MAXIMIZING CACHE PERFORMANCE UNDER UNCERTAINTY

Nathan Beckmann

CMU

Daniel Sanchez

MIT

HPCA-23 in Austin TX, February 2017

The problem

- Caches are a critical for overall system performance
  - DRAM access = ~1000x instruction time & energy
- Cache space is scarce
- With perfect information (ie, of future accesses), a simple **metric** is **optimal**
  - Belady’s MIN: Evict candidate with largest time until next reference
- In practice, policies must cope with **uncertainty**, never knowing when candidates will next be referenced

WHAT'S THE RIGHT REPLACEMENT METRIC UNDER UNCERTAINTY?

PRIOR WORK HAS TRIED MANY APPROACHES

Practice

- Traditional: LRU, LFU, random
- Statistical cost functions [Takagi ICS'04]
- Bypassing [Qureshi ISCA'07]
- Likelihood of reuse [Khan MICRO'10]
- Reuse interval prediction [Jaleel ISCA'10][Wu MICRO'11]
- Protect lines from eviction [Duong MICRO'12]
- Data mining [Jimenez MICRO'13]
- Emulating MIN [Jain ISCA'16]

Theory

- MIN—optimal! [Belady, IBM'66][Mattson, IBM'70]
  - But needs perfect future information
- LFU—Independent reference model [Aho, J. ACM'71]
  - But assumes reference probabilities are static

Impractical—unrealizable assumptions

- Modeling many other reference patterns [Garetto'16, Beckmann HPCA'16, …]

Don't address optimality

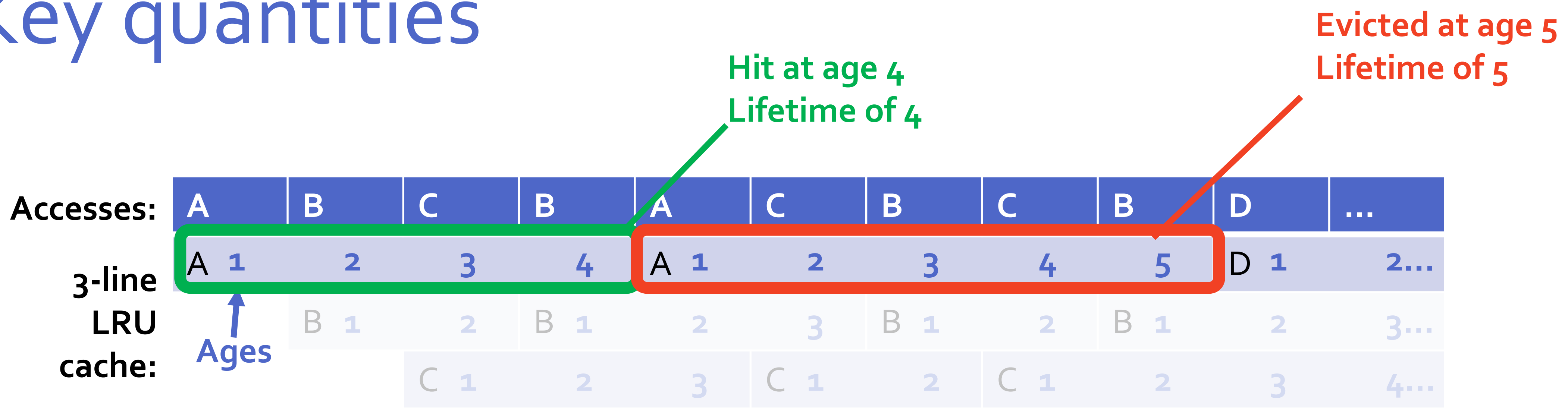
Without a foundation in theory, are any "doing the right thing"?

GOAL: A PRACTICAL REPLACEMENT METRIC WITH FOUNDATION IN THEORY

Fundamental challenges

- *Goal:* Maximize cache hit rate
- *Constraint:* Limited cache space
- *Uncertainty*: In practice, don’t know what is accessed when

Key quantities



- *Age* is how long since a line was referenced
- Divide cache space into *lifetimes* at hit/eviction boundaries
- Use *probability* to describe distribution of *lifetime* and *hit age*
  - $\mathrm{P}[L = a]$ ← probability a randomly chosen access lives a accesses in the cache
  - $\mathrm{P}[H = a]$ ← probability a randomly chosen access hits at age a

Fundamental challenges

- *Goal:* Maximize cache hit rate

$$\text{P}[\text{hit}] = \sum_{a=1}^{\infty} \text{P}[H = a]$$

Every hit occurs at some age $< \infty$

- *Constraint:* Limited cache space

$$S = \text{E}[L] = \sum_{a=1}^{\infty} a \times \text{P}[L = a]$$

Little's Law

<u>Observations:</u>
Hits beneficial irrespective of age
Cost (in space) increases in proportion to age

Insights & Intuition

- Replacement metric must balance *benefits* and *cost*

hits (benefits) **cache space** (cost)

<u>Observations:</u>

Hits beneficial irrespective of age

Cost (in space) increases in proportion to age

<u>Conclusion:</u>

$$\text{Replacement metric} \propto \text{hit probability}$$

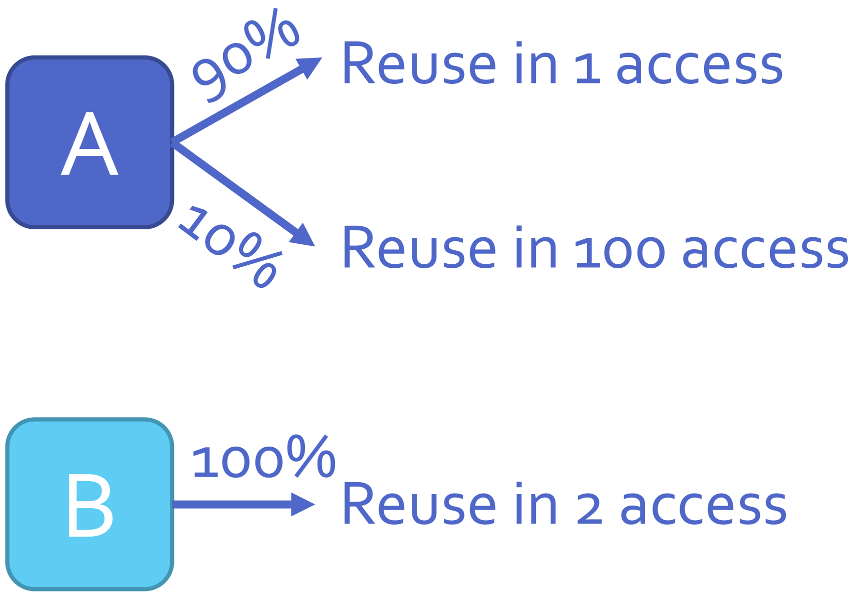
$$\text{Replacement metric} \propto -\text{expected lifetime}$$

Simpler ideas don't work

- MIN evicts the candidate with largest time until next reference
- Common generalization ➔ largest **predicted** time until next reference

Simpler ideas don’t work

- MIN evicts the candidate with largest time until next reference
- Common generalization ➔ largest **predicted** time until next reference



Q: Would you rather have A or B?

