Hyperbolic Caching: Flexible Caching for Web Applications

Aaron Blankstein
Princeton University
(now @ Blockstack Inc.)

Siddhartha Sen
Microsoft Research NY

Michael J. Freedman
Princeton University

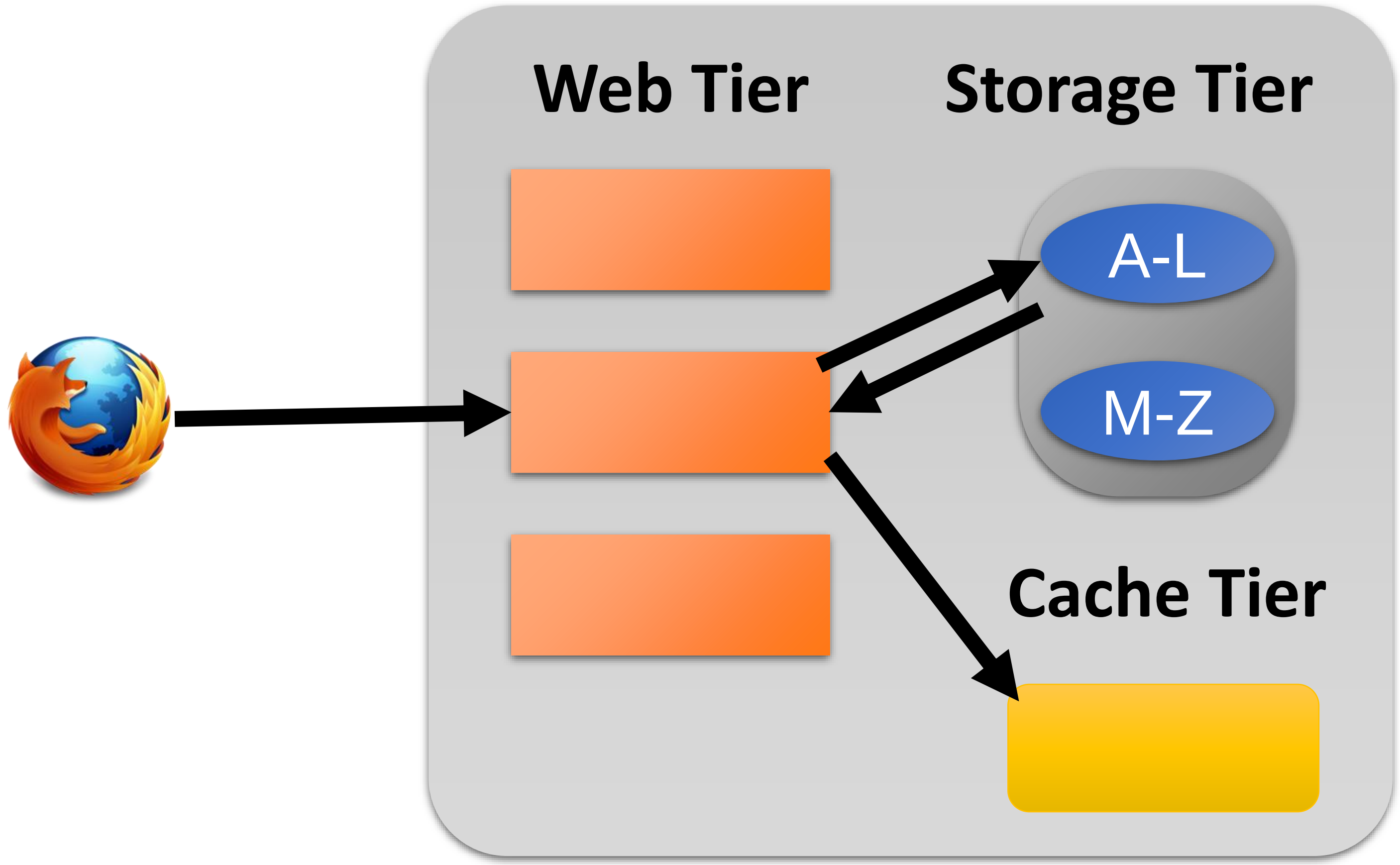
Modern Web Applications

- Ubiquitous, important, diverse

Users Expect Performance

- Diversity of app ecosystem makes this hard
- Improving web app performance is not trivial
- Application caches are aggressively deployed for this
    - But can hit rates be improved?

Application Caching on the Web



- Web-like Request Patterns

- Varying item costs
- Item Expirations
- Etc.

