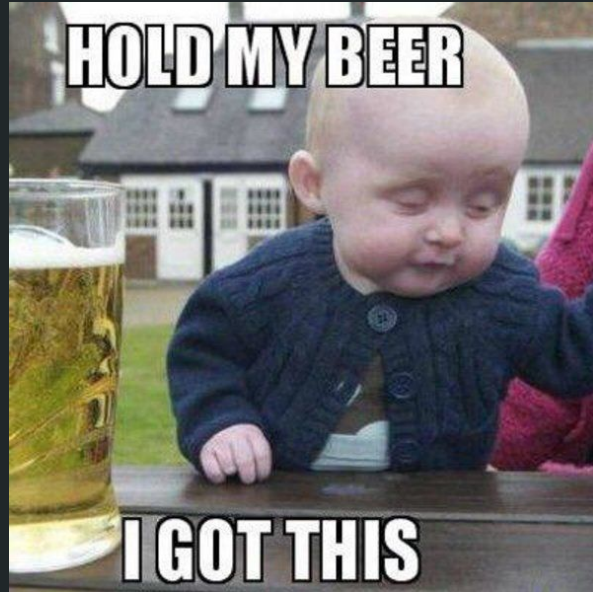
Example: 2-way FIFO cache with 3 blocks  $\{a, b, c\}$