We would rather have **A**, because we can *gamble* that it will hit in 1 access and evict it otherwise

…But **A**’s expected time until next reference is larger than **B**’s.

THE KEY IDEA: REPLACEMENT BY ECONOMIC VALUE ADDED

Our metric: Economic value added (EVA)

- EVA reconciles **hit probability** and **expected lifetime** by measuring time in cache as **forgone hits**

- Thought experiment: how long does a hit need to take before it isn't worth it?
- *Answer: As long as it would take to net another hit from elsewhere.*
  - On average, each access yields hits = $\frac{\text{Hit rate}}{\text{Cache size}}$
  - ➔ Time spent in the cache costs this many **forgone hits**

$$\text{EVA} = \textit{Candidate's}\ \textbf{expected hits} - \frac{\text{Hit rate}}{\text{Cache size}} \times \textit{Candidate's}\ \textbf{expected time}$$

Our metric: Economic value added (EVA)

- EVA reconciles **hit probability** and **expected lifetime** by measuring time in cache as **forgone hits**

$$\text{EVA} = \textit{Candidate's expected hits} - \frac{\text{Hit rate}}{\text{Cache size}} \times \textit{Candidate's expected time}$$

- **EVA** measures how many hits a candidate nets vs. the average candidate
- **EVA** is essentially a cost-benefit analysis: is this candidate worth keeping around?
- Replacement policy evicts candidate with **lowest EVA**

Efficient implementation!

Estimate EVA using informative features

- EVA uses **conditional probability**
- Condition upon informative features, e.g.,
  - **Recency:** how long since this candidate was referenced? (candidate's age)
  - **Frequency:** how often is this candidate referenced?
  - Many other possibilities: requesting PC, thread id, …

This talk

The paper

Estimating EVA from recent accesses

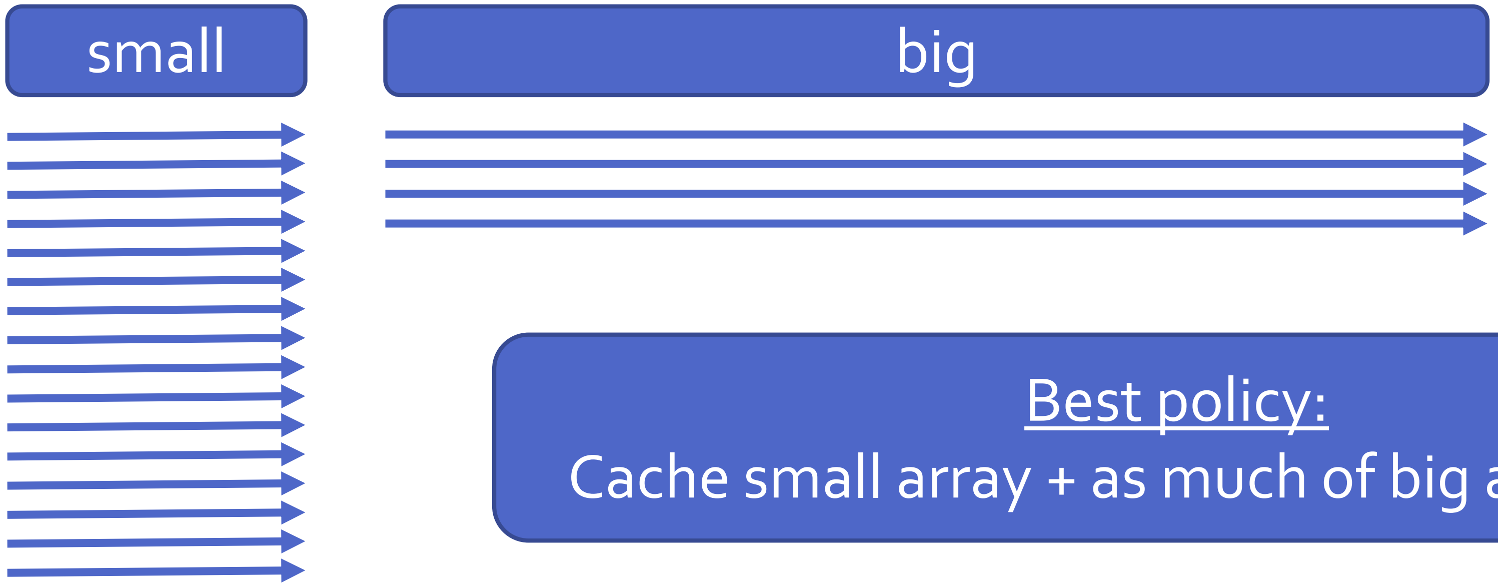
- Compute EVA using **conditional probability**

- A candidate of age a by definition hasn't hit or evicted at ages $\leq a$
- ➔ Can only hit at ages $> a$ and lifetime must be $> a$

- $\text{Hit probability} = \mathrm{P}[\text{hit} \mid \text{age } a] = \frac{\sum_{x=a}^{\infty} \mathrm{P}[H=a]}{\sum_{x=a}^{\infty} \mathrm{P}[L=x]}$
- $\text{Expected remaining lifetime} = \mathrm{E}[L-a \mid \text{age } a] = \frac{\sum_{x=a}^{\infty} (x-a)\, \mathrm{P}[L=a]}{\sum_{x=a}^{\infty} \mathrm{P}[L=x]}$

EVA by example

- Program scans alternating over two arrays: ‘**big**’ and ‘**small**’

