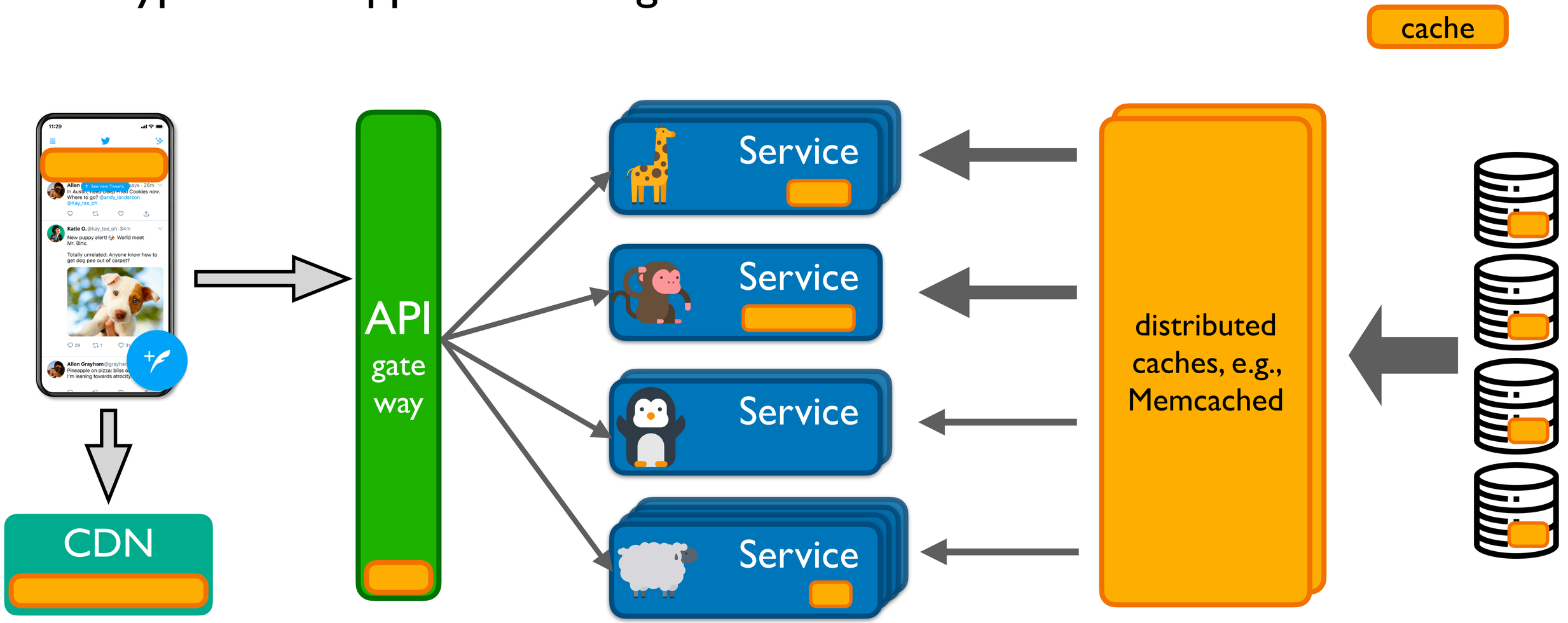
Designing Efficient and Scalable Key-value Cache Management Systems

Juncheng Yang

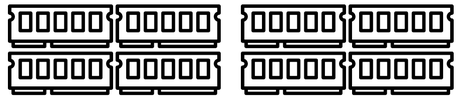
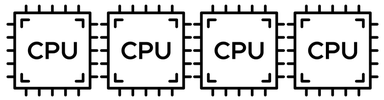
advisor: Rashmi Vinayak

Software caches are everywhere

A typical web application using microservice architecture



Cache performance

- Cache metrics
  - efficiency (miss ratio)
  - throughput and scalability (requests / sec)
- Critical to improve cache efficiency and scalability
  - higher efficiency => less DRAM 
  - better scalability => fewer servers, less load imbalance 