# Previous work

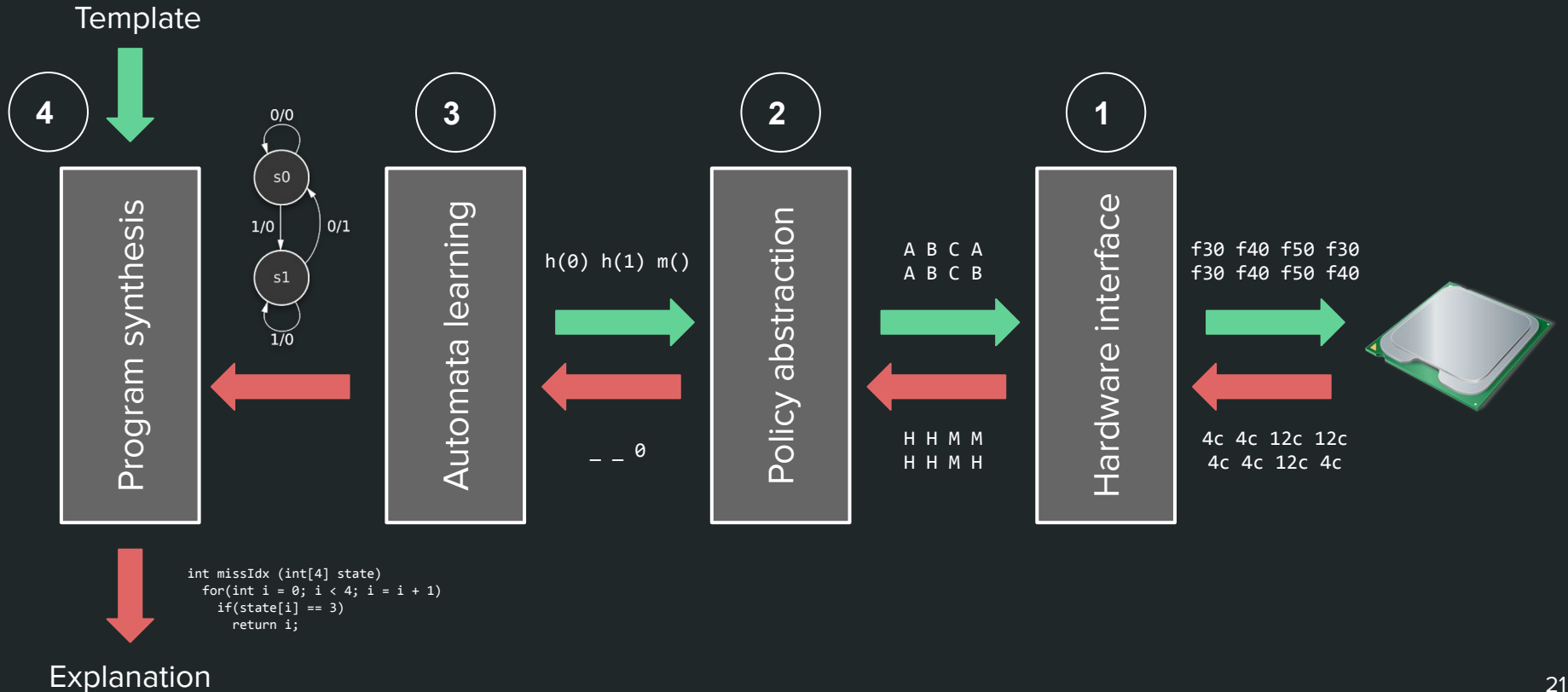


# Previous work

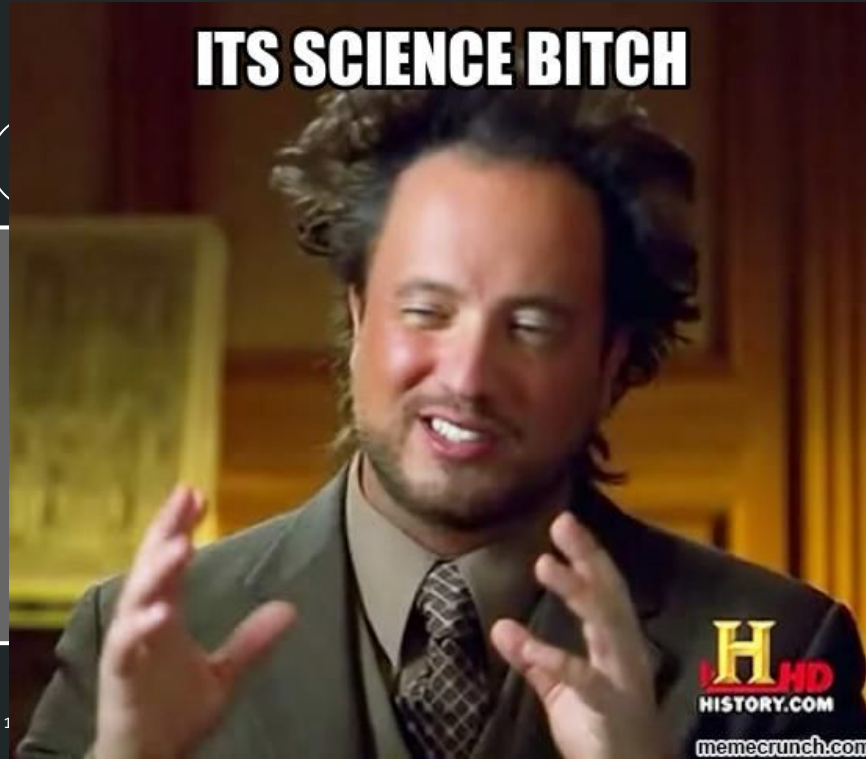
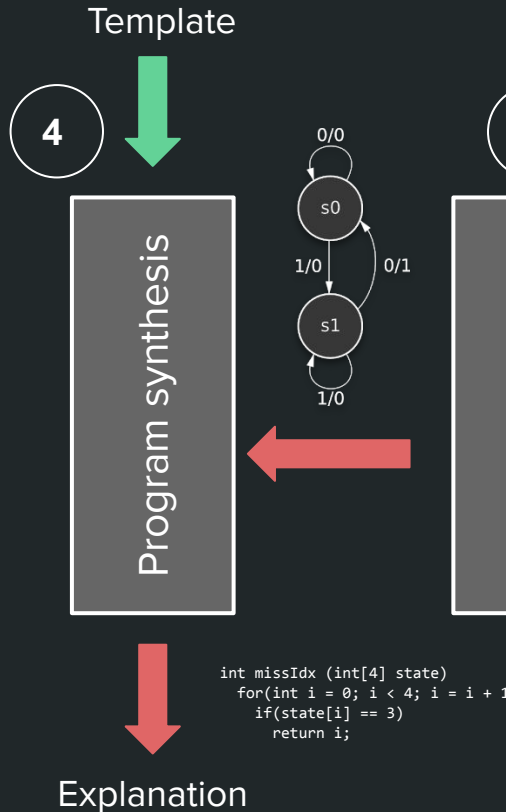
	Others	Abel & Reineke	Rueda's MS
Automatic	NO	YES	YES
Supported class of policies	Individual	Permutation-based	Deterministic
On real hardware	YES	YES	NO
Scalability	NO	YES	NO
Human readable	NO	NO	NO
Correctness	NO	YES	NO



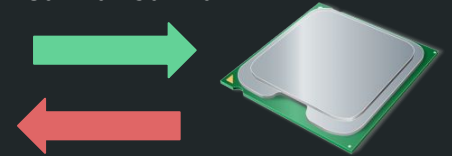