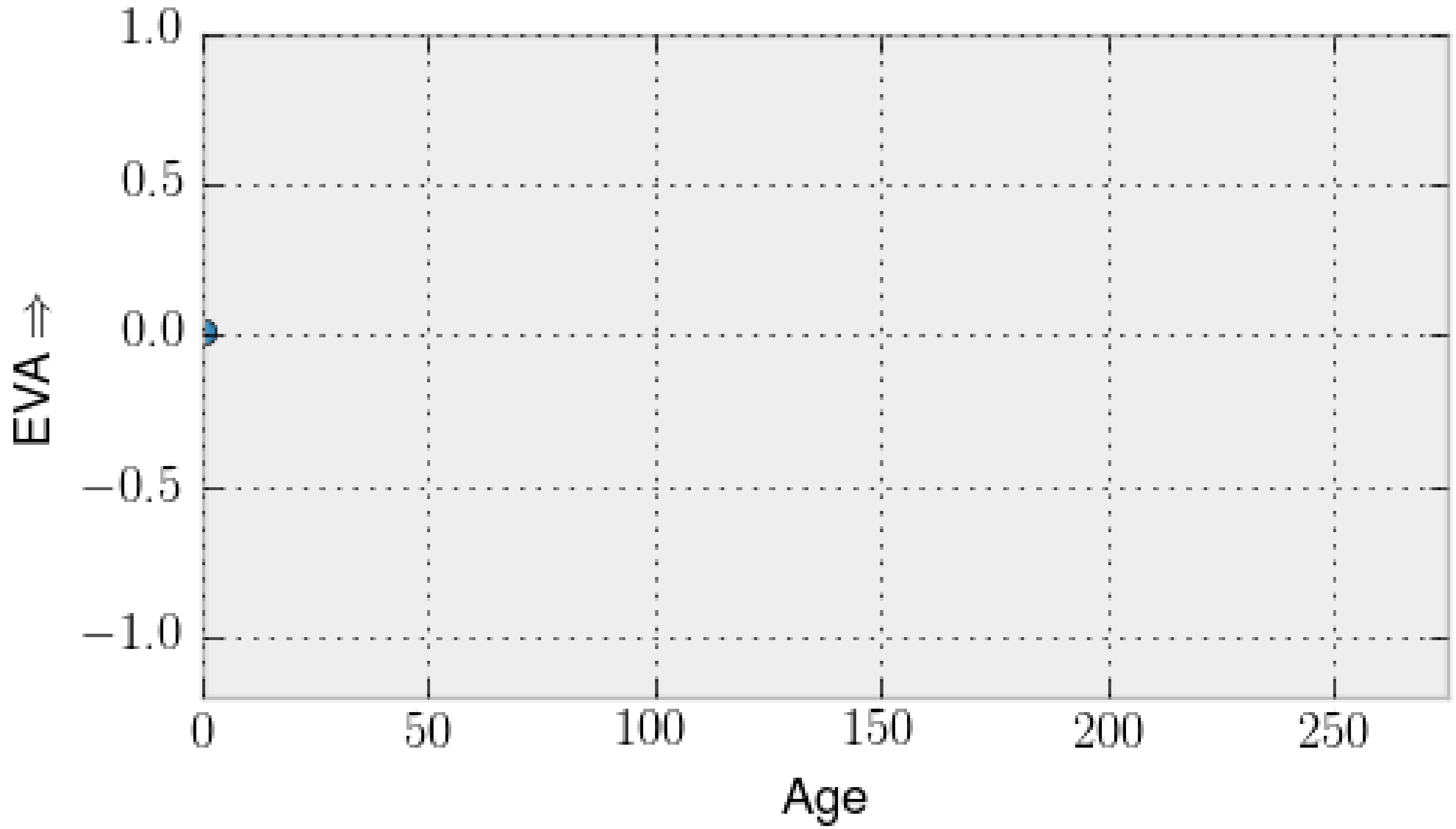


<u>Best policy:</u>
Cache small array + as much of big array as fits

EVA by example

- Program scans alternating over two arrays: '**big**' and '**small**'

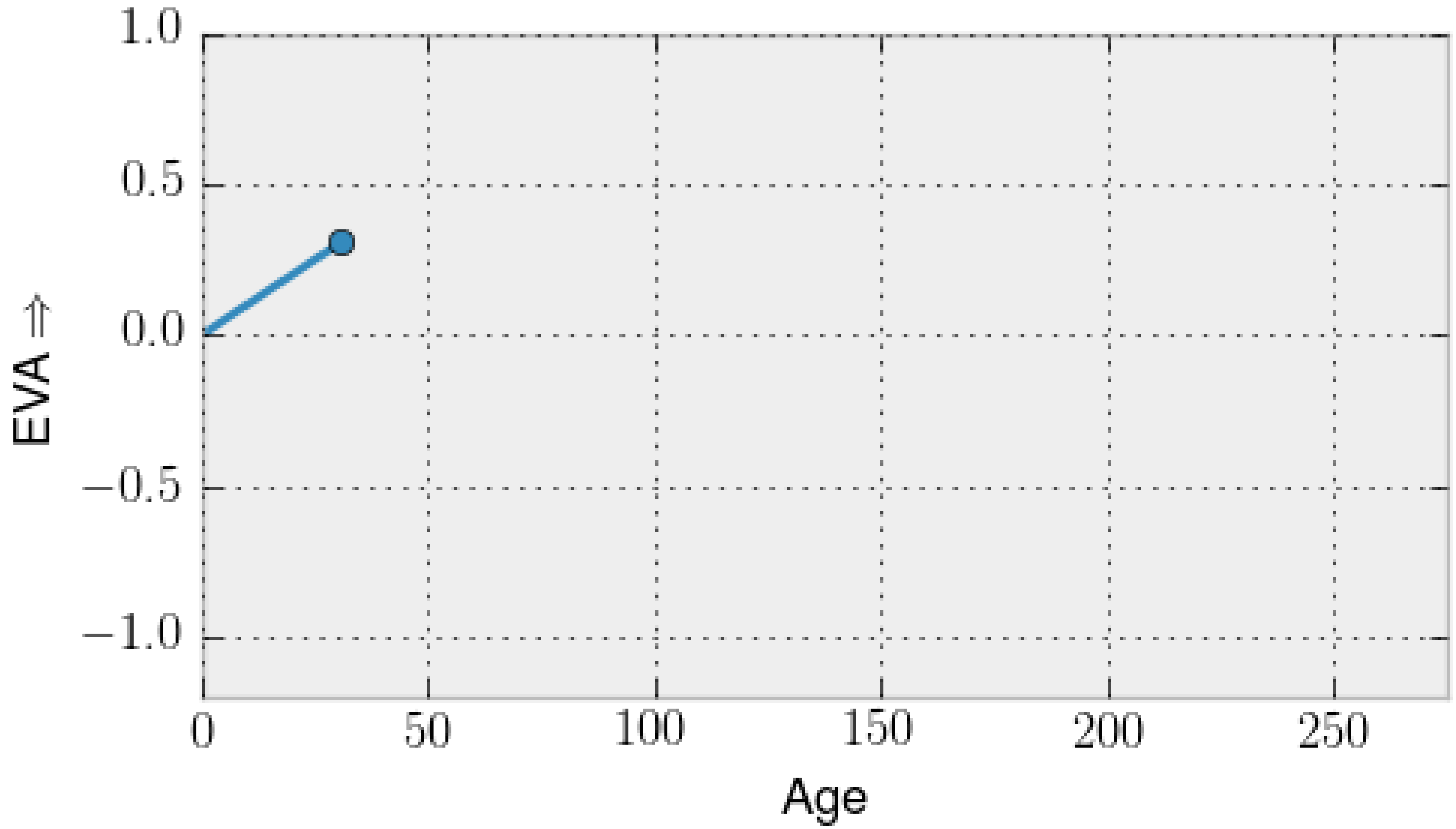
EVA policy on example (1/4)