Cache Performance is About Eviction

- For long-tailed workloads, you CANNOT cache everything
- Hit rate (and miss rate) will depend on what you kick out
- Ideally – kick out things that are least likely to be requested

Tailoring Cache Eviction

- Web apps are different than disk or CPU caches:
  - Size and cost are important!
  - Request patterns are different

- Two goals of a tailored eviction strategy:
  - Tailor to web-specific request distributions
  - Tailor to the varying needs of different app settings

Traditional Caching Strategies Have Issues

- LRU and other recency based approaches:
  - Perform generally very well, but on stable, memoryless distributions, outperformed by frequency strategies
- LFU:
  - Problems with traditional implementation (evict item with fewest hits)
  - Punishes *new* items
  - *Old items* may survive even after dropping in importance

Many Variants to Improve These Strategies

- GreedyDual incorporates cost with recency
- *k*-LRU uses multiple LRU queues (ARC is a self-balancing approach)
- Some even model this as an optimization problem

Many Variants to Improve These Strategies

- GreedyDual incorporates cost with recency
- *k*-LRU uses multiple LRU queues (ARC is a self-balancing approach)
- Some even model this as an optimization problem

Problem: All limited by use of an eviction data structure!

Key Insight:

Decouple item priorities
from eviction data structures

But How to Evict? Use Random Sampling

- We can use *random sampling* for eviction
- Now, item priorities do not necessarily need to be tied to a particular data structure
- This opens up the design space for prioritization

Why Now?

- Systems such as Redis already use random sampling
  - Use for efficiency and simplicity of implementation
  - Approximates LRU
- Theoretical justification already exists (Psounis and Prabhakar)
- However, no one has proposed a strategy that leverages this flexibility

Hyperbolic Caching

- Flexible caching scheme
- Define *priority function* and do *lazy evaluation with sampling* to evict
- Focus on defining how important an object is, not data structures!

Hyperbolic Caching

- We define priority function

$$pr(i) = \frac{number\ of\ accesses}{time\ since\ i\ entered\ cache}$$

- We allow for many different variations on this priority scheme

Hyperbolic Cac

- We define priority fun

$$pr(i) = \frac{number\ of\ accesses}{time\ since\ i\ entered\ cache}$$

- We allow for many different variations on t

Frequency captures independent draws property of workloads

Addresses problems of LFU by measuring relative popularity

Implementing Hyperbolic Caching

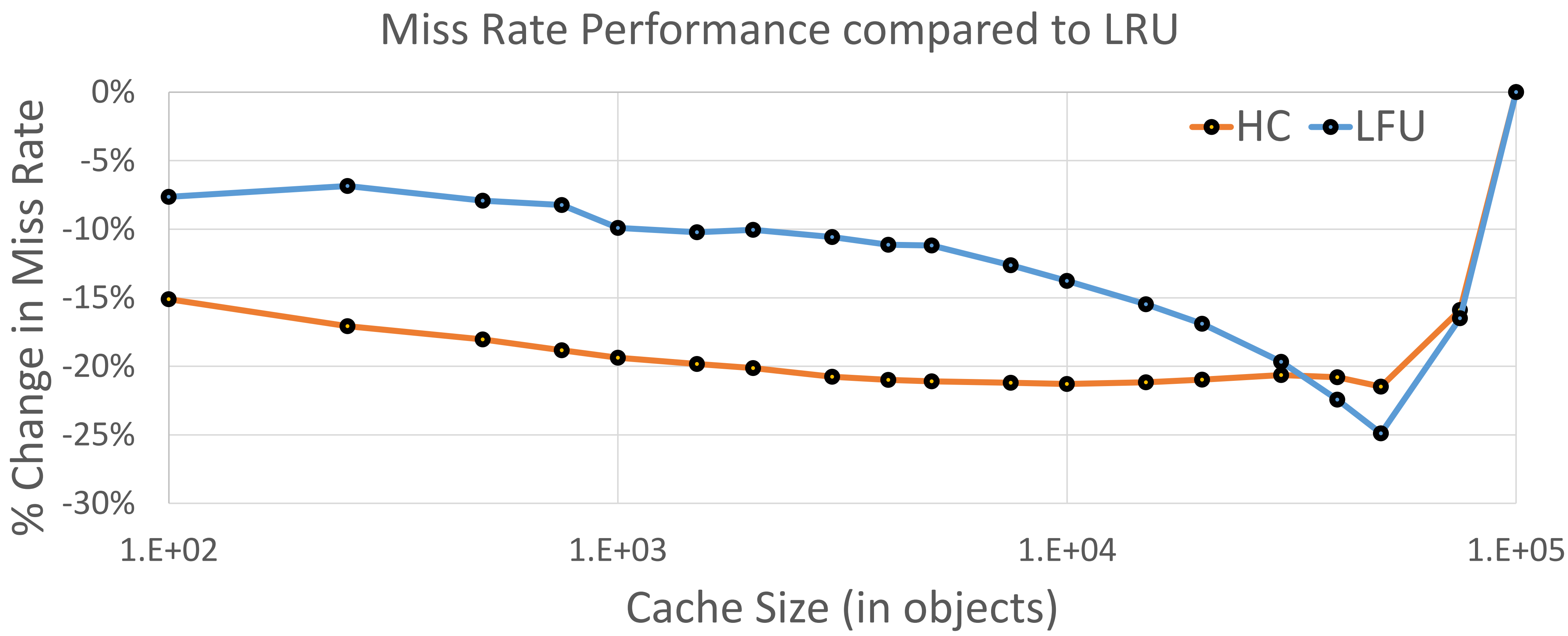
- Traditional eviction uses data structures for *ordering*
- Hyperbolic caching creates item re-orderings
- Example:

  Item requests: A A B C C

A and B reordered when *unrelated item* is requested!

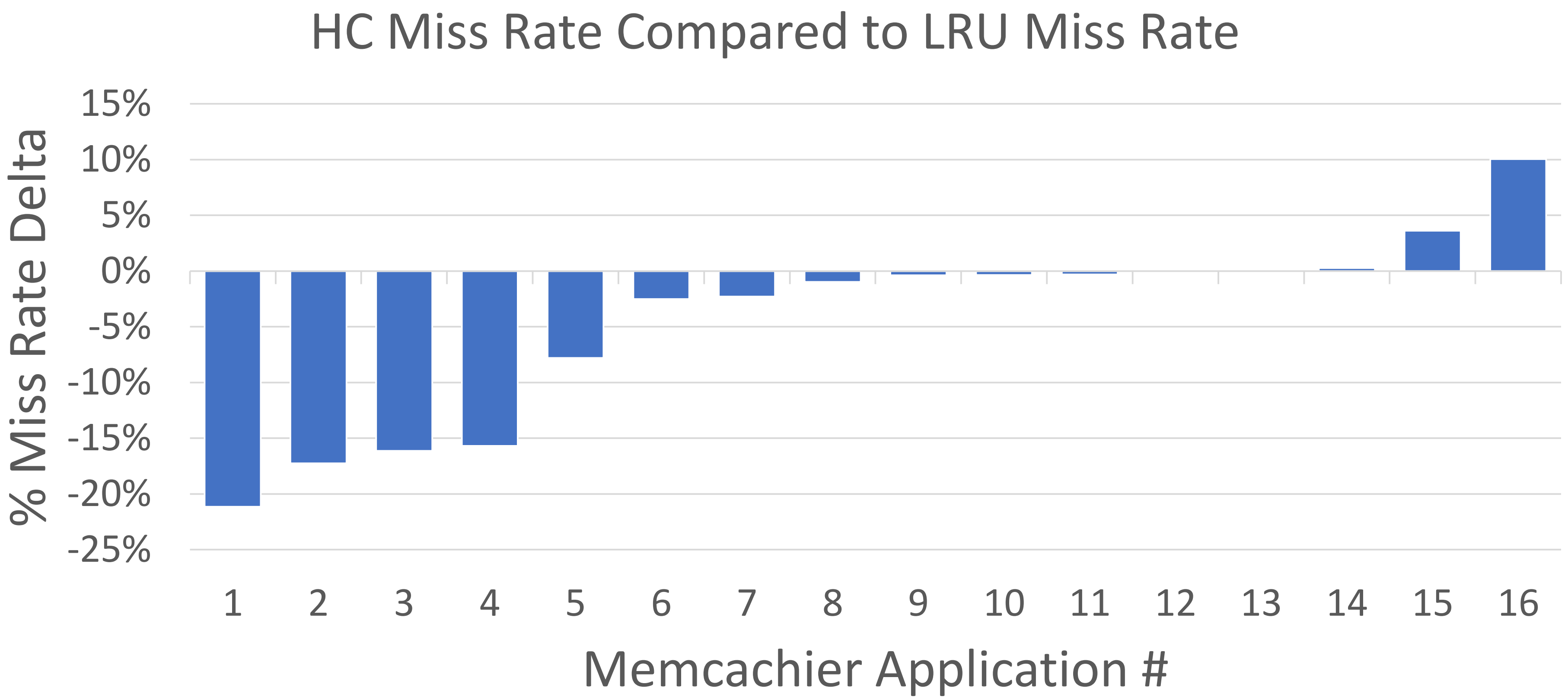
We can only do this because of random sampling!