# Our approach



# Our approach



f30 f40 f50 f30  
f30 f40 f50 f40



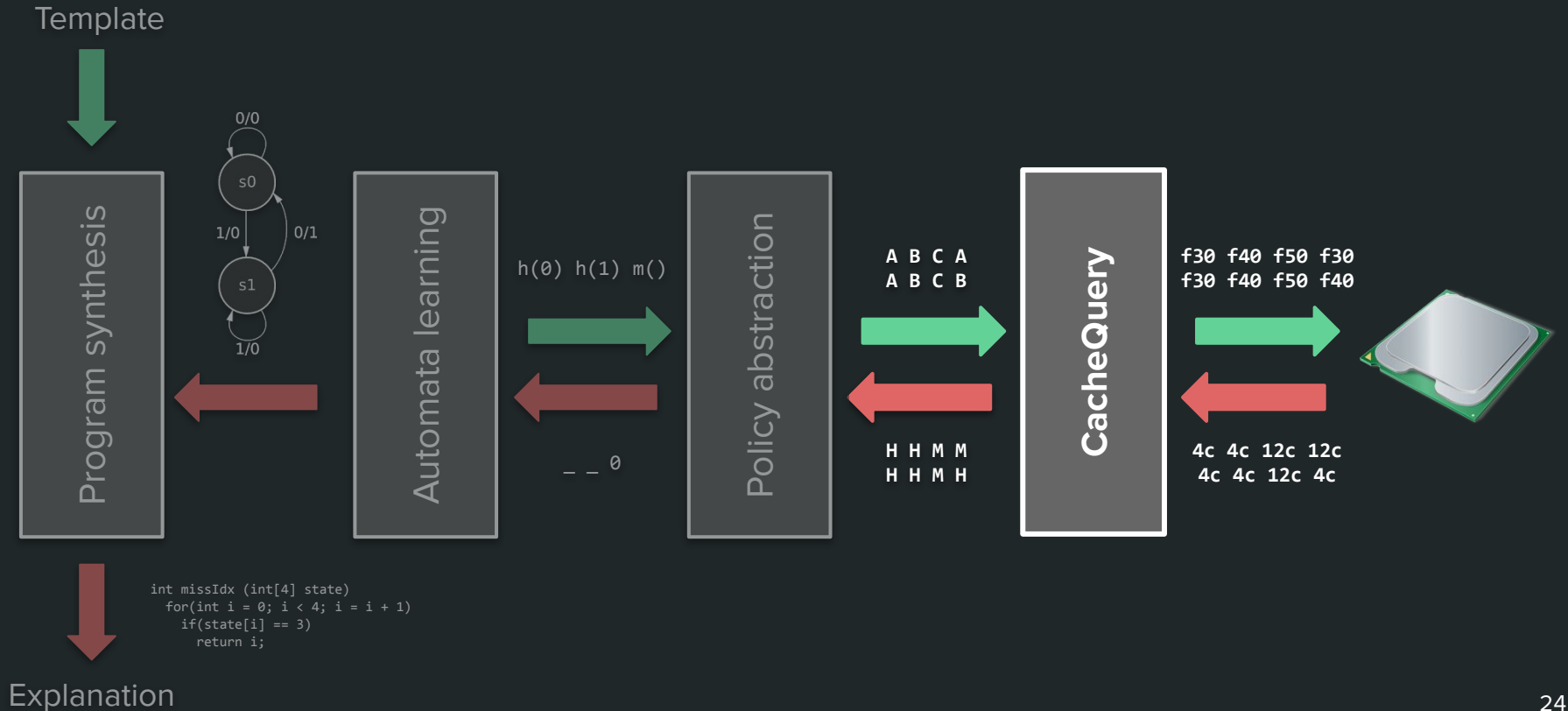
4c 4c 12c 12c  
4c 4c 12c 4c



# Previous work vs. our approach

	Others	Abel & Reineke	Rueda's MS	Our
Automatic	NO	YES	YES	YES
Supported class of policies	Individual	Permutation-based	Deterministic	Deterministic
On real hardware	YES	YES	NO	YES
Scalability	NO	YES	NO	YES
Human readable	NO	NO	NO	YES
Correctness	NO	YES	NO	YES