At age zero, the replacement policy has learned **nothing** about the candidate.

Therefore, its EVA is **zero** – i.e., no difference from the average candidate.

EVA policy on example (2/4)

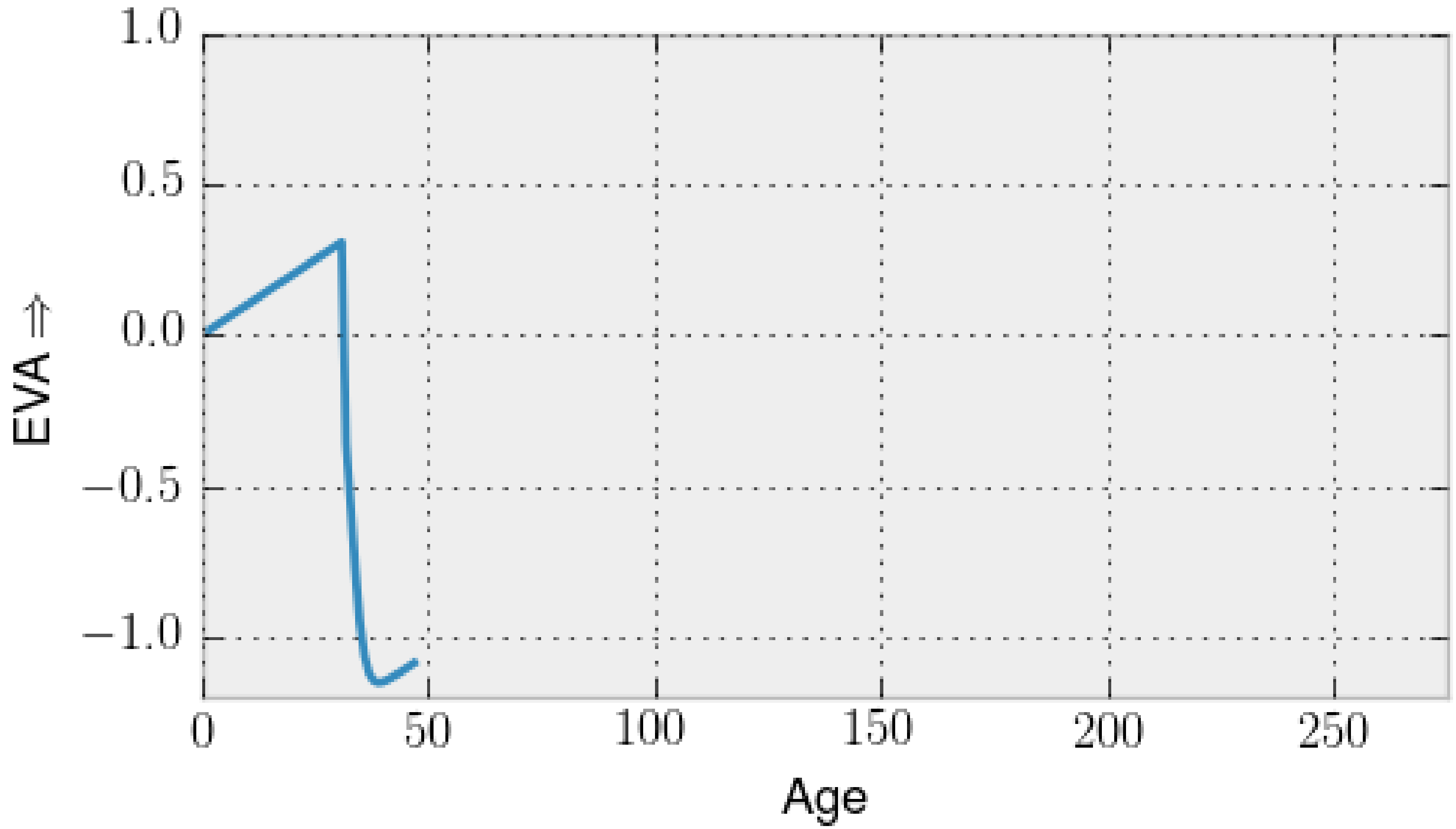


Until size of small array, EVA doesn’t know which array is being accessed.

But **expected remaining lifetime** decreases ➔ EVA increases.

EVA evicts MRU here, **protecting** candidates.