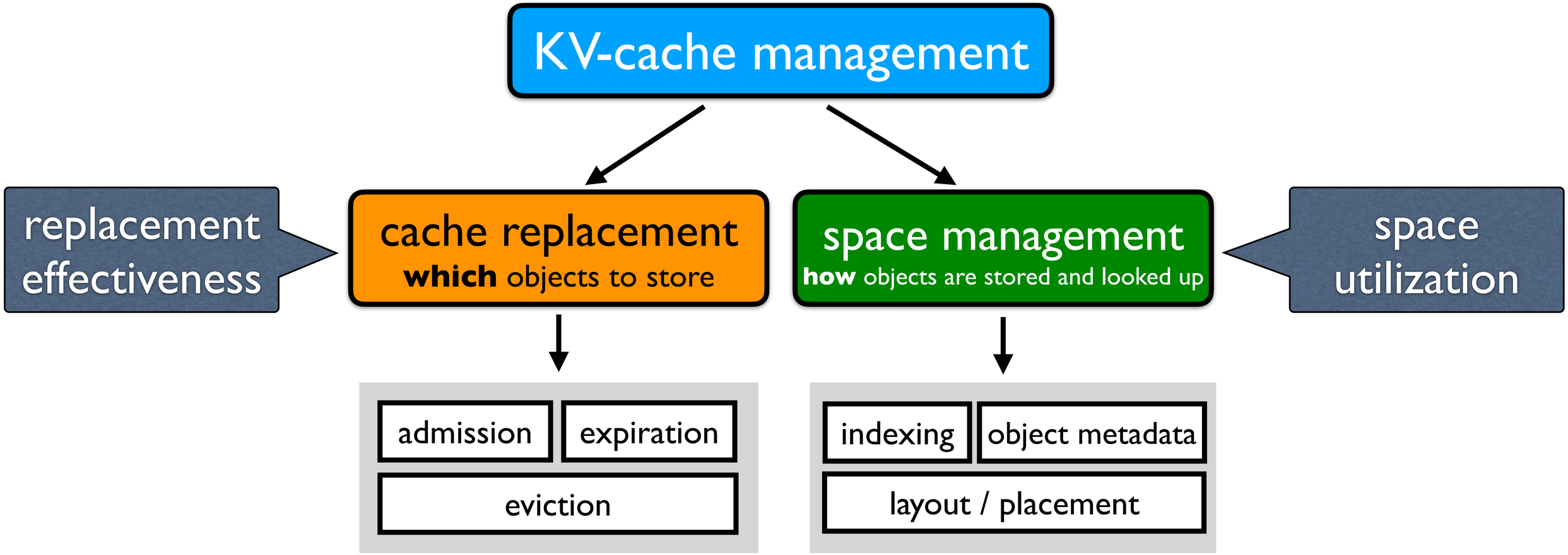
  - => more sustainable

Cache performance

- Decades of research on improving **efficiency**
  - mostly focus on the eviction algorithm
- Increasing research on cache **scalability**
  - increasing #cores per CPU
    - AMD EPYC 9654 has 96c/192t
    - requirement in production systems
      - RocksDB added Clock Cache in 2022
      - Apache Trafficserver explores better lock design and lockless DRAM cache in 2023

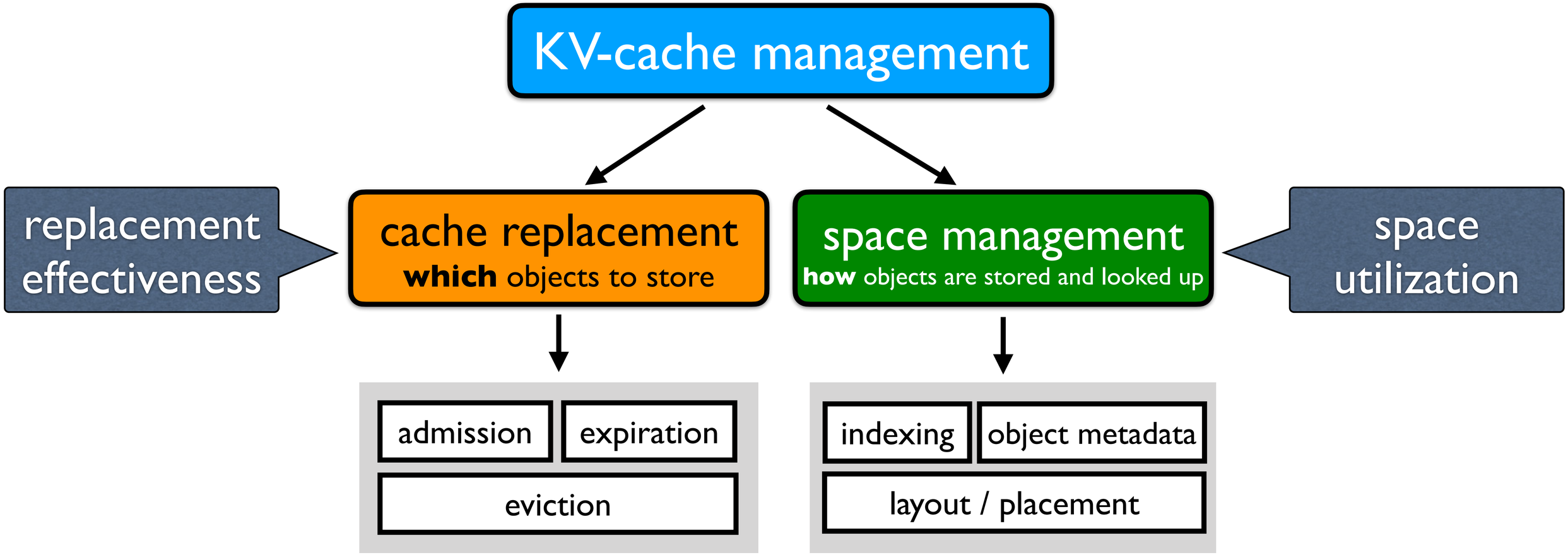
This thesis explores different approaches to improving the **efficiency** and **scalability** of a key-value cache

