LHD: IMPROVING CACHE HIT RATE BY MAXIMIZING HIT DENSITY

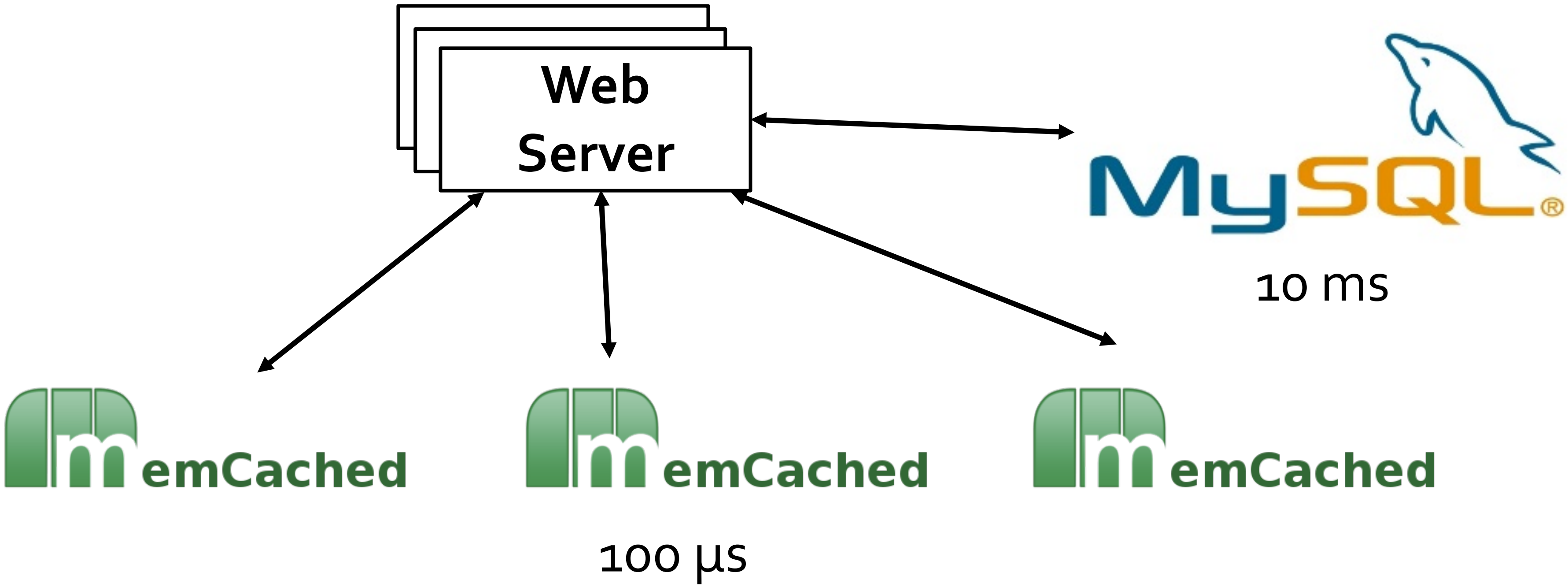
Nathan Beckmann
CMU

Haoxian Chen
U. Penn

Asaf Cidon
Stanford &
Barracuda Networks

USENIX NSDI 2018

Key-value cache is 100X faster than database



Key-value cache hit rate determines web application performance

- At 98% cache hit rate:
- +1% hit rate → 35% speedup
  - Old latency: 374 µs
  - New latency: 278 µs
  - Facebook study [Atikoglu, Sigmetrics '12]
- Even small hit rate improvements cause significant speedup

Choosing the right eviction policy is hard

- Key-value caches have unique challenges
  - **Variable object sizes**
  - Variable workloads
- Prior policies are heuristics that combine **recency** and **frequency**
  - No theoretical foundation
  - Require hand-tuning ➔ fragile to workload changes
- No policy works for all workloads
  - Prior system simulates many cache policy configurations to find right one per workload [Waldspurger, ATC '17]