EVA policy on example (3/4)

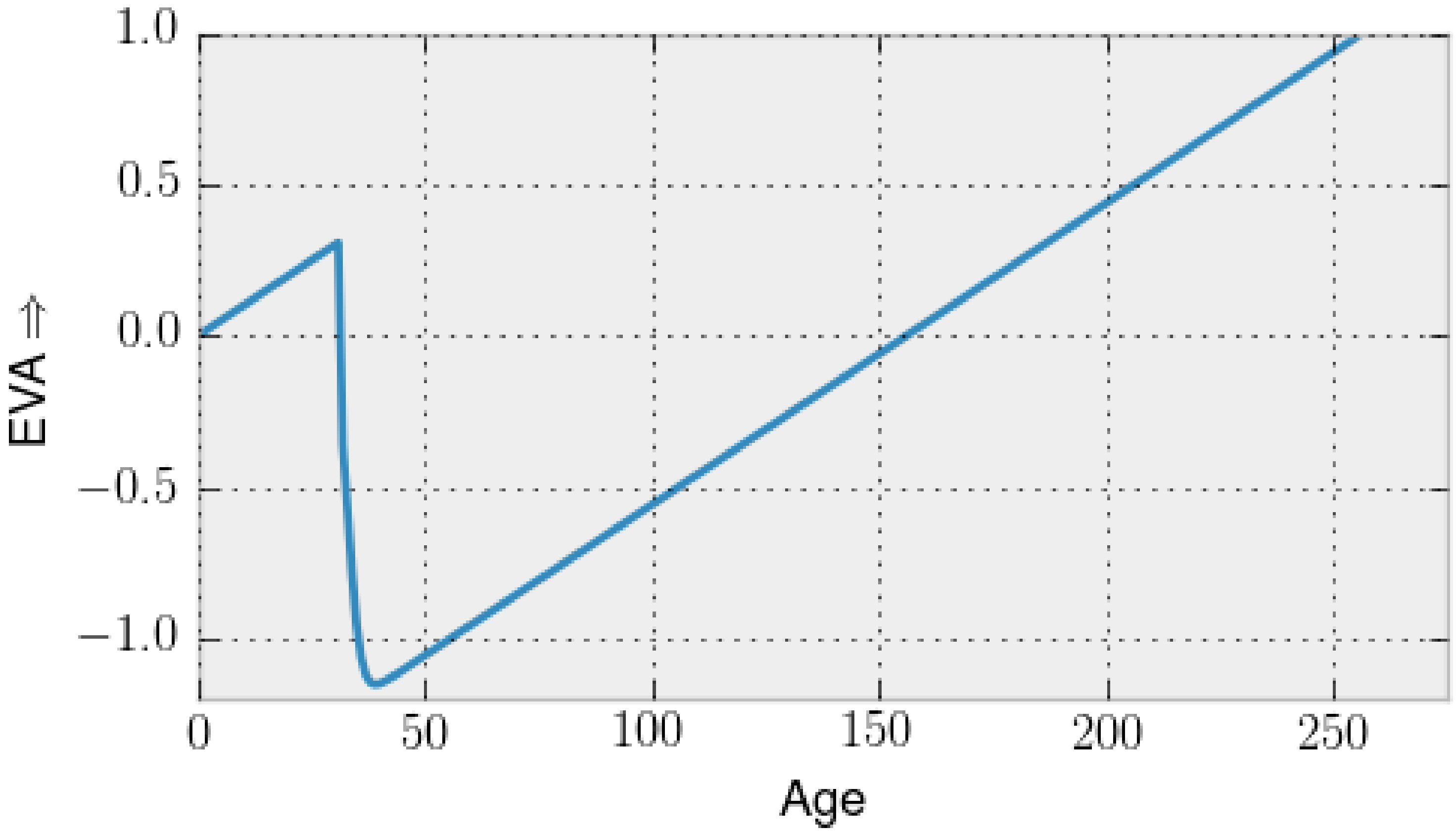


If candidate doesn't hit at size of small array, it must be an access to the big array.

So **expected remaining lifetime** is large, and **EVA is negative**.

EVA prefers to evict these candidates.

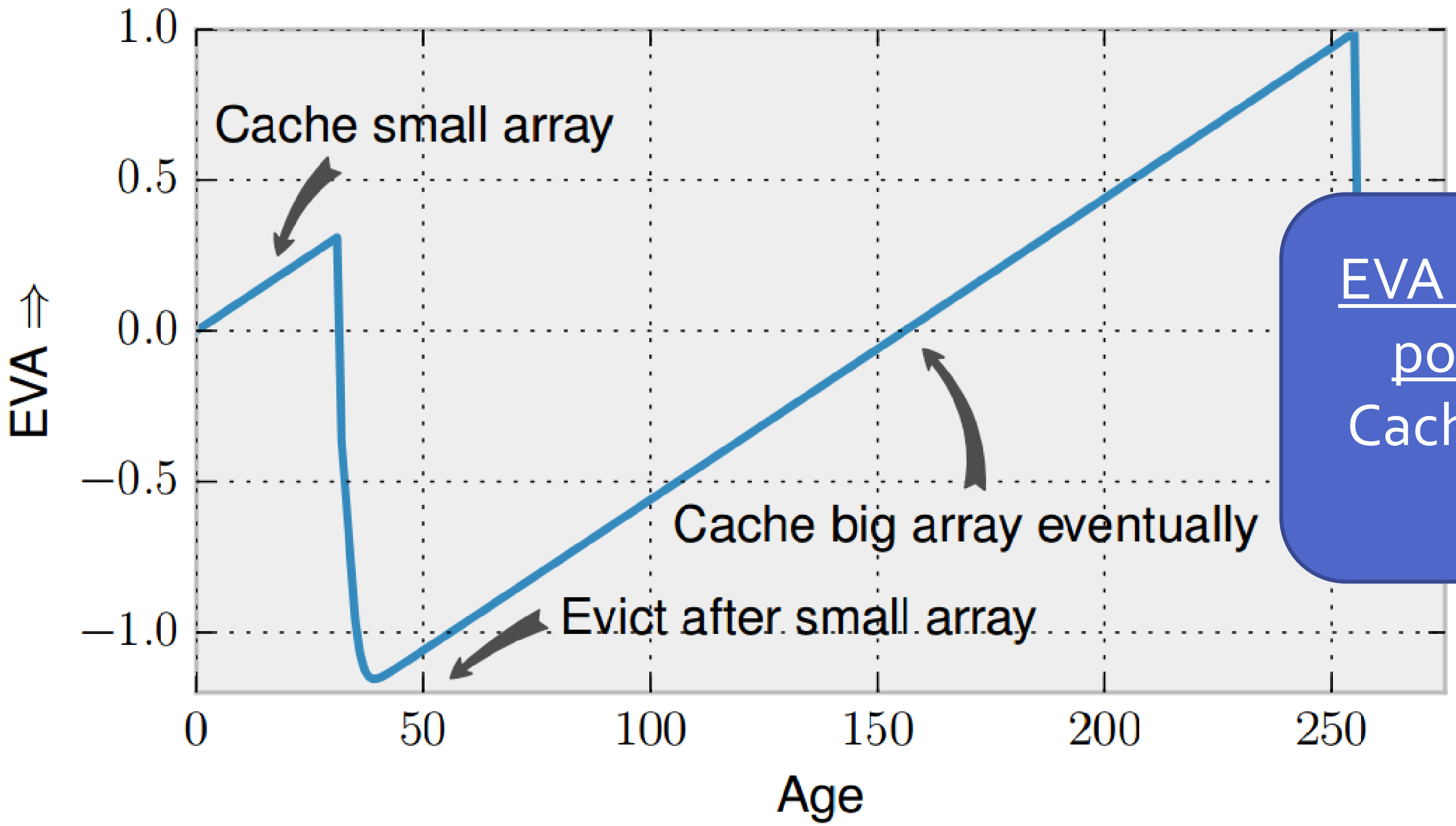
EVA policy on example (4/4)