Performance on Static Workload



- Items sampled from a static zipfian popularity distribution

Performance on Memcachier Traces



- Cache sizes chosen by app developers

Tailoring Caching for App Needs

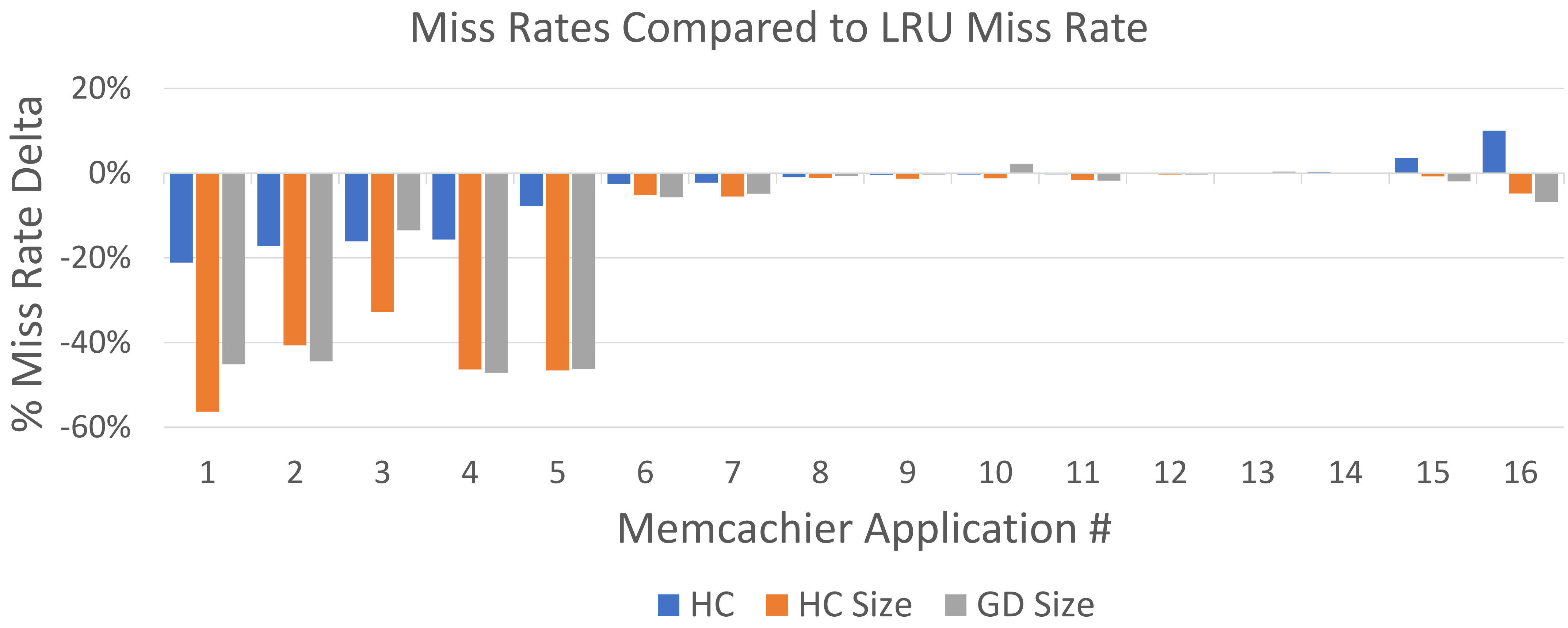
Tailoring Hyperbolic Caching

- Item costs
  $$pr'(i) = cost_i \cdot pr(i)$$
  - Items may impose different CPU or DB load on misses
  - Item sizes affect per-item hit rate
- Expiration times
  $$pr'(i) = (1 - e^{\alpha \cdot (time\ till\ epires)}) \cdot pr(i)$$
  - Apps can give expirations to prevent staleness
- Item classes
  $$pr'(i) = cost(group_i) \cdot pr(i)$$
  - Items may have related costs, and should have grouped costs

Cost-Aware Caching: State of the Art

- GreedyDual is well-known approach for incorporating cost
- However, implementation is not trivial
  - LRU->GD requires changing the cache’s data structures
  - HC -> HC+Cost just adds metadata and redefines priority function
- Furthermore, GD suffers on web workloads, because it is a recency based approach