GOAL:
AUTO-TUNING EVICTION POLICY ACROSS WORKLOADS

The “big picture” of key-value caching

- *Goal:* Maximize cache hit rate

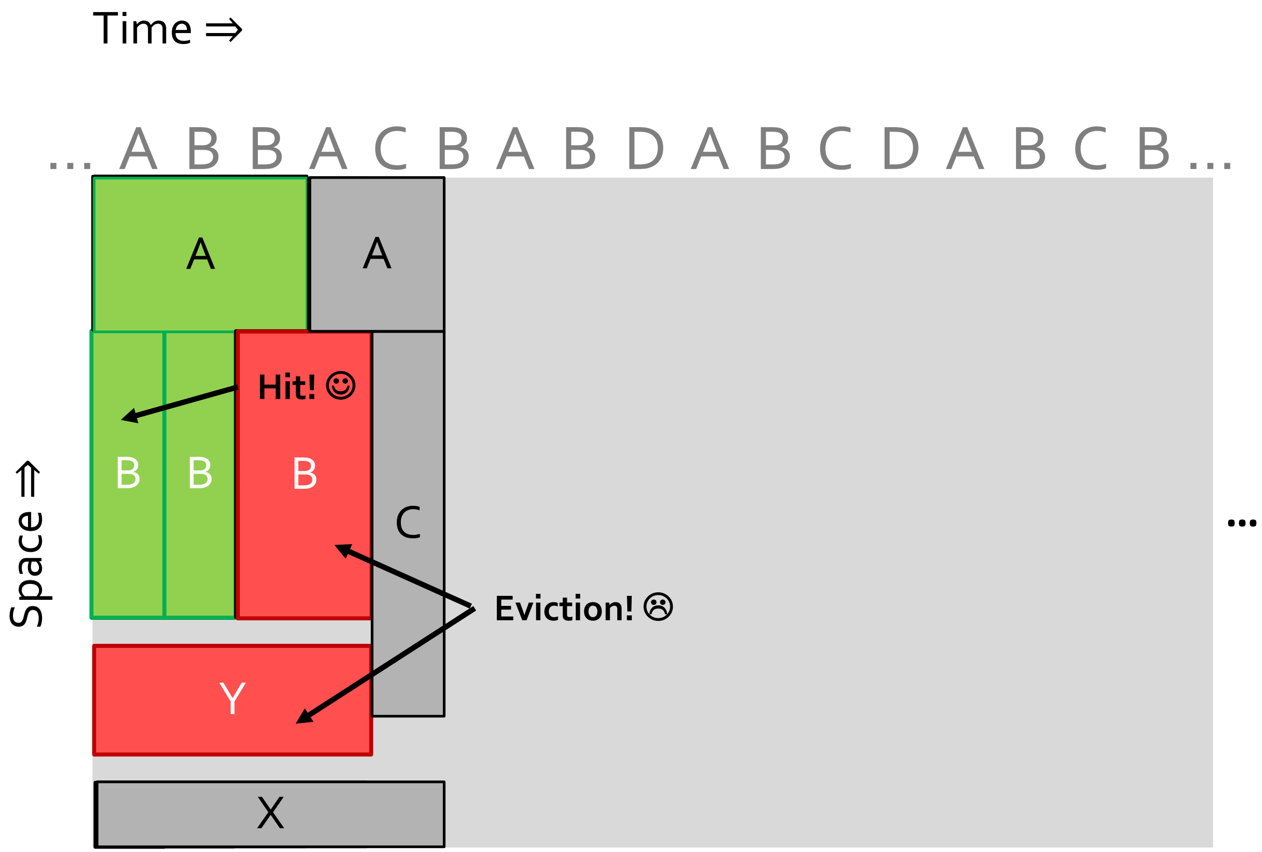
- *Constraint:* Limited cache space

- *Uncertainty*: In practice, don’t know what is accessed when

- *Difficulty:* Objects have variable sizes

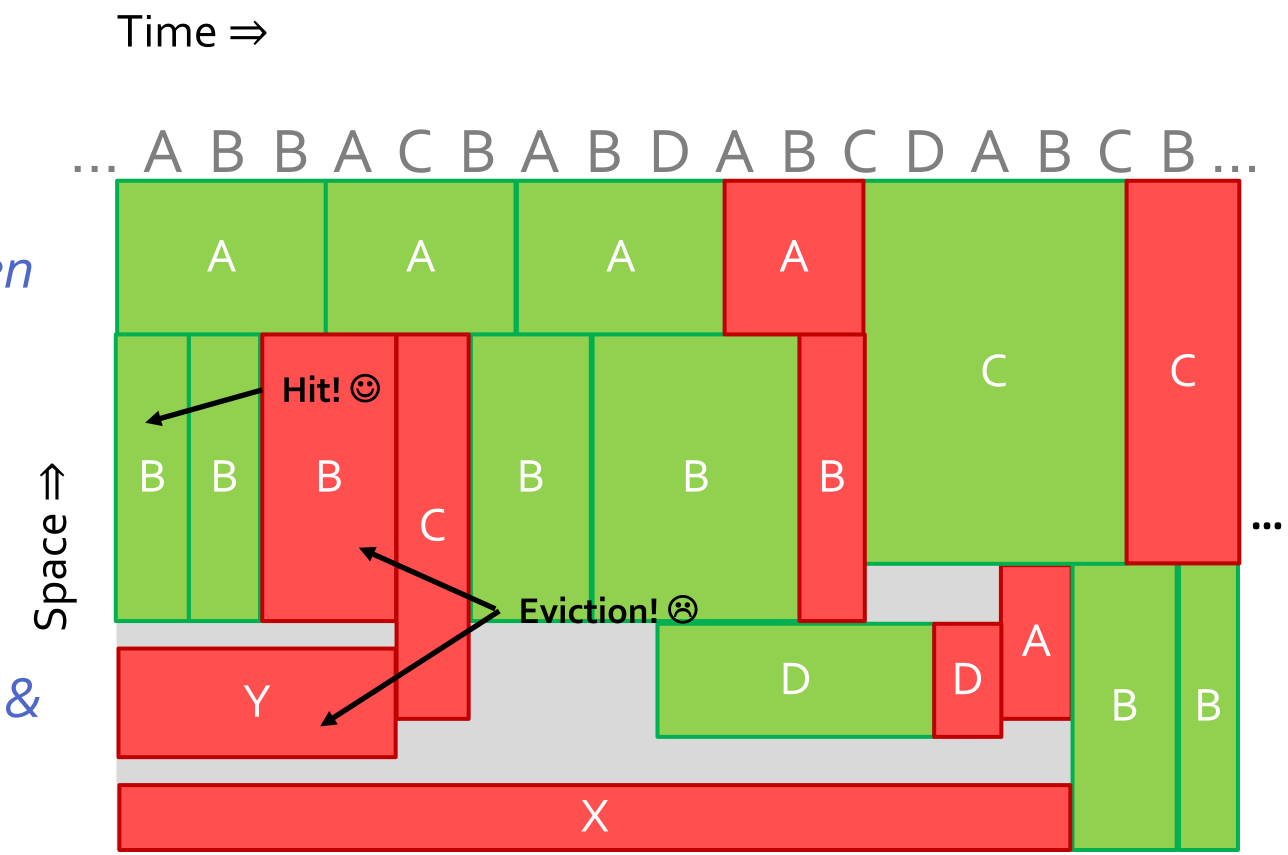
Where does cache space go?

- Let's see what happens on a short trace…



Where does cache space go?

- Green box = 1 hit
- Red box = 0 hits
- ➔ *Want to fit as many green boxes as possible*
- Each box costs resources = area
- ➔ *Cost proportional to size & time spent in cache*



THE KEY IDEA:
HIT DENSITY

Our metric: Hit density (HD)

- Hit density combines **hit probability** and **expected cost**

$$\text{Hit density} = \frac{\boldsymbol{Object's}\ \textbf{hit probability}}{Object's\ \text{size} \times \boldsymbol{Object's}\ \textbf{expected lifetime}}$$

- Least hit density (LHD) policy: Evict object with smallest hit density
- **But how do we predict these quantities?**

Estimating hit density (HD)

- Age – # accesses since object was last requested
- Random variables
  - H – hit age (e.g., $\mathrm{P}[H = 100]$ is probability an object hits after 100 accesses)
  - L – lifetime (e.g., $\mathrm{P}[\mathrm{L} = 100]$ is probability an object hits *or is evicted* after 100 accesses)
- Easy to estimate HD from these quantities:

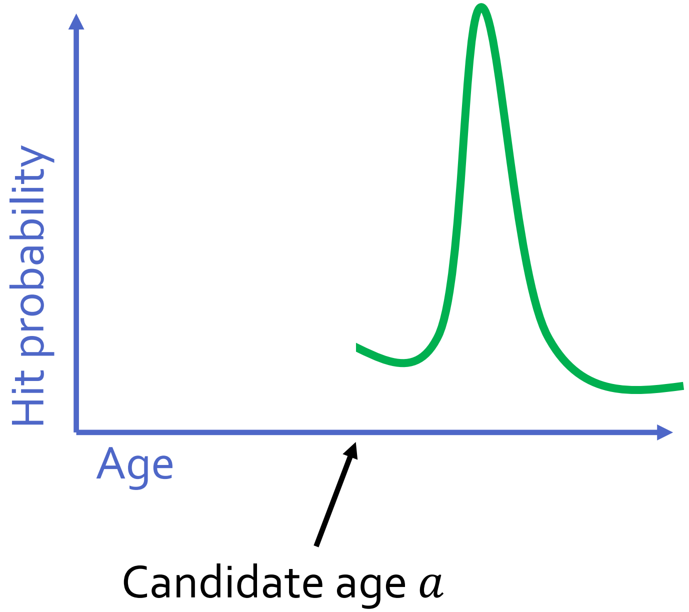
hit probability

$$HD = \frac{\sum_{a=1}^{\infty} \mathrm{P}[H = a]}{Size \times \sum_{a=1}^{\infty} a\,\mathrm{P}[L = a]}$$