Candidates that survive further are guaranteed to hit, but it takes a long time.

As remaining lifetime decreases, EVA increases to maximum of ≈1 at size of big array.

EVA policy summary



EVA implements the optimal policy given uncertainty:

Cache small array + as much of big array as fits

WHY IS EVA THE RIGHT METRIC?

Markov decision processes

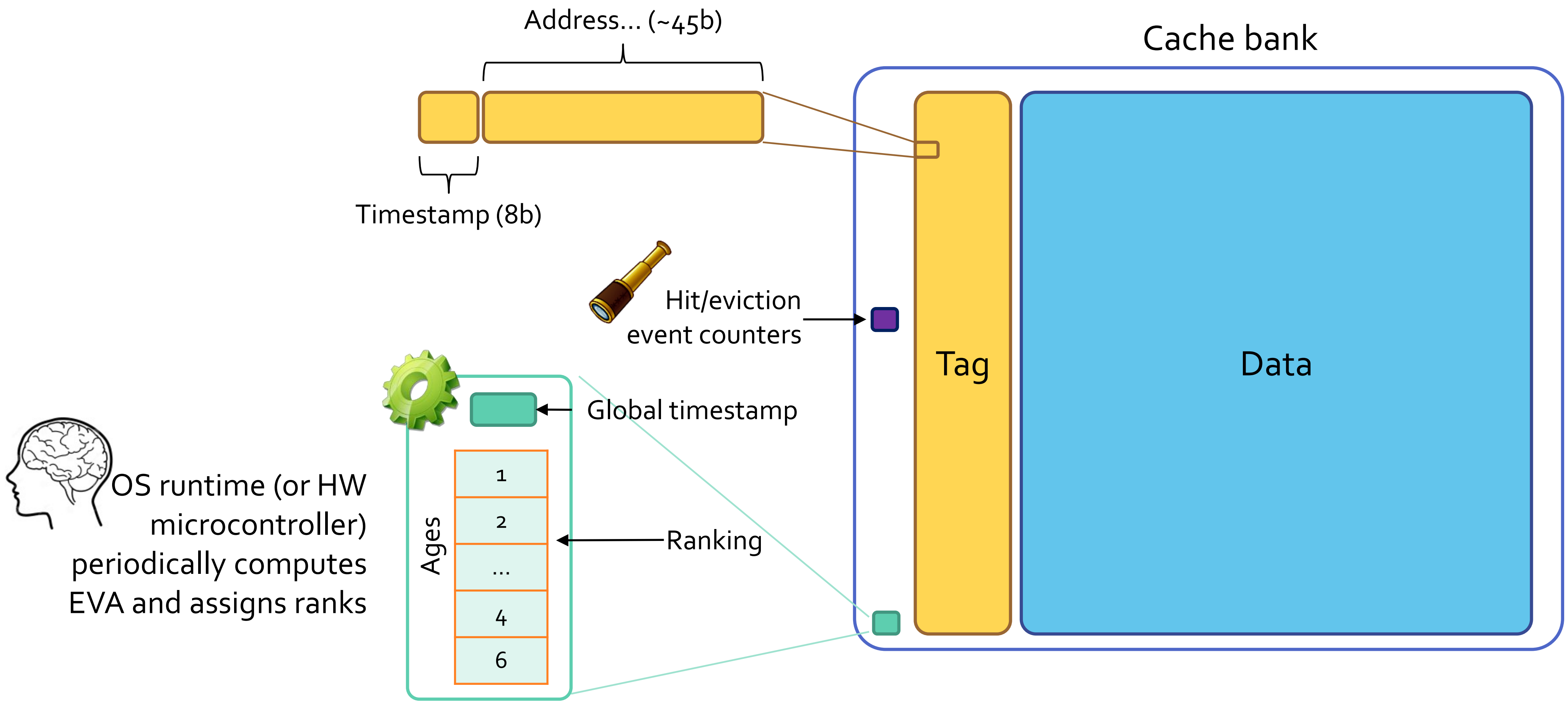
- Markov decision processes (MDPs) model decision-making under uncertainty
- MDP theory gives provably optimal decision-making metrics

- We can model cache replacement as an MDP
- EVA corresponds to a decomposition of the appropriate MDP policy

- (Paper gives high-level discussion & intuition; my PhD thesis gives details)
  Happy to discuss in depth offline!

TRANSLATING THEORY TO PRACTICE

Simple hardware, smart software



Updating EVA ranks

- Assign ranks to order $(age, reused?)$ by EVA

- Simple implementation in three passes over ages + sorting:
  1. Compute miss probabilities
  2. Compute unclassified EVA
  3. Add classification term

- Low complexity in software
  - 123 lines of C++

- …or a HW controller (0.05mm^2 @ 65nm)

Algorithm 1. Algorithm to compute EVA and update ranks.