Cost-Aware Perf. on Memcachier Traces



Cost Classes

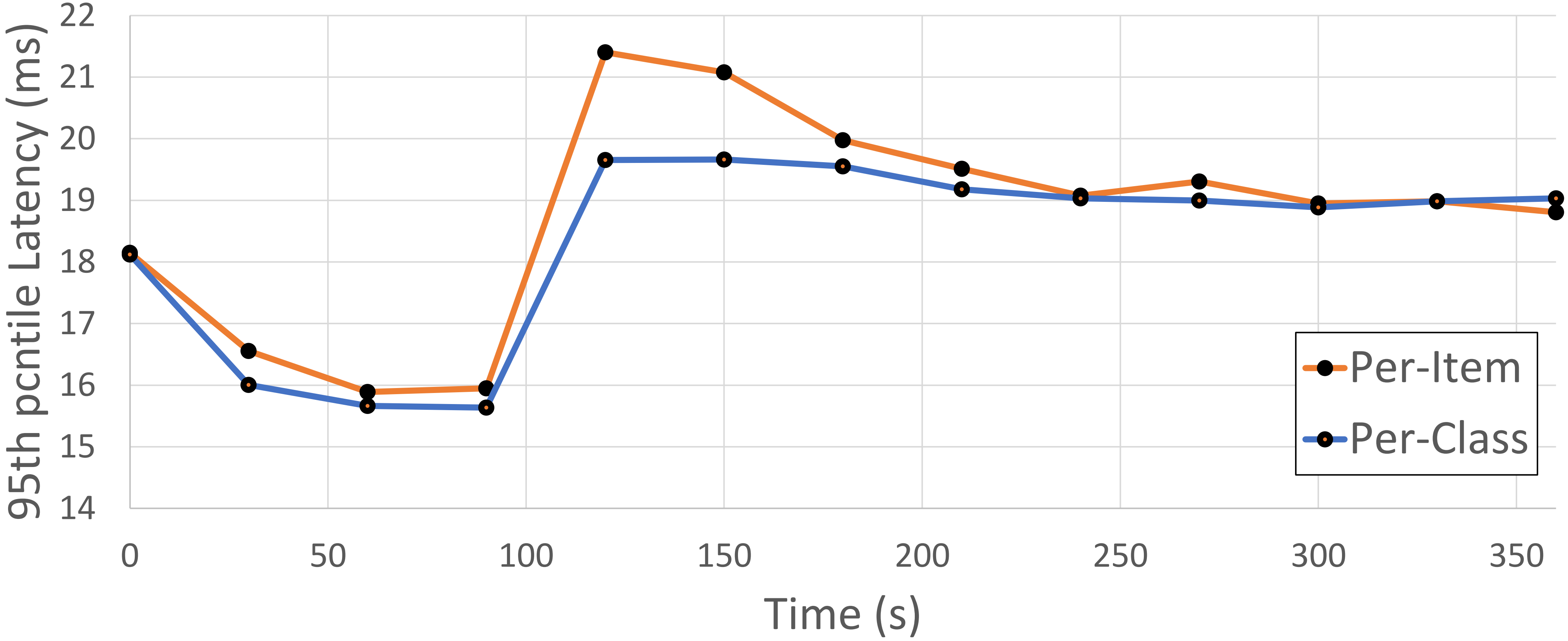
- Measure moving average of item costs over the class

$$pr'(i) = cost(group_i) \cdot pr(i)$$

- Cost of class can be updated while item A *in cache*
- Updating whole class very easy in our scheme

- Example use cases:
  - Class of items shares the same backend and related load

Dealing with Backend Load



- Items are requested from two different backends
- At time *t=120,* one server is stressed

Hyperbolic Caching Related Work

- Recent Application Cache Eviction Work
  - RIPQ – implementing size-awareness on flash
  - GDWheel – fast implementation of GD
  - CliffScaler – improving the LRU approx. of Memcached

- Web Proxy Caching
  - Many different projects demonstrating performance benefits of GD
  - Hyperbolic Caching’s prioritization outperforms these on our workloads

Conclusion

- Focusing on prioritizing items, hyperbolic caching improves caching performance on web-like workloads

- The scheme allows for a multitude of easily constructed variants

- We demonstrate performance as good as competitive baselines, and in many cases much better

- Fork us! Our Redis prototype and simulation code are at:

  github.com/kantai/hyperbolic-caching