expected lifetime

Example: Estimating HD from object age

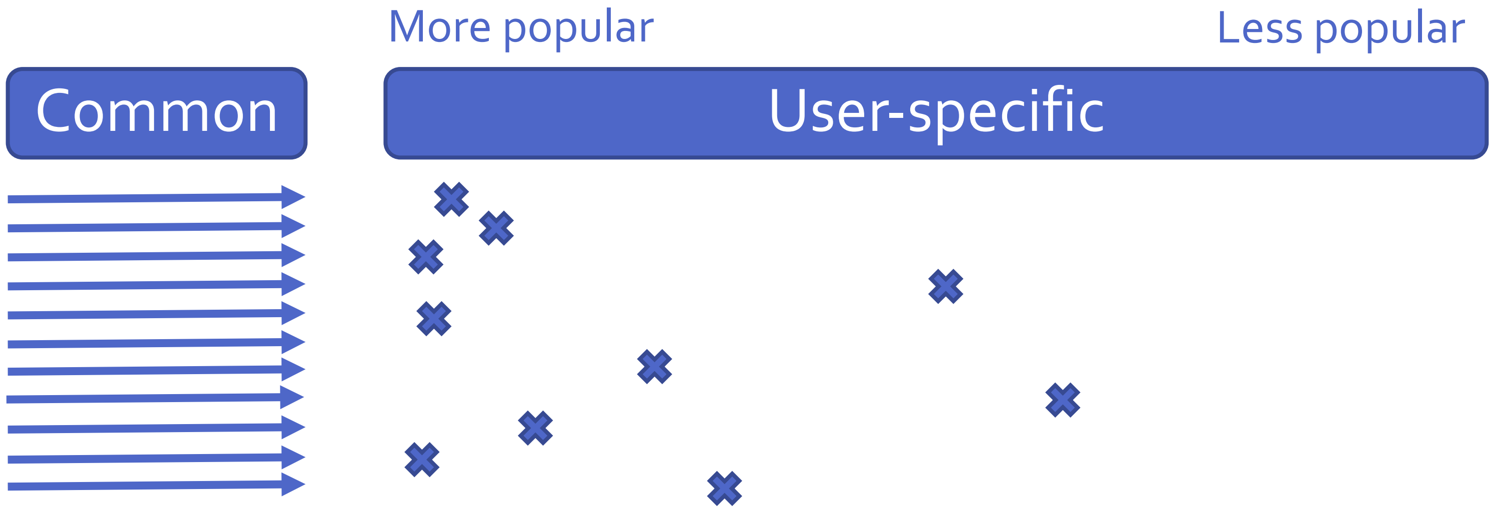
- Estimate HD using **conditional probability**
- Monitor distribution of H & L online
- By definition, object of age a wasn't requested at age $\leq a$
- ➔ Ignore all events before a



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

LHD by example

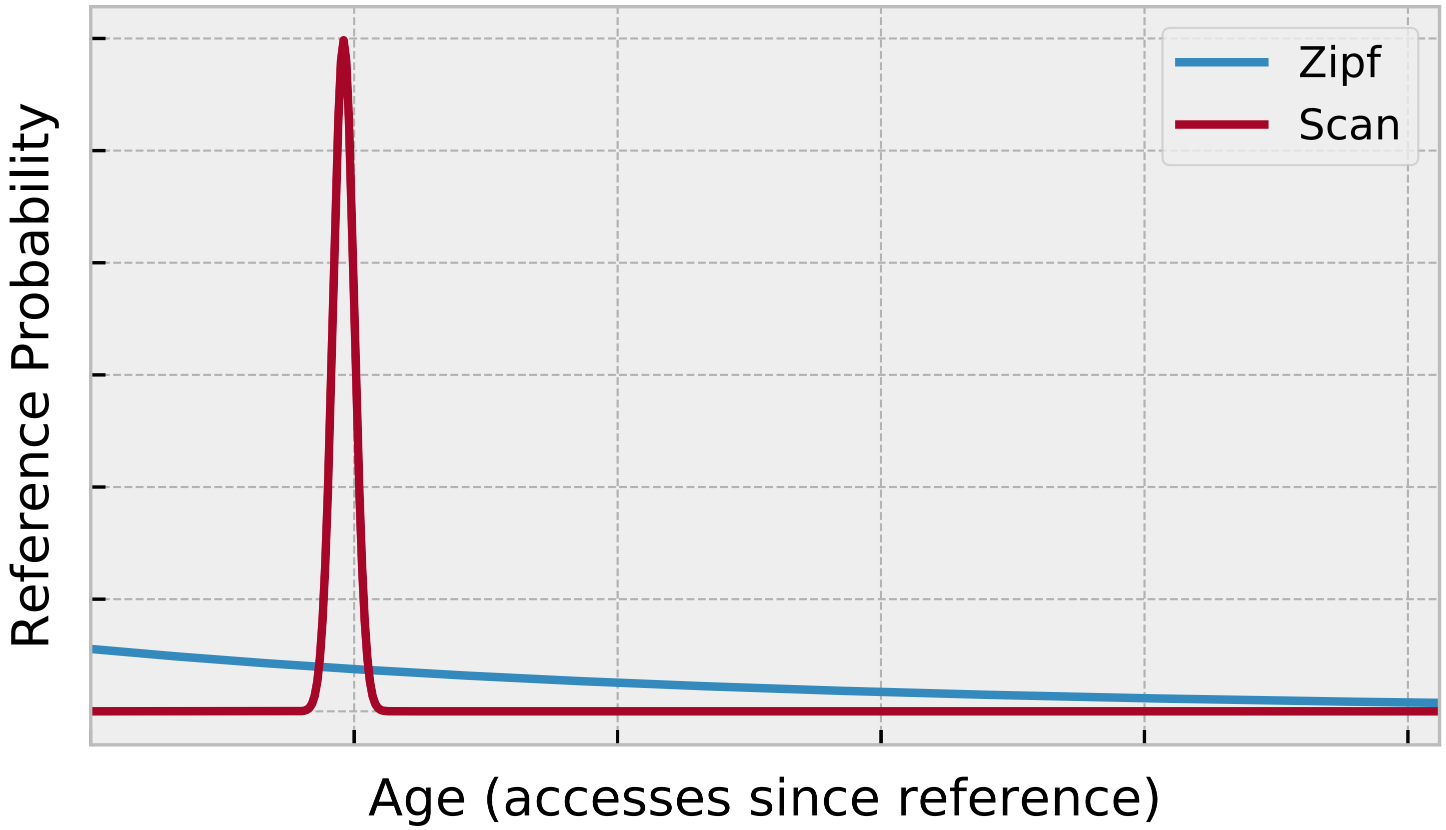
- Users ask repeatedly for common objects and some user-specific objects