Inputs: hitCtrs, evictionCtrs — event counters, A — age granularity
Returns: rank — eviction priorities for all ages and classes

```
function UPDATE
    for a ← 2^k to 1 :                          ▷ Miss rates from summing over counters.
        for c ∈ {nonReused, reused} :
            hits_c += hitCtrs[c,a]
            misses_c += evictionCtrs[c,a]
        m_R[a] ← misses_R/(hits_R + misses_R)
        m_NR[a] ← misses_NR/(hits_NR + misses_NR)
    m ← (hits_R + hits_NR)/(misses_R + misses_NR)
    perAccessCost ← (1 − m) × A/S
    for c ∈ {nonReused, reused} :                ▷ Compute EVA backwards over ages.
        expLifetime, hits, events ← 0
        for a ← 2^k to 1 :
            expectedLifetime += events
            eva[c,a] ← (hits − perAccessCost × expectedLifetime)/events
            hits += hitCtrs[c,a]
            events += hitCtrs[c,a] + evictionCtrs[c,a]
    evaReused ← eva[reused, 1]/m_R[0]           ▷ Differentiate classes.
    for c ∈ {nonReused, reused} :
        for a ← 2^k to 1 :
            eva[c,a] += (m − m_c[a]) × evaReused
    order ← ARGSORT(eva)                         ▷ Finally, rank ages by EVA.
    for i ← 1 to 2^(k+1) :
        rank[order[i]] ← 2^(k+1) − i
    return rank
```

Overheads

- Software updates
  - 43Kcycles / 256K accesses
  - Average **0.1%** overhead

- Hardware structures
  - **1%** area overhead (mostly tags)
  - **7mW** with frequent accesses

Easy to reduce further with little performance loss.

EVALUATION

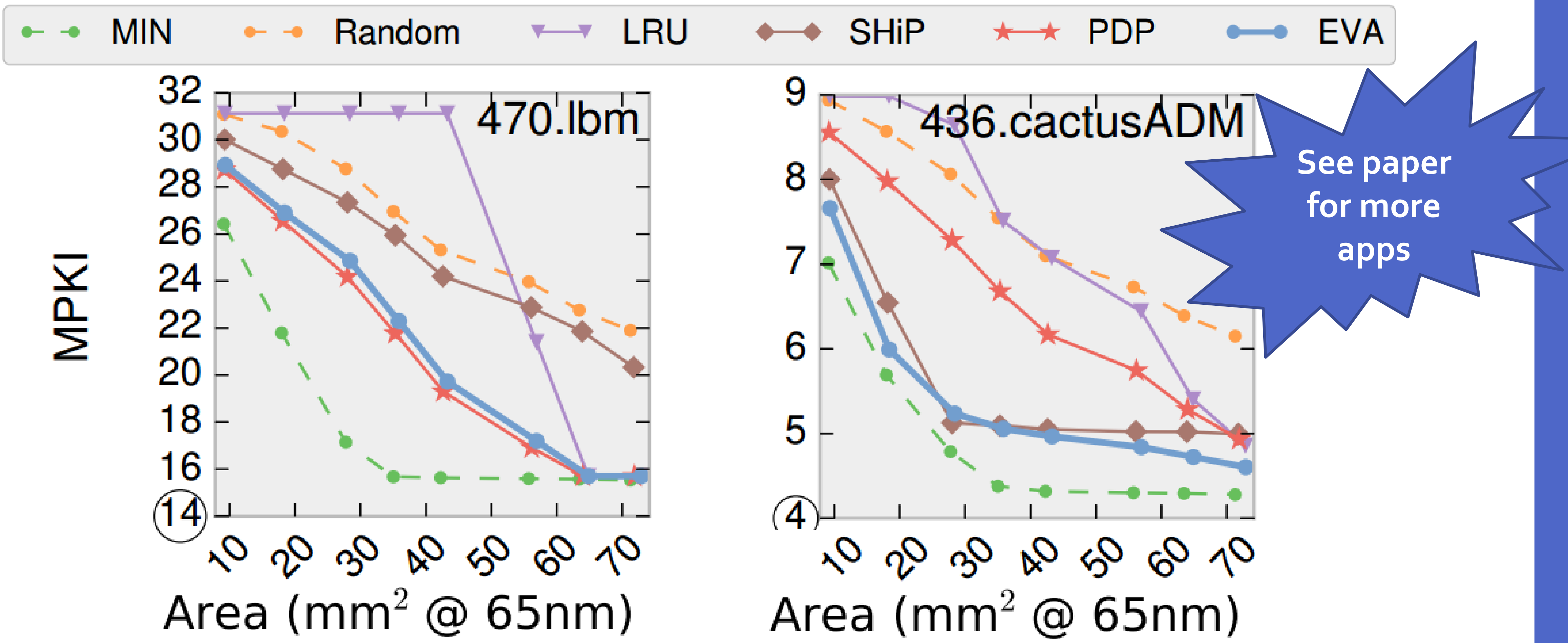
Methodology

- Simulation using zsim
- Workloads: SPECCPU2006 (multithreaded in paper)
- System: 4GHz OOO, 32KB L1s & 256KB L2

- **Study replacement policy in L3 from 1MB → 8MB**
  - EVA vs random, LRU, SHiP [Wu MICRO'11], PDP [Duong MICRO'12]

- **Compare *performance* vs. *total cache area***
  - Including replacement, ≈1% of total area