# CacheQuery: a hardware interface



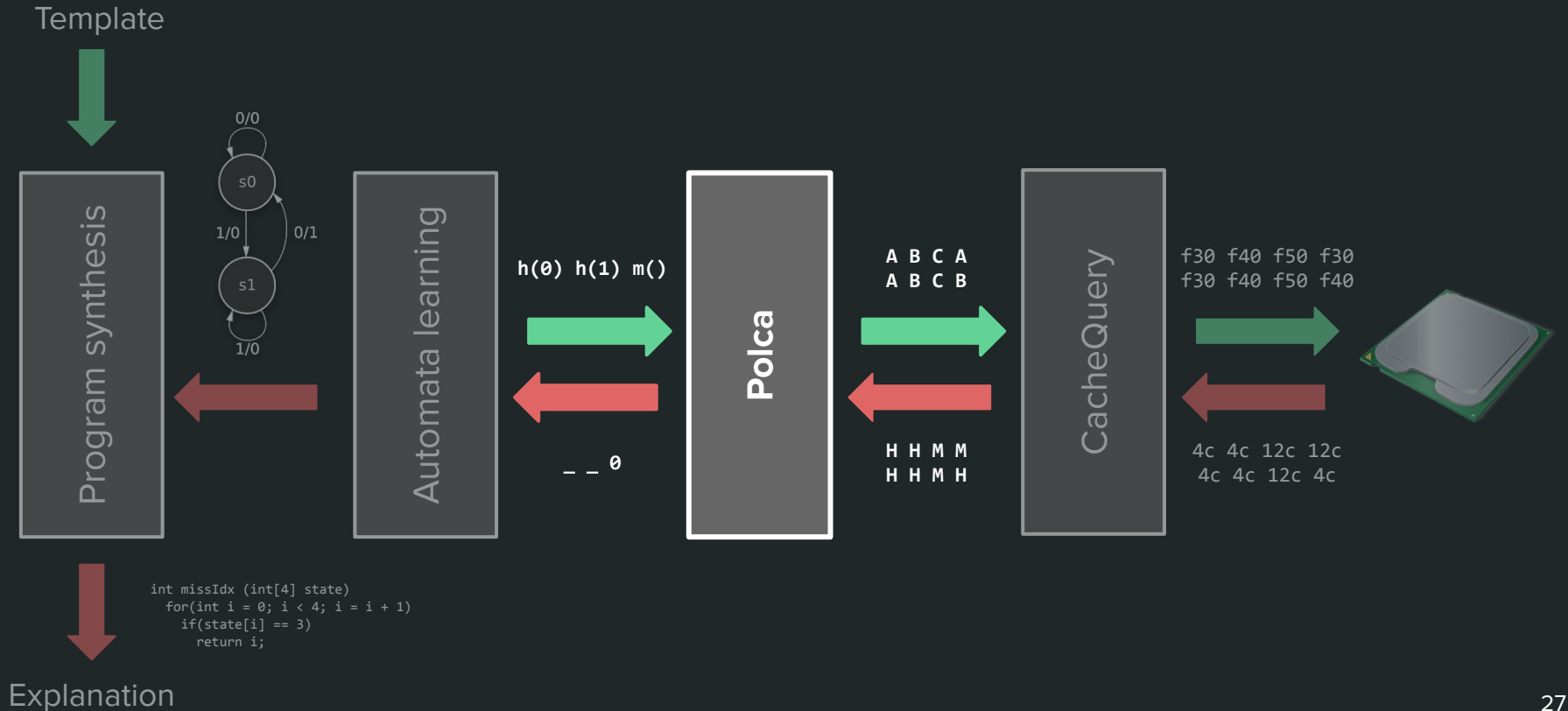
# CacheQuery: a hardware interface

- 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

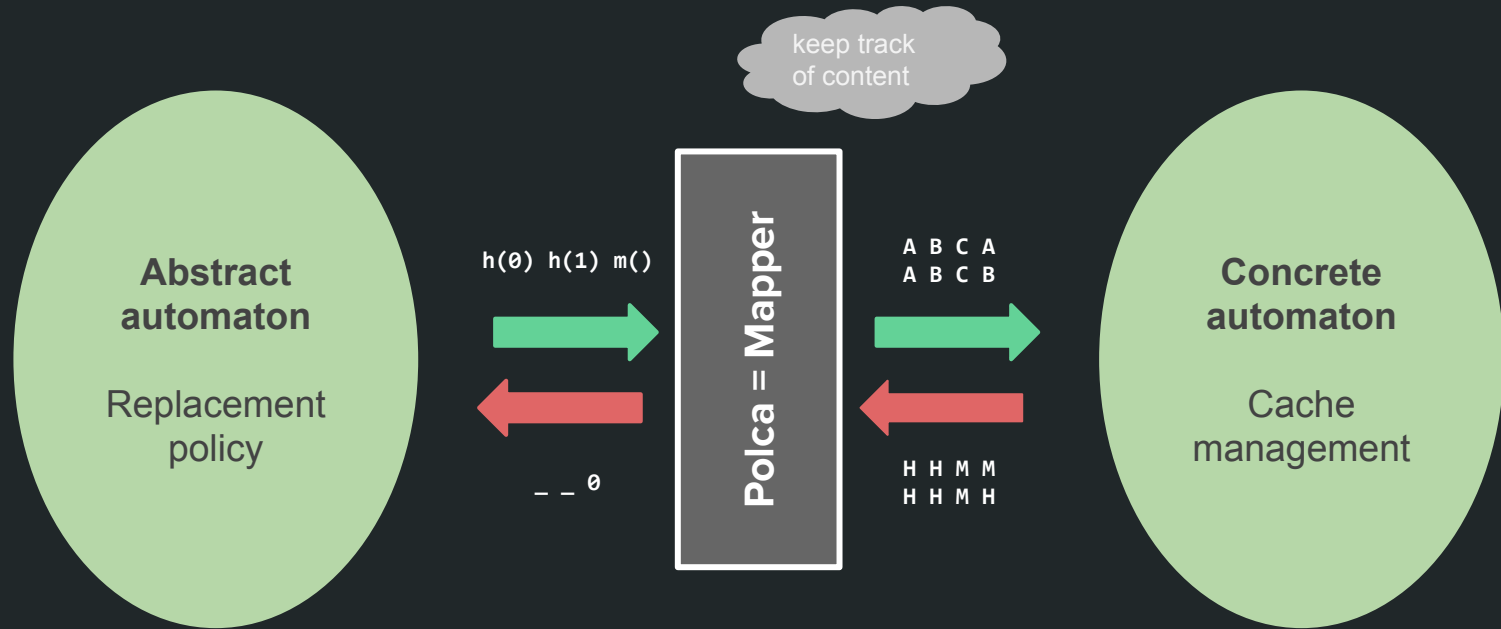
# Polca: a cache automaton abstraction



# 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



Input:  $\{h(0), h(1), \dots, h(n-1), m()\}$

$\{A, B, C, \dots\}$

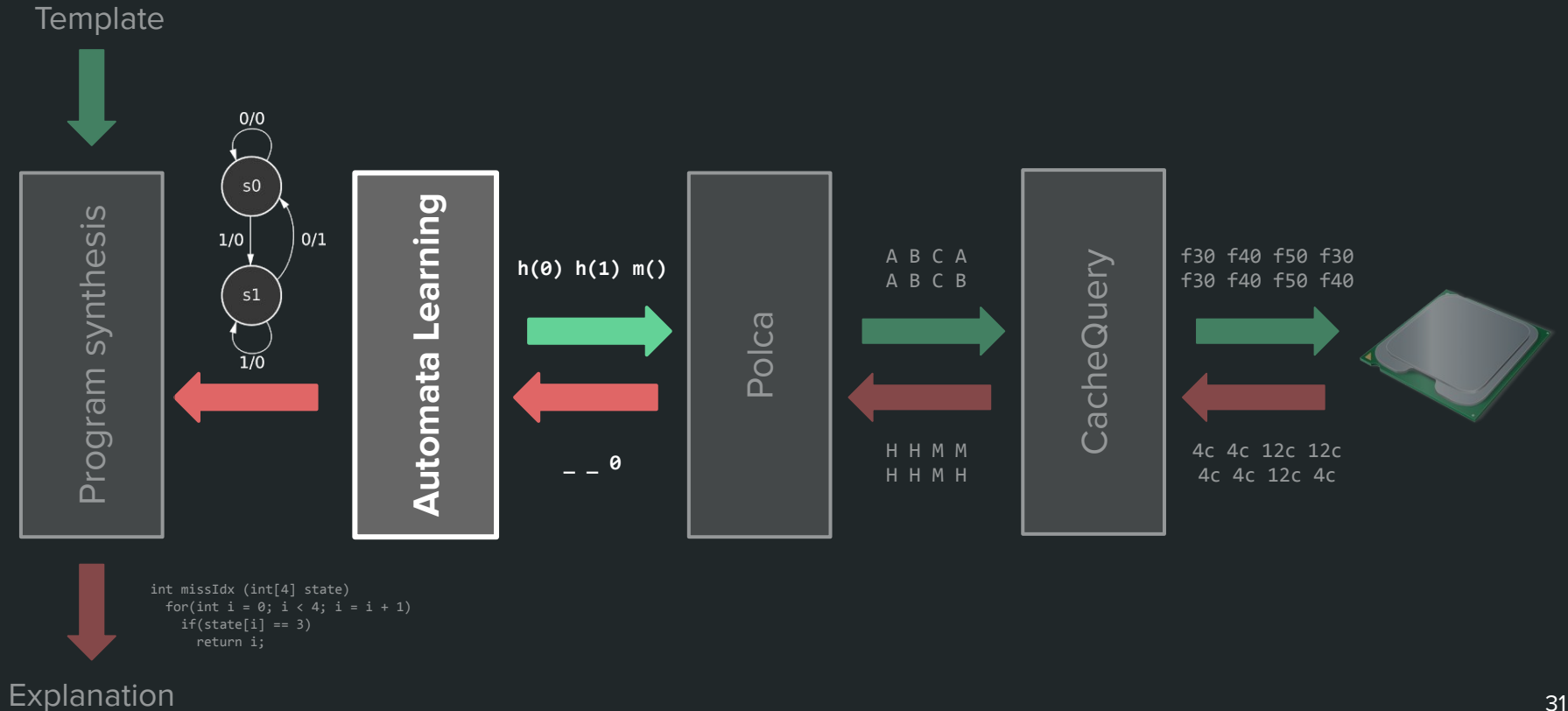
Output:  $\{\_, 0, 1, \dots, n-1\}$

$\{H, M\}$





# LearnLib: an automata learning framework



# 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  $\mathbf{U} = \{\mathbf{aa}, \mathbf{bb}\}$  (alphabet  $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ )
- Student asks if 'ε', 'a', and 'b' are in U and obtains the following **Observation Table**:

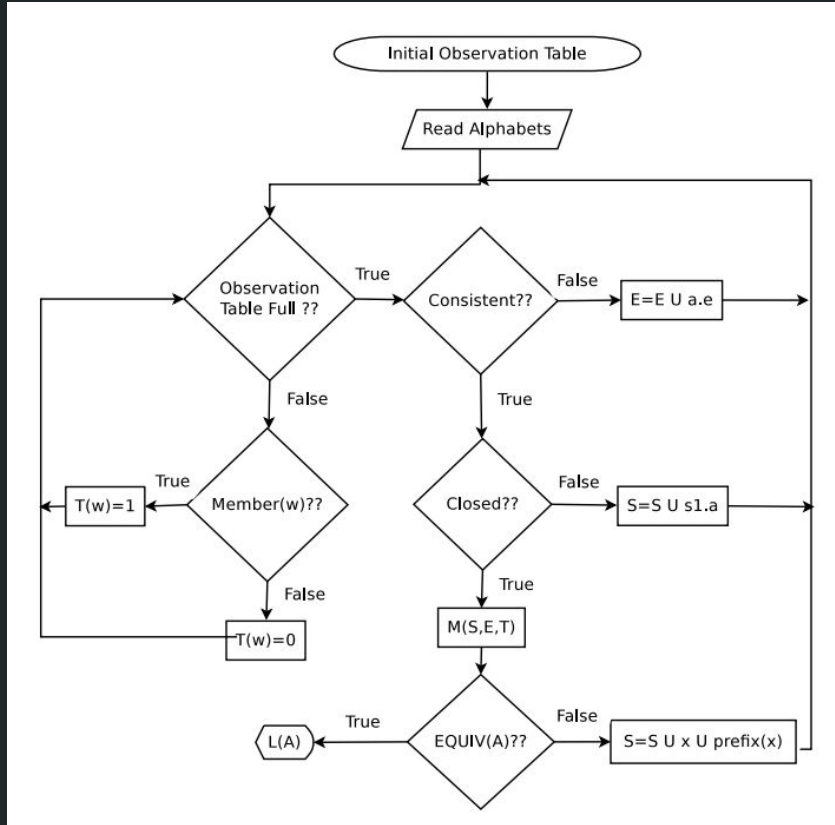
	ε
ε	0
a	0
b	0

Set of strings  $\mathbf{S}$ , represents the states

$\mathbf{S} \cdot \Sigma$

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

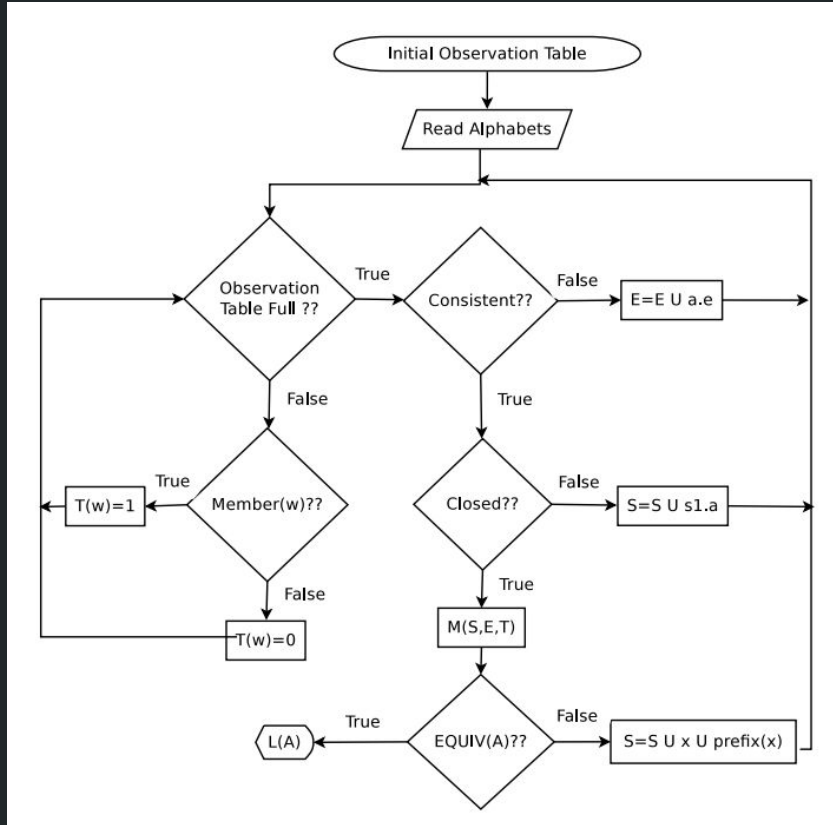
# L\* by example



Observation Table:

	$\epsilon$
$\epsilon$	0
a	0
b	0

# L\* by example



Observation Table:

	$\epsilon$
$\epsilon$	0
a	0
b	0

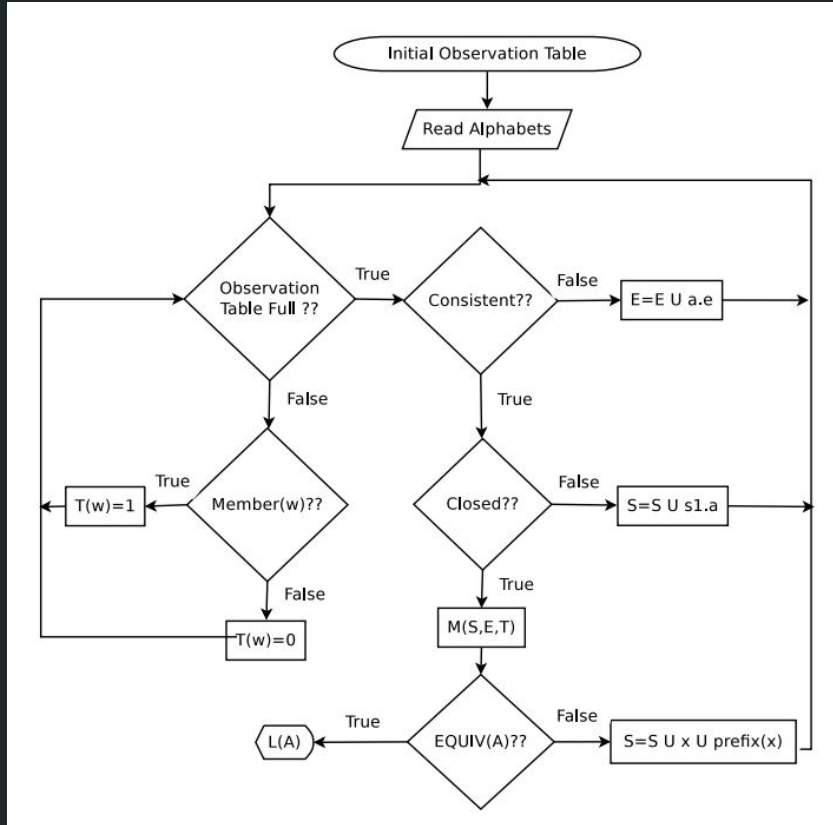


It is closed and consistent  
Hypothesis: empty language!