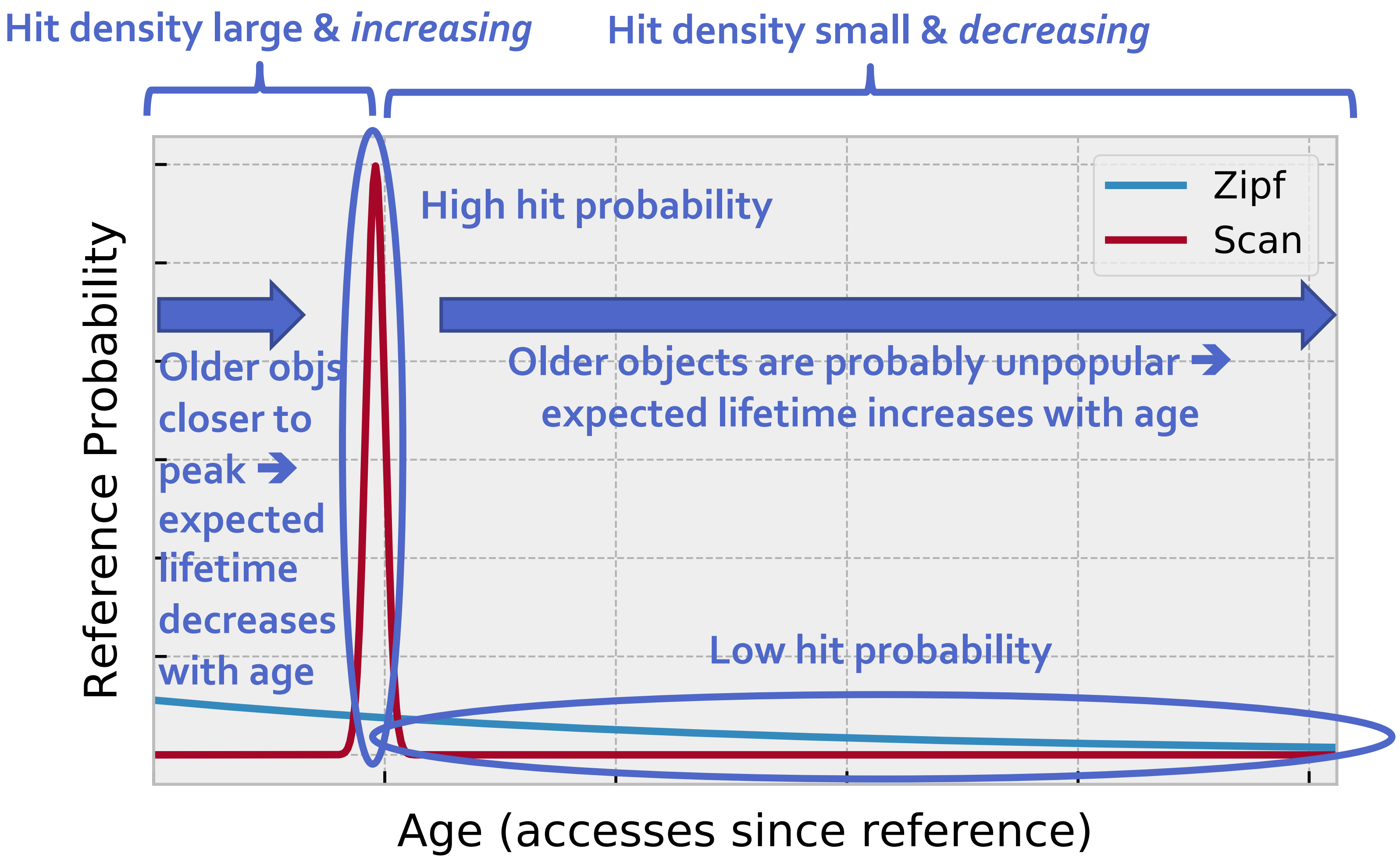
Best hand-tuned policy for this app:
Cache common media + as much user-specific as fits

Probability of referencing object again

- Common object modeled as scan, user-specific object modeled as Zipf



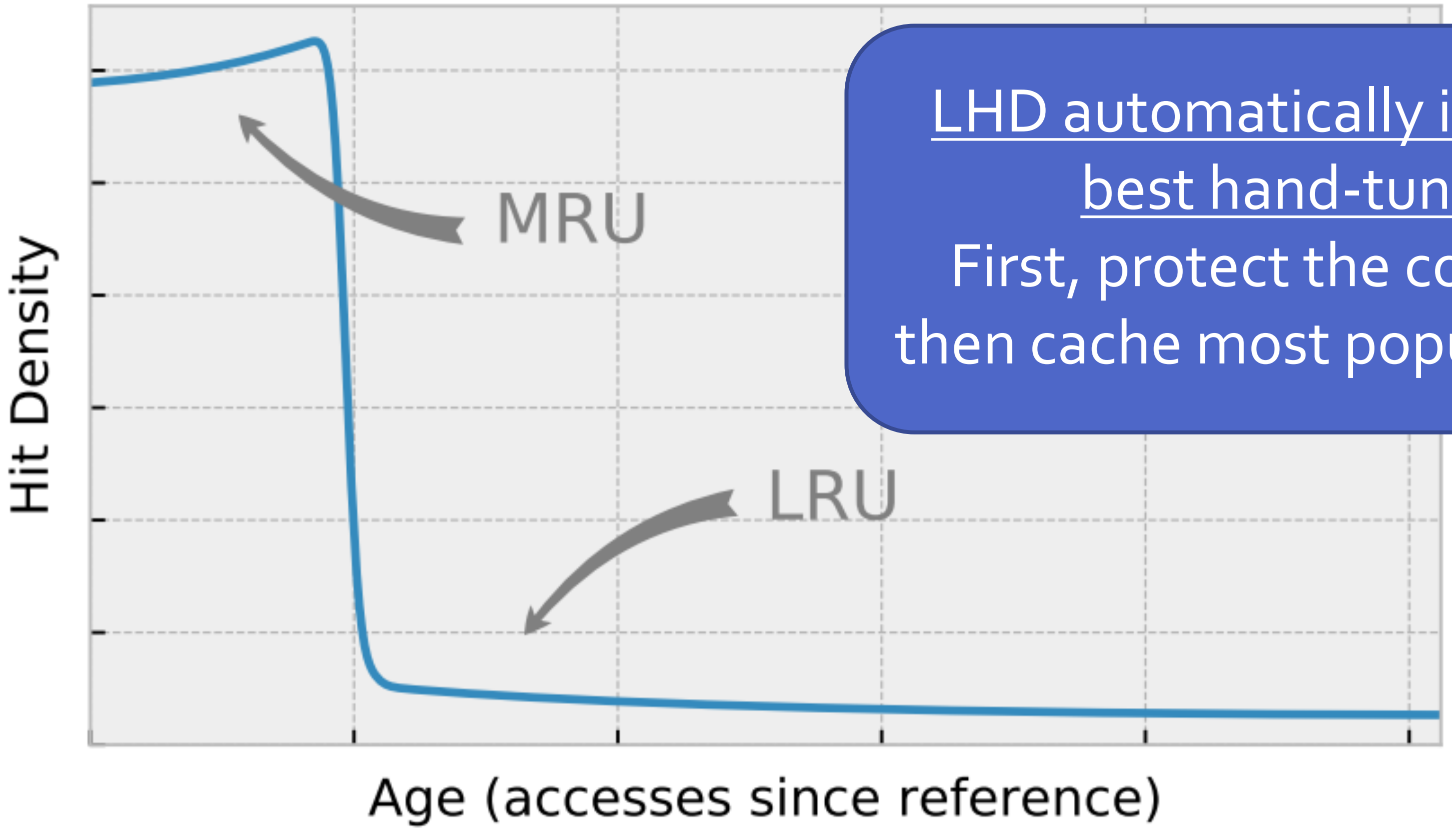
LHD by example: what's the hit density?