Key-value cache management system



Cache management is no longer only about eviction

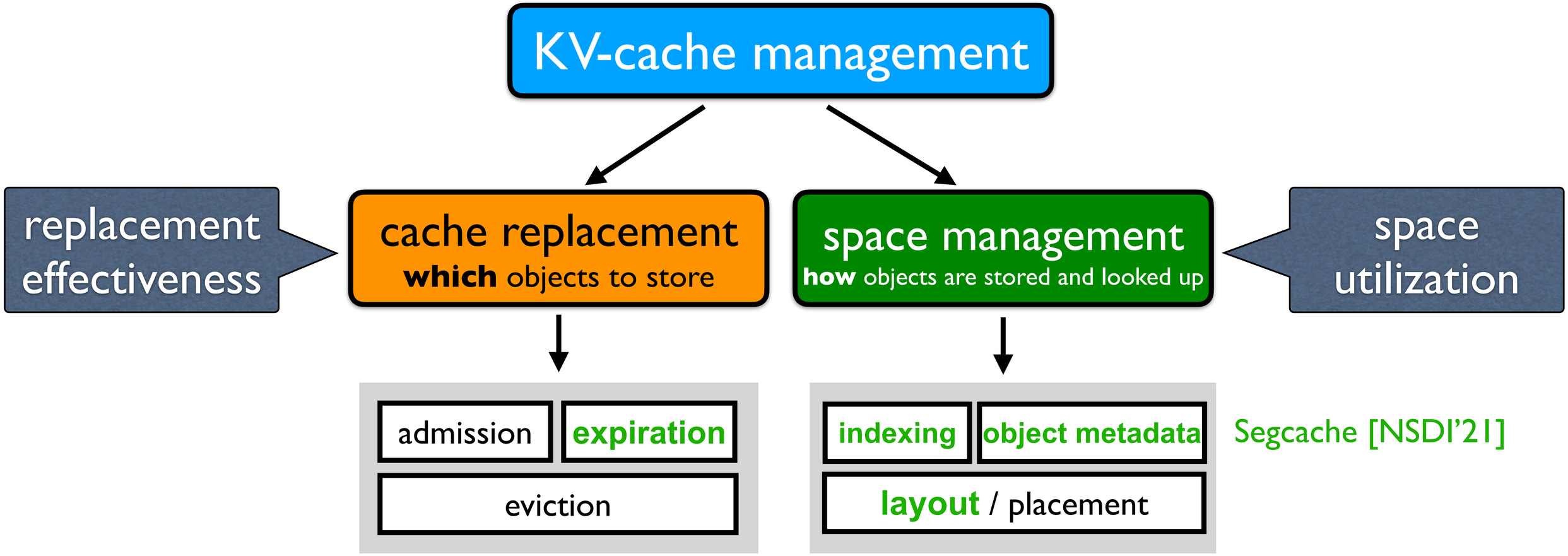
Agenda



Workload analysis [OSDI'20]