Teacher says NO and returns: **ce = aa**

We need to **extend S with 'ce' and all its prefixes**

# L\* by example

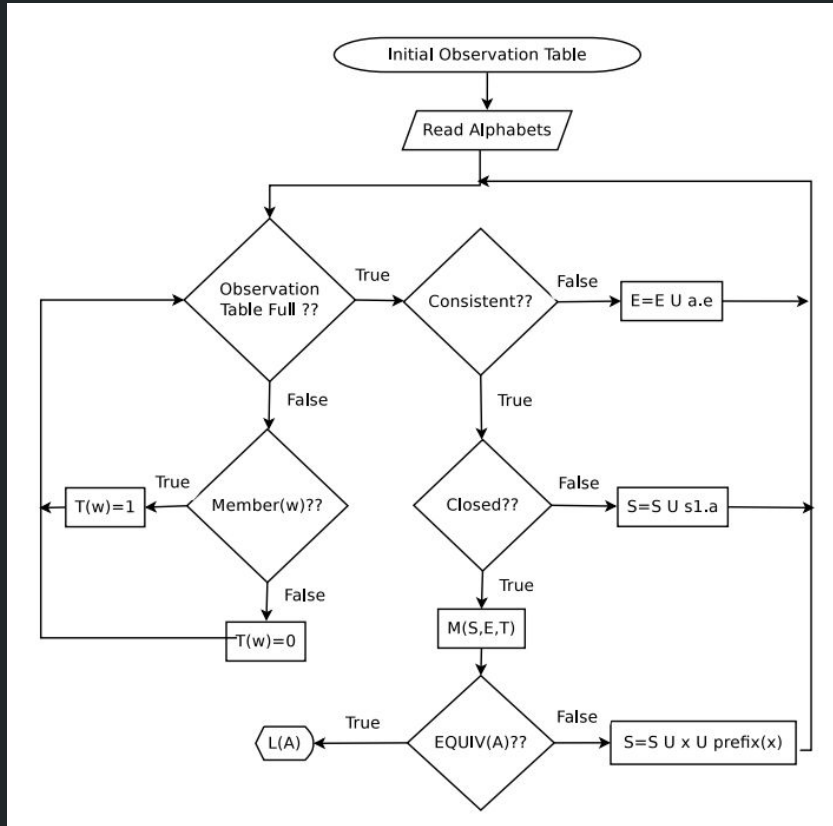


Observation Table:

	$\epsilon$
$\epsilon$	0
a	0
aa	?
b	0
ab	?
aaa	?
aab	?

perform a more membership queries

# L\* by example



Observation Table:

	$\epsilon$
$\epsilon$	0
a	0
aa	1
b	0
ab	0
aaa	0
aab	0

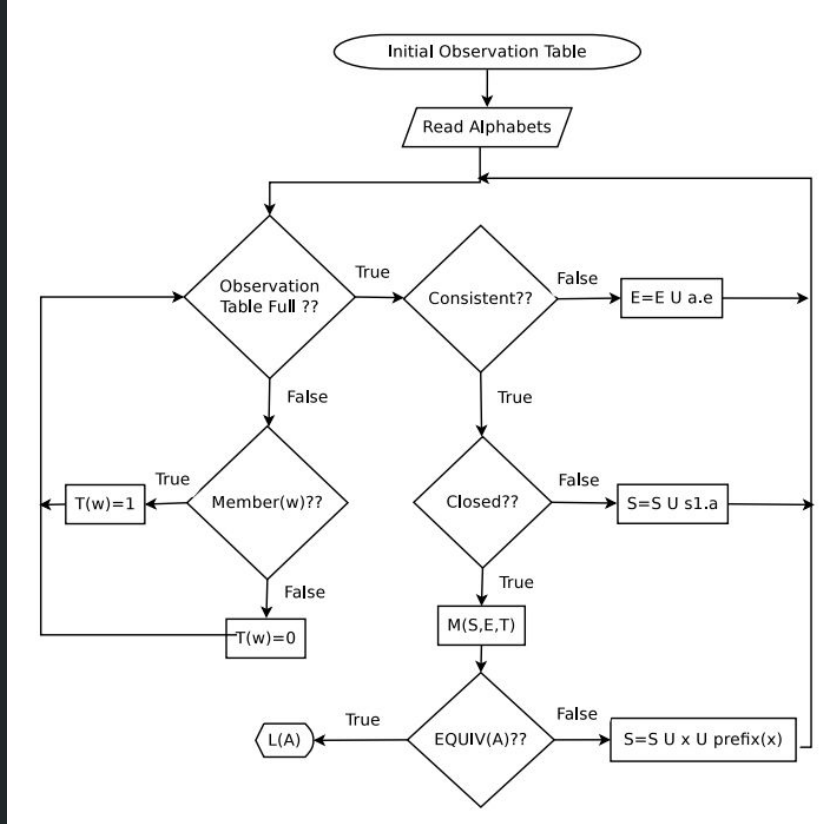
new table is closed, but **not consistent**

row( $\epsilon$ ) = row(a), but row( $\epsilon.a$ )  $\neq$  row(a.a)

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



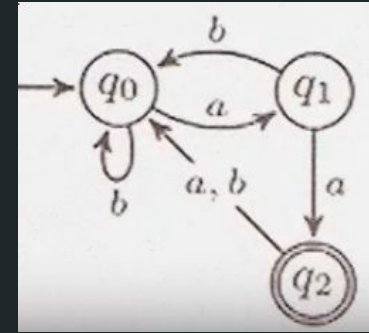
# L\* by example



Observation Table:

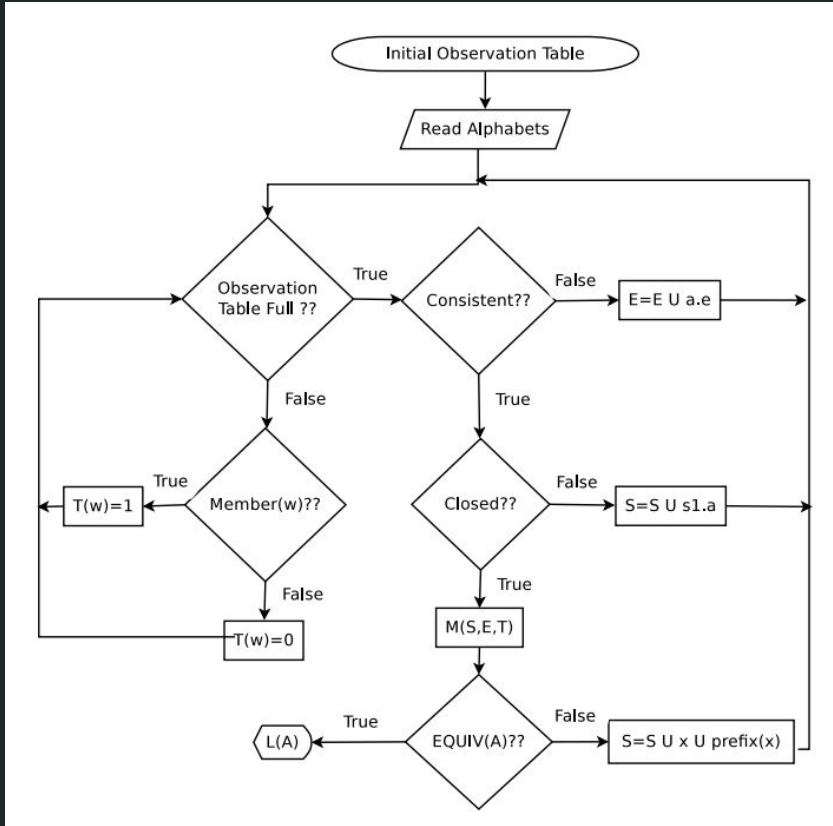
	$\epsilon$	a
$\epsilon$	0	0
a	0	1
aa	1	0
b	0	0
ab	0	0
aaa	0	0
aab	0	0