LHD by example: policy summary

Hit density large & *increasing*

Hit density small & *decreasing*



LHD automatically implements the best hand-tuned policy:
First, protect the common media, then cache most popular user content

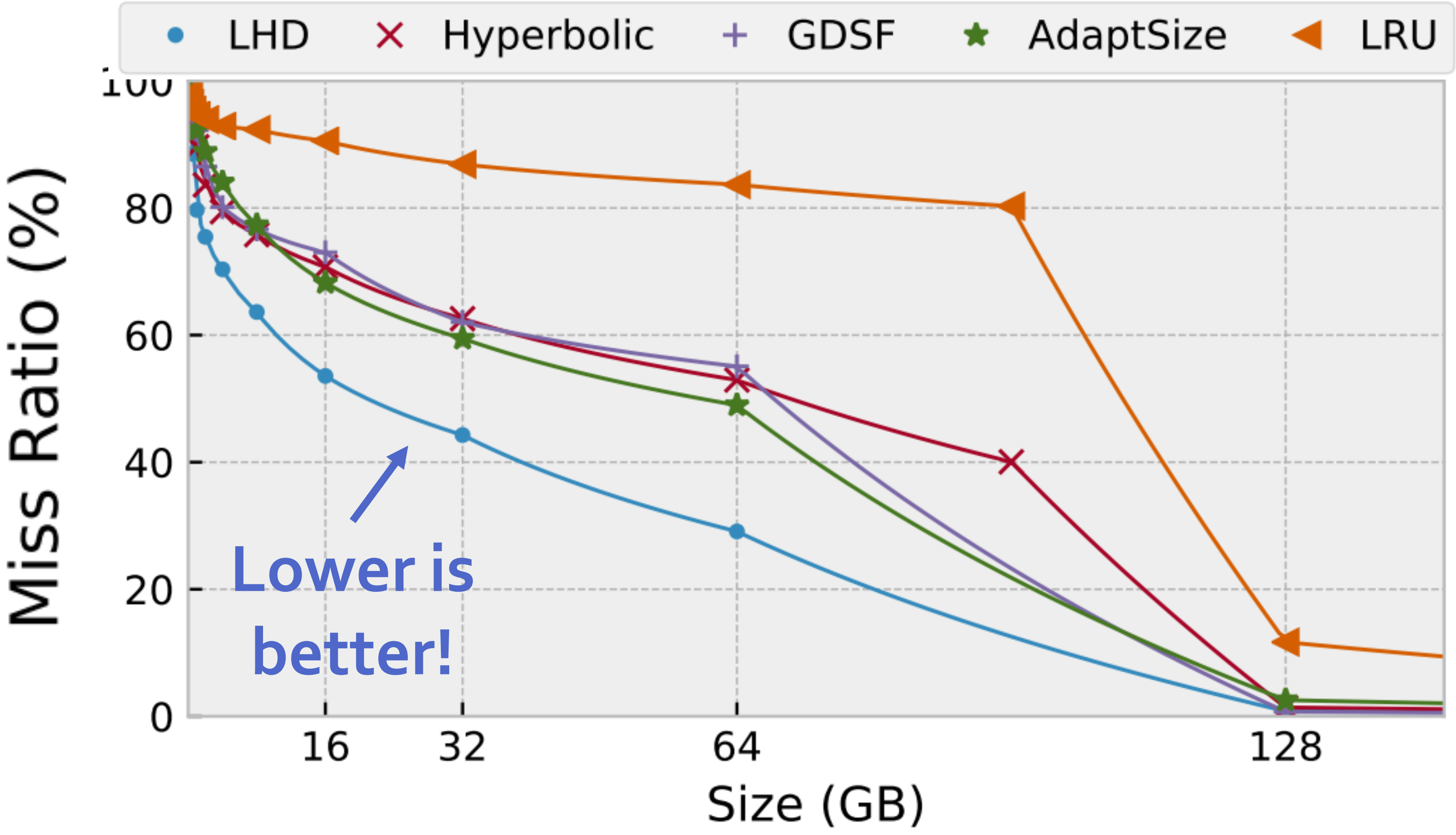
Improving LHD using additional object features

- Conditional probability lets us easily add information!