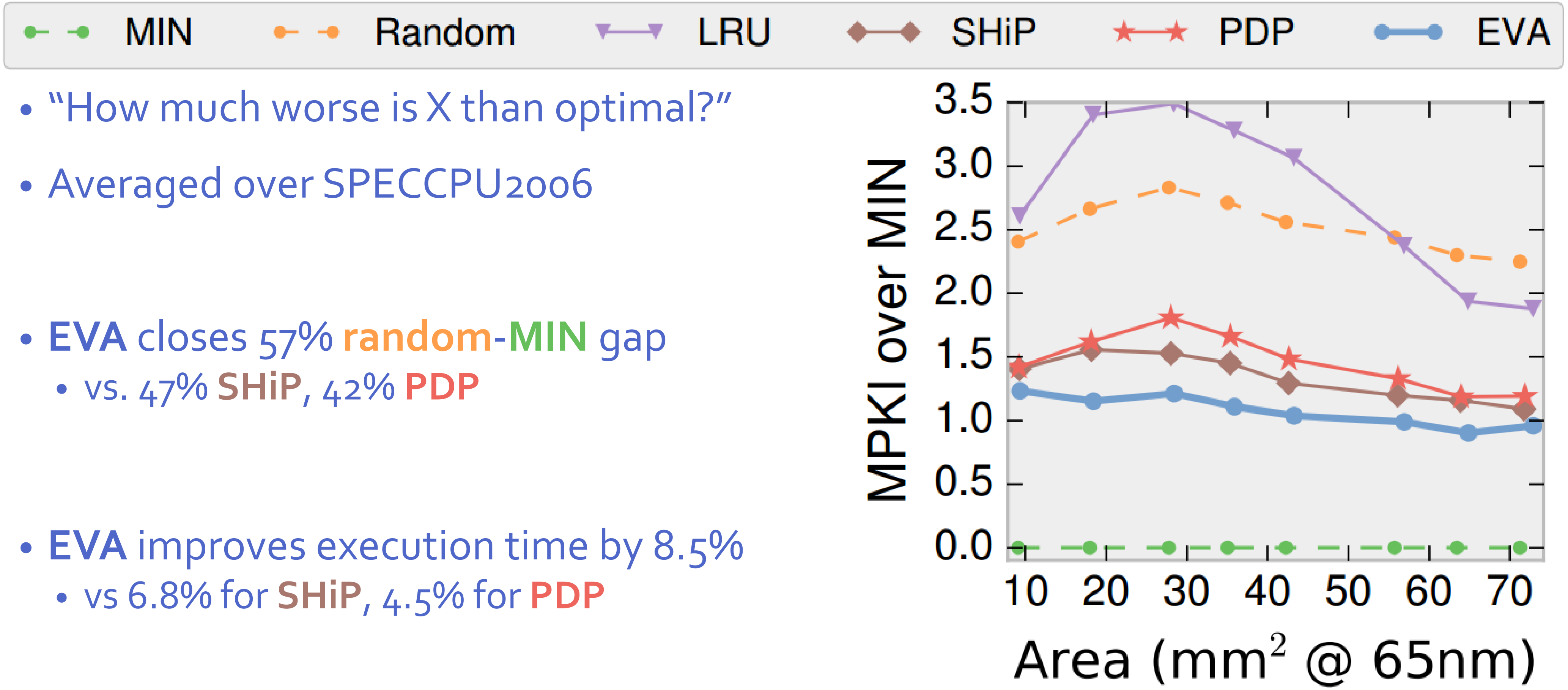
EVA performs consistently well



SHiP performs poorly

PDP performs poorly

EVA closes gap to optimal replacement

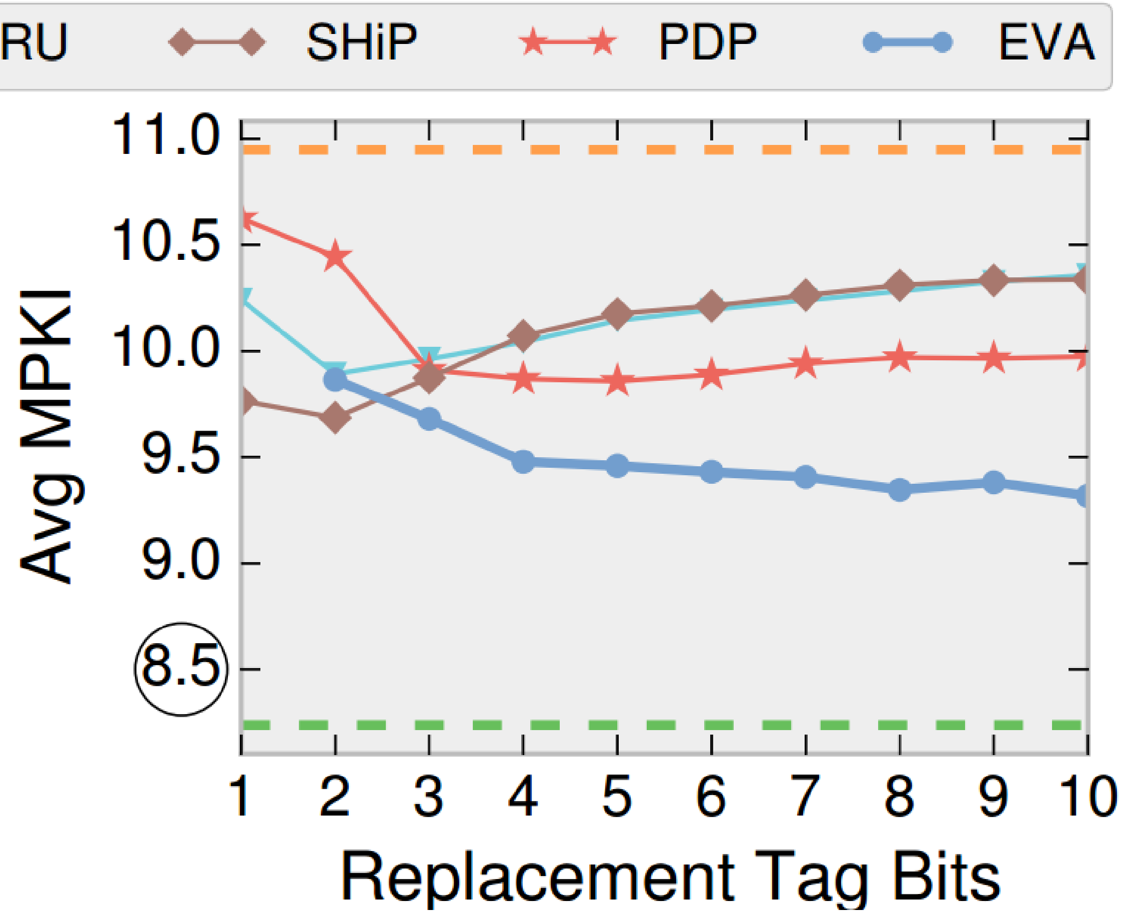


- "How much worse is X than optimal?"
- Averaged over SPECCPU2006

- **EVA** closes 57% **random**-**MIN** gap
  - vs. 47% **SHiP**, 42% **PDP**

- **EVA** improves execution time by 8.5%
  - vs 6.8% for **SHiP**, 4.5% for **PDP**

EVA makes good use of add'l state



- Adding bits improves **EVA**'s perf.
  - Not true of **SHiP**, **PDP**, **DRRIP**
- ➔ Even with larger tags, ***EVA*** *saves 8% area vs* ***SHiP***
- Open question: how much space should we spend on replacement?
  - Traditionally: as little as possible
  - But is this the best tradeoff?

EVA is easy to apply to new problems

Just change **cost**/**benefit** terms in **EVA** to adapt to…

- Objects of different size (eg, compressed caches)
- Different optimization metrics (eg, byte-hit-rate)
- QoS or application priorities
- …and so on

THANK YOU!