now it is closed and consistent



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

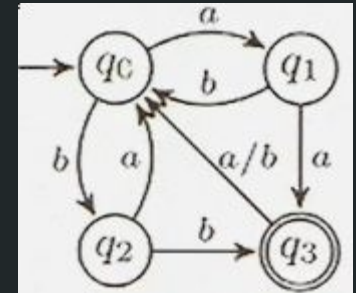
# L\* by example



Observation Table:

	$\epsilon$	a	b
$\epsilon$	0	0	0
a	0	1	0
aa	1	0	0
b	0	0	1
bb	1	0	0
ab	0	0	0
aaa	0	0	0
aab	0	0	0
ba	0	0	0
bba	0	0	0
bbb	0	0	0

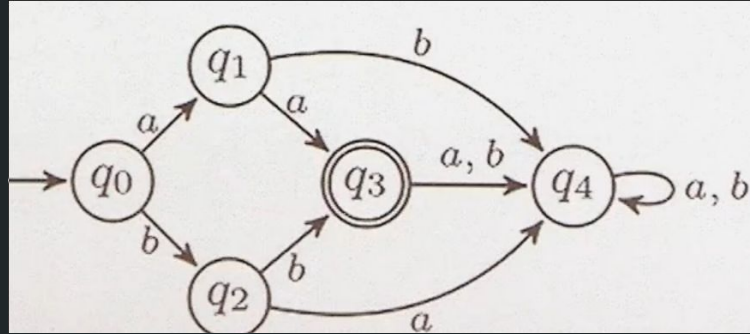
table is closed and consistent, let's see if hypothesis is correct not?



nope **ce = babb**

# $L^*$ by example


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



- The algorithm ensures that on every hypothesis the **automaton is minimal**.
- Teacher can give arbitrarily long counterexamples.