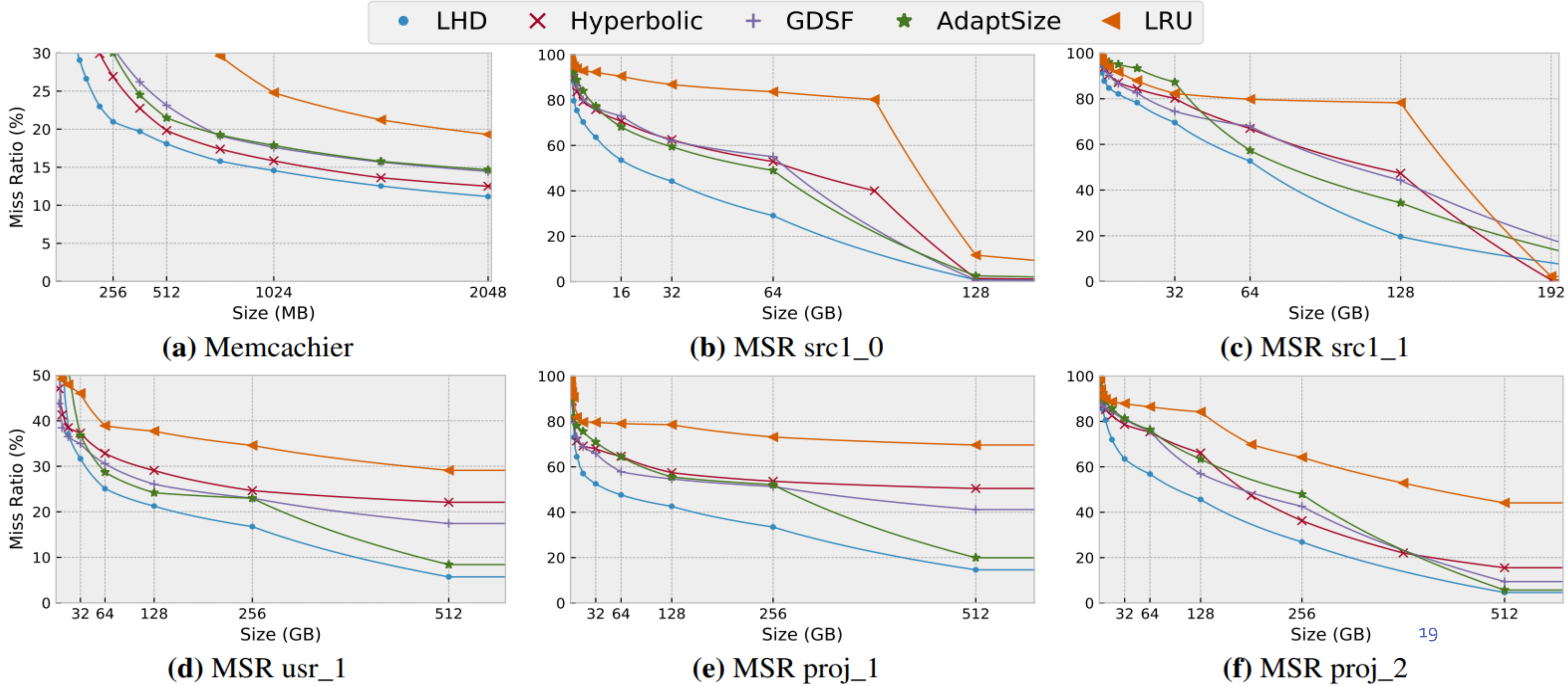
- Condition H & L upon additional informative object features, e.g.,
  - *Which app requested this object?*
  - *How long has this object taken to hit in the past?*

- Features inform decisions ➔ LHD *learns* the "right" policy
  - No hard-coded heuristics!

LHD gets more hits than prior policies

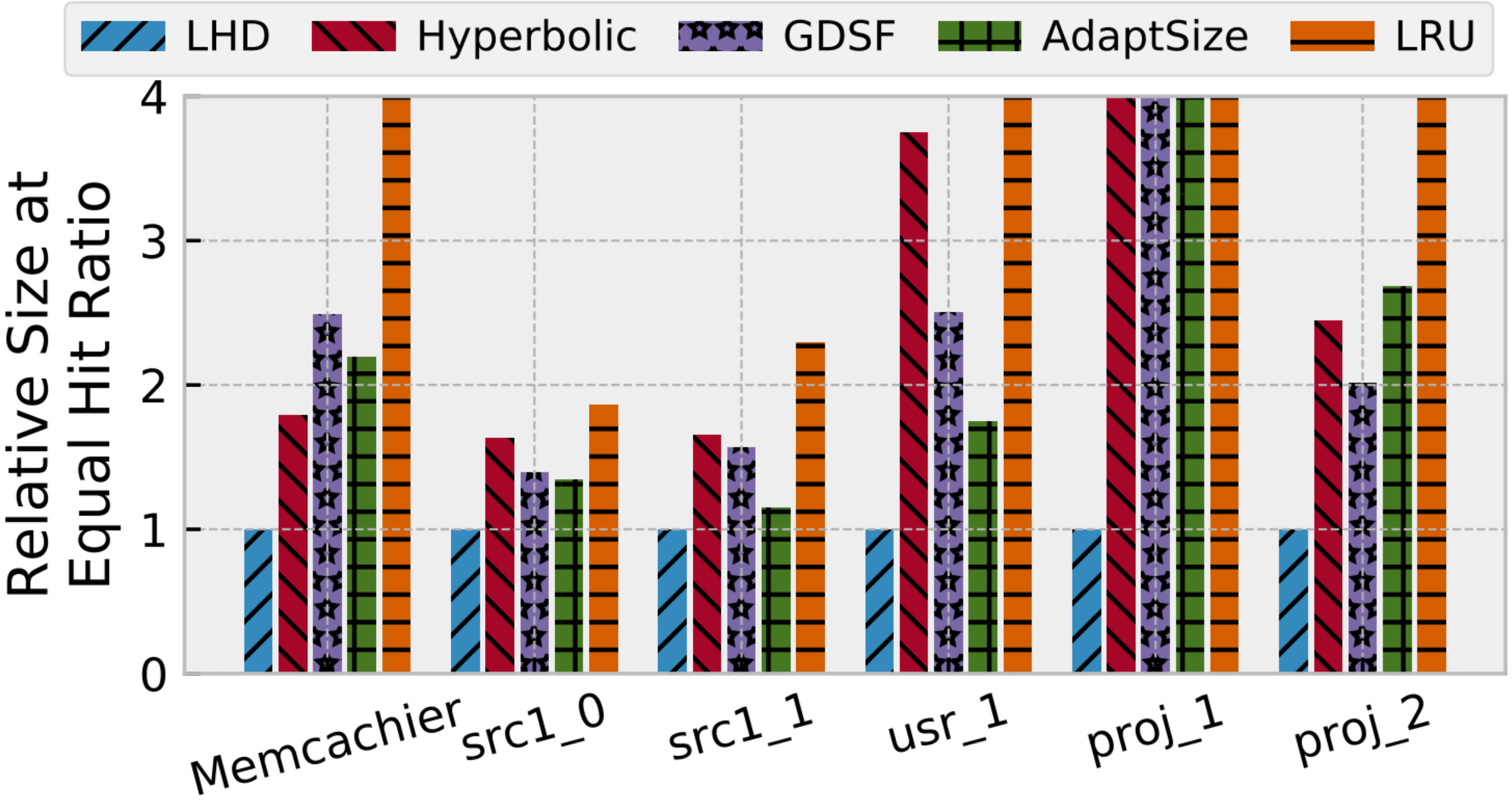


LHD gets more hits across many traces



(a) Memcachier
(b) MSR src1_0
(c) MSR src1_1
(d) MSR usr_1
(e) MSR proj_1
(f) MSR proj_2