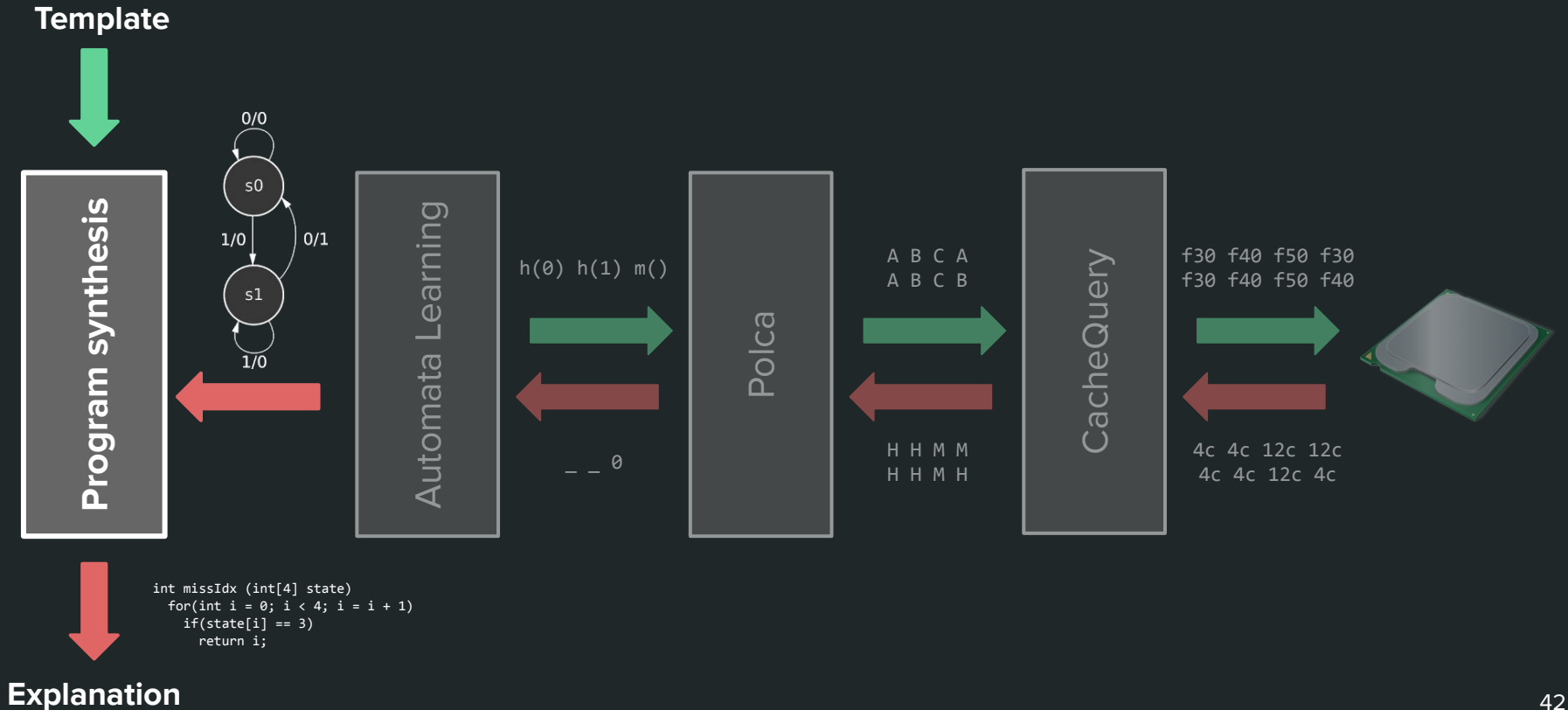
# LearnLib handles all the learning

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



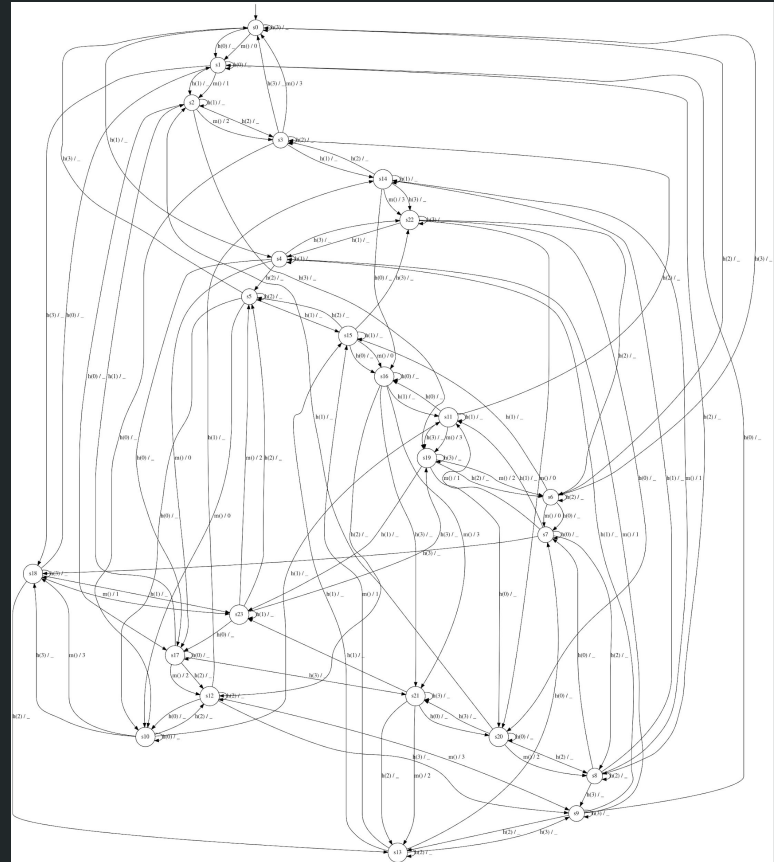
**WP-method:** test sequence selection - given an upper bound on the number of states of the System Under Learning (SUL), guarantees equivalence

# Sketch: synthesizing programs as explanations



# Sketch: synthesizing programs as explanations

- 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.



# Sketch: synthesizing programs as explanations

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:

```
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:

```
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
```

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 :(

Policy	Assoc.	# States	Time
<i>FIFO</i>	2	2	0 h 0 m 0.14 s
	...	...	...
	16	16	0 h 0 m 0.38 s
<i>LRU</i>	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
<i>PLRU</i>	2	2	0.10 s
	4	8	0.22 s
	8	128	1.46 s
	16	32768	34 h 18 m 25 s
<i>MRU</i>	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	
<i>LIP</i>	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
<i>SRRIP-HP</i>	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
<i>SRRIP-FP</i>	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.

# Case Study: Learning 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
i7-8850U (Kaby Lake)	L1	8	1	64
	L2	4	1	1024
	L3	16	8	1024

# 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

CPU	Level	Assoc.	Sets	States	Policy	Reset Seq.
<i>i7-4790</i> (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)	– –	– –	– –
<i>i5-6500</i> (Skylake)	L1	8	0 – 63	128	PLRU	@
	L2	4	0 – 1023	160	<i>New1</i>	D C B A @
	L3	4 <sup>†</sup>	0 33 132 165 264 297 396 429 528 561 660 693 792 825 924 957	175	<i>New2</i>	@
<i>i7-8550U</i> (Kaby Lake)	L1	8	0 – 63	128	PLRU	@
	L2	4	0 – 1023	160	<i>New1</i>	D C B A @
	L3	4 <sup>†</sup>	0 33 132 165 264 297 396 429 528 561 660 693 792 825 924 957	175	<i>New2</i>	@

**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

Policy	States	Time
FIFO	4	18ms
LRU	24	81ms
PLRU	8	-
LIP	24	4s
MRU	14	40s
SRRIP-HP	178	105h
SRRIP-FP	256	48h
<i>New1</i>	160	9h
<i>New2</i>	175	26h



# Case Study: Synthesizing Explanations



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

Initial insertion on a flushed cache set:

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

```
int[4] hitState (int[4] state, int pos)
  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;
```

```
int[4] missState (int[4] state)
  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;

// 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;
```

# Case Study: Synthesizing Explanations



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

Initial insertion on a flushed cache set:

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

```
int[4] hitState (int[4] state, int pos)
  int[4] final = state;
  // Promotion
  if (final[pos] > 1)
    final[pos] = 1;
  else
    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
  if(!found)
    for(int i = 0; i < 4; i = i + 1)
      final[i] = final[i] + 1;
  return final;
```

```
int[4] missState (int[4] state)
  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
  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;
```



# 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>



<https://arxiv.org/pdf/1912.09770.pdf>

# Thank you for listening! Questions?



<https://github.com/cgwwzq/cachequery>



<https://github.com/cgwwzq/polca>



<https://arxiv.org/pdf/1912.09770.pdf>

# References

Adaptive Insertion Policies for High Performance Caching

<https://researcher.watson.ibm.com/researcher/files/us-moinqureshi/papers-dip.pdf>

Intel Ivy Bridge Cache Replacement Policy

<http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>

Measurement-based Modeling of the Cache Replacement Policy

<http://embedded.cs.uni-saarland.de/publications/CacheModelingRTAS2013.pdf>

Learning Cache Replacement Policies using Register Automata

<https://uu.diva-portal.org/smash/get/diva2:678847/FULLTEXT01.pdf>

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  $((((\text{set} \ \& \ 0x3e0) \gg 5) \oplus (\text{set} \ \& \ 0x1f)) = 0x0) \wedge ((\text{set} \ \& \ 0x2) = 0x0)$  fixed policy susceptible to thrashing (group 1)
    - rest of sets seem to use an adaptive policy
    - but sets whose indexes satisfy  $((((\text{set} \ \& \ 0x3e0) \gg 5) \oplus (\text{set} \ \& \ 0x1f)) = 0x1f) \wedge ((\text{set} \ \& \ 0x2) = 0x2)$  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