LHD needs much less space

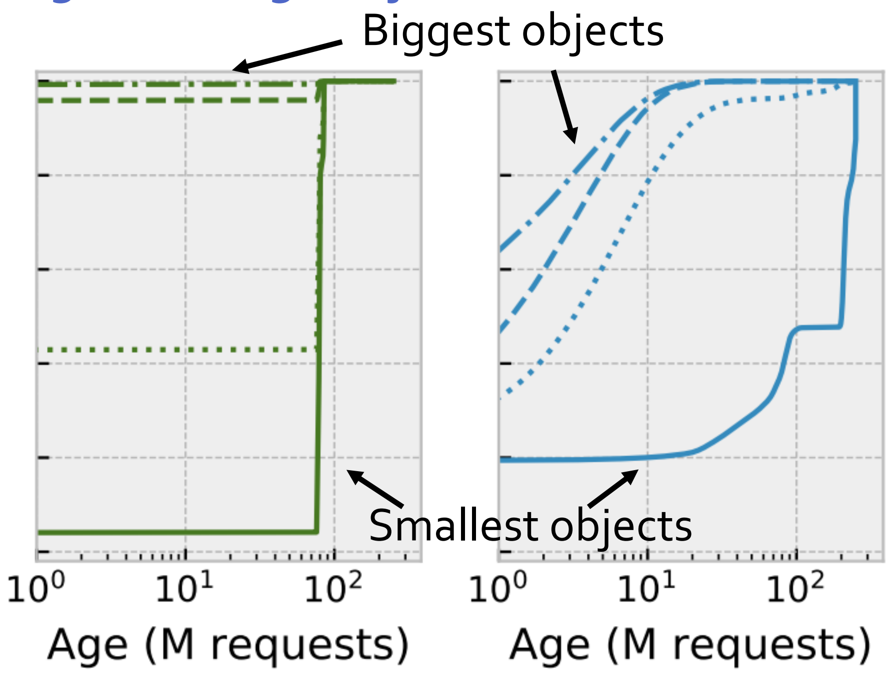


Why does LHD do better?

- Case study vs. AdaptSize [Berger et al, NSDI'17]
  - AdaptSize improves LRU by bypassing most large objects

LHD admits all objects ➔ more hits from big objects

LHD evicts big objects quickly ➔ small objects survive longer ➔ more hits



(b) AdaptSize. **(c)** LHD.

RANKCACHE: TRANSLATING THEORY TO PRACTICE

The problem

- Prior complex policies require complex data structures
- Synchronization ➔ poor scalability ➔ unacceptable request throughput

- Policies like GDSF require $O(\log N)$ heaps
- Even $O(1)$ LRU is sometimes too slow because of synchronization
- Many key-value systems approximate LRU with CLOCK / FIFO
  - MemC3 [Fan, NSDI '13], MICA [Lim, NSDI '14]...

- *Can LHD achieve similar request throughput to production systems?*

RankCache makes LHD fast

1. Track information approximately (eg, coarsen ages)

2. Precompute HD as table indexed by age & app id & etc

3. Randomly sample objects to find victim
    - Similar to Redis, Memshare [Cidon, ATC '17], [Psounis, INFOCOM '01],

4. Tolerate rare races in eviction policy

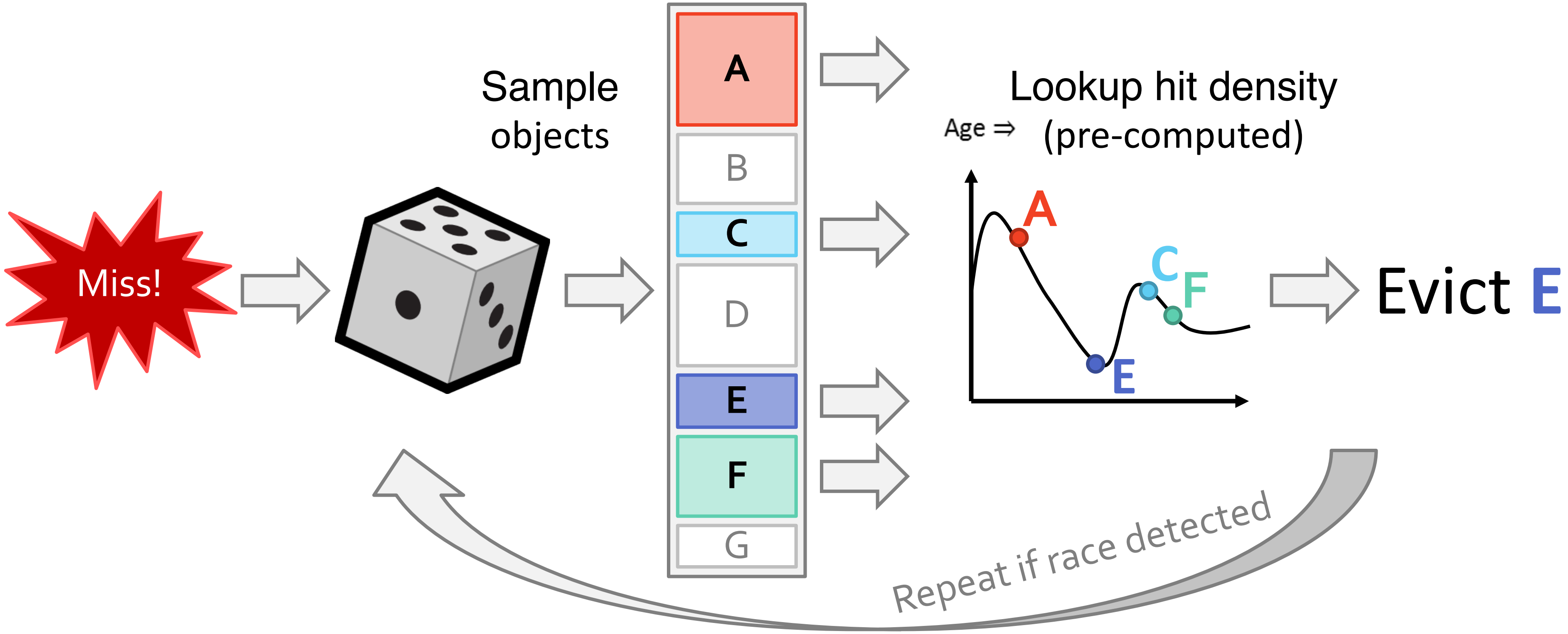
Making hits fast

- Metadata updated locally → no global data structure

- Same scalability benefits as CLOCK, FIFO vs. LRU

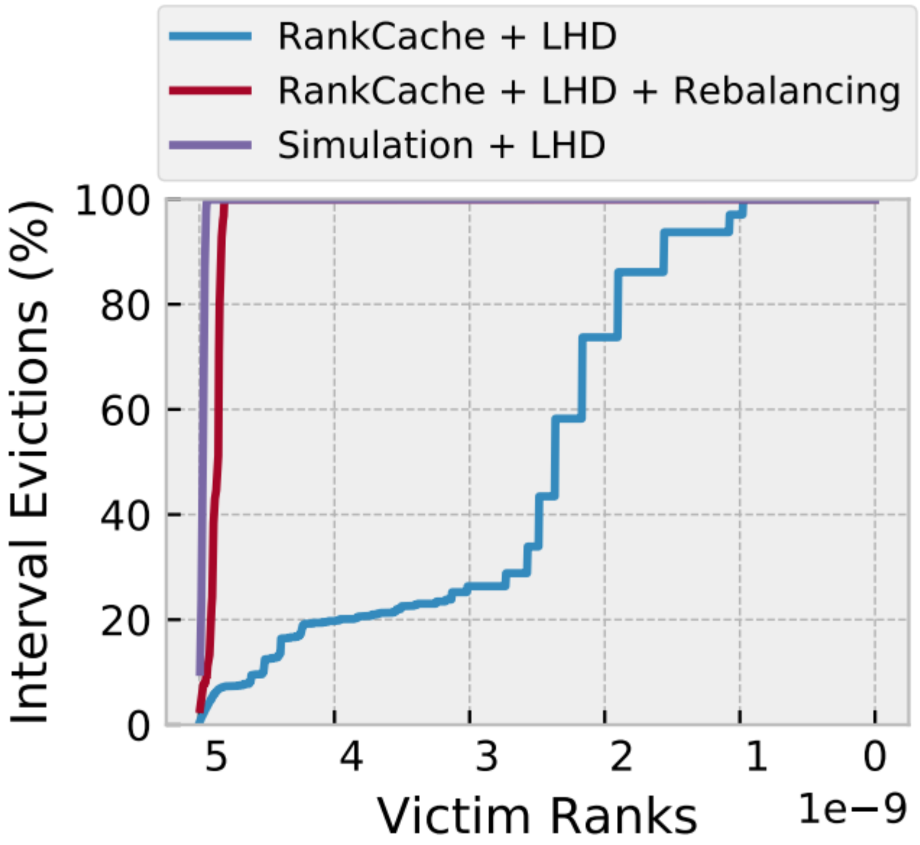
Making evictions fast

- No global synchronization ➔ Great scalability!
  (Even better than CLOCK/FIFO!)



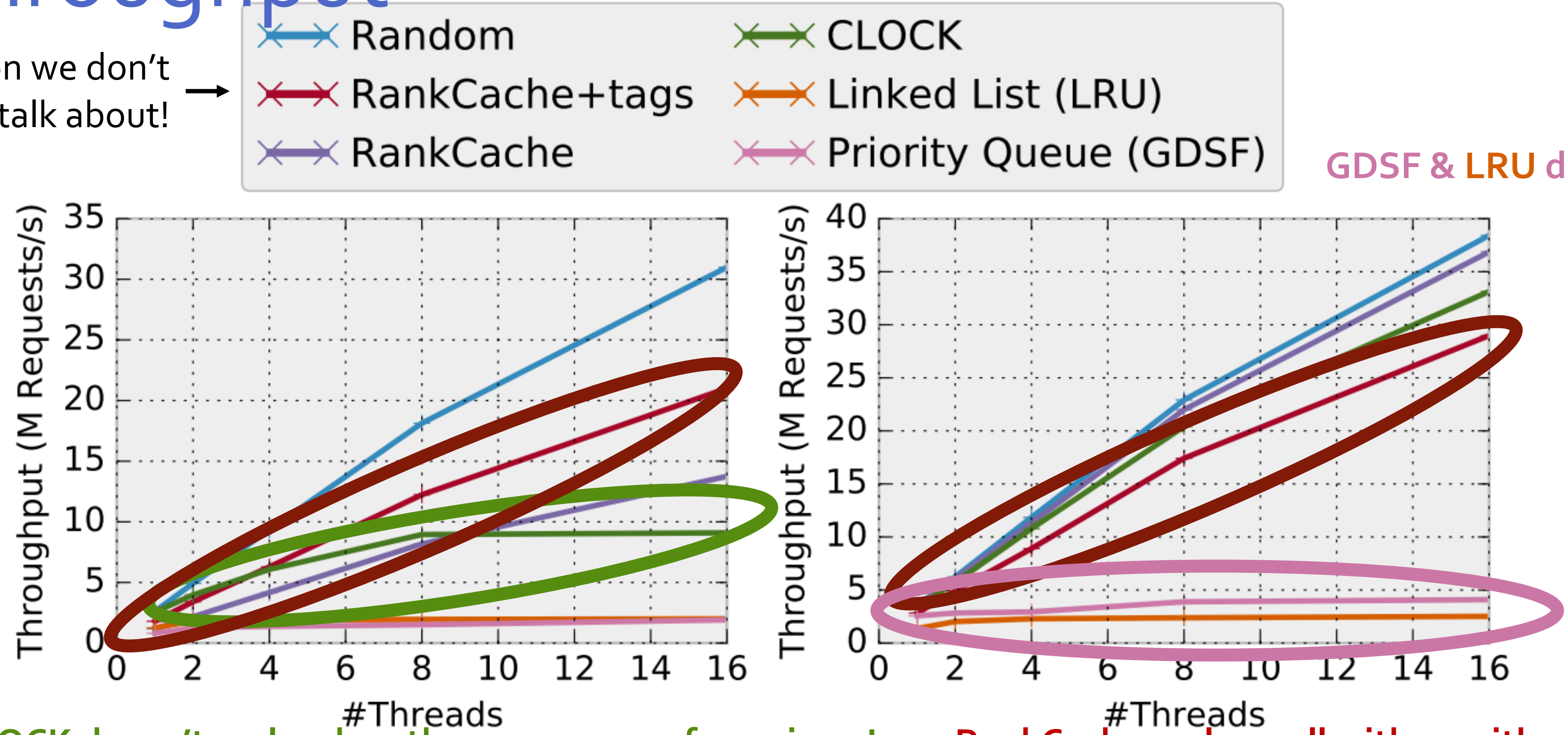
Memory management

- Many key-value caches use slab allocators (eg, memcached)
- Bounded fragmentation & fast
- …But no global eviction policy ➔ poor hit ratio
- Strategy: balance **victim hit density** across slab classes
  - Similar to Cliffhanger [Cidon, NSDI'16] and GD-Wheel [Li, EuroSys'15]
- Slab classes incur negligible impact on hit rate



Serial bottlenecks dominate → LHD best throughput

Optimization we don't have time to talk about! →



GDSF & LRU don't scale!

CLOCK doesn't scale when there are even a few misses!

RankCache scales well with or without misses!

(a) 90% Hit ratio.

(b) 100% Hit ratio.

Related Work

- Using conditional probabilities for eviction policies in CPU caches
  - EVA [Beckmann, HPCA '16, '17]
  - Fixed object sizes
  - Different ranking function

- Prior replacement policies
  - Key-value: Hyperbolic [Blankstein, ATC '17], Simulations [Waldspurger, ATC '17], AdaptSize [Berger, NSDI '17], Cliffhanger [Cidon, NSDI '16]…
  - Non key-value: ARC [Megiddo, FAST '03], SLRU [Karedla, Computer '94], LRU-K [O'Neil, Sigmod '93]…
  - Heuristic based
  - Require tuning or simulation

Future directions

- Dynamic latency / bandwidth optimization
  - Smoothly and dynamically switch between optimized hit ratio and byte-hit ratio

- Optimizing end-to-end response latency
  - App touches multiple objects per request
  - One such object evicted ➔ others should be evicted too

- Modeling cost, e.g., to maximize write endurance in FLASH / NVM
  - Predict which objects are worth writing to 2^{nd} tier storage from memory

THANK YOU!