Cache management is no longer only about eviction

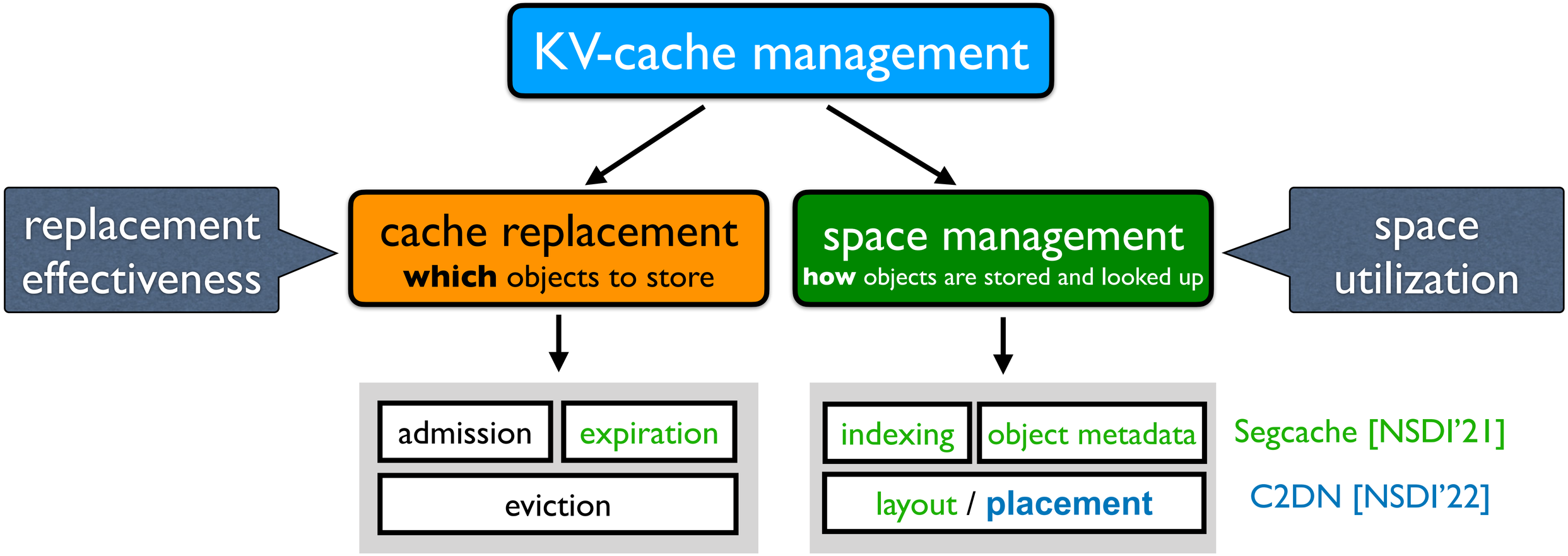
Agenda



Workload analysis [OSDI'20]

Cache management is no longer only about eviction

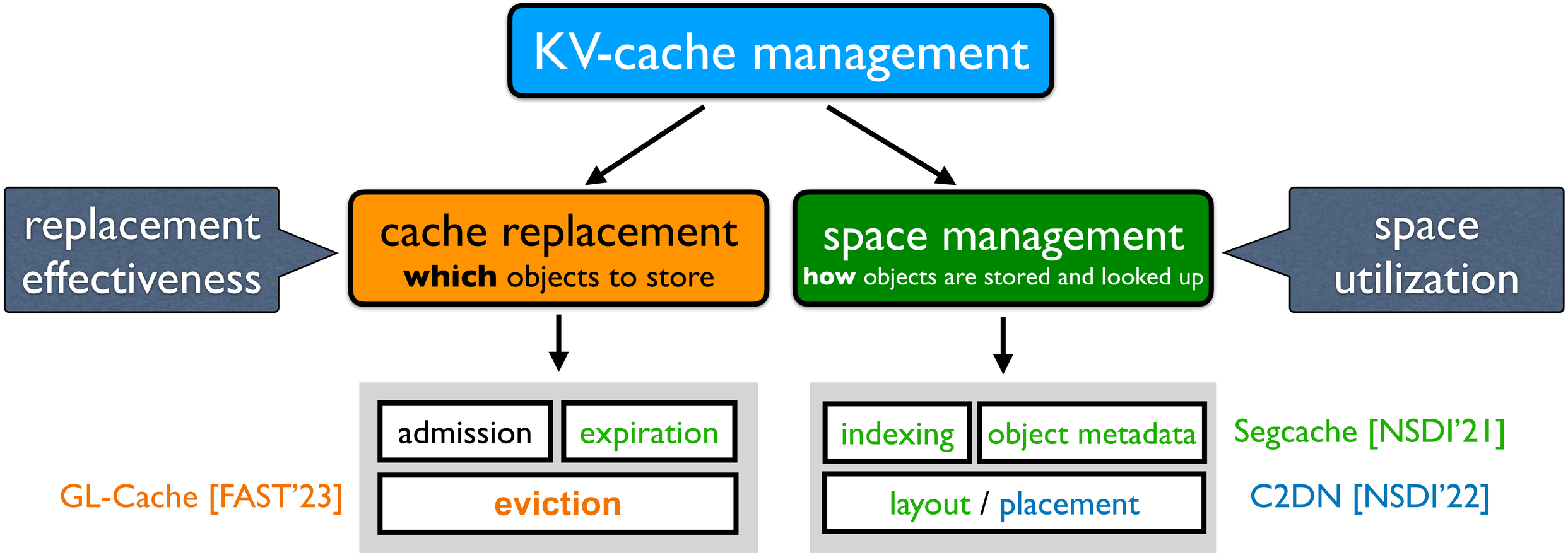
Agenda



Workload analysis [OSDI'20]

Cache management is no longer only about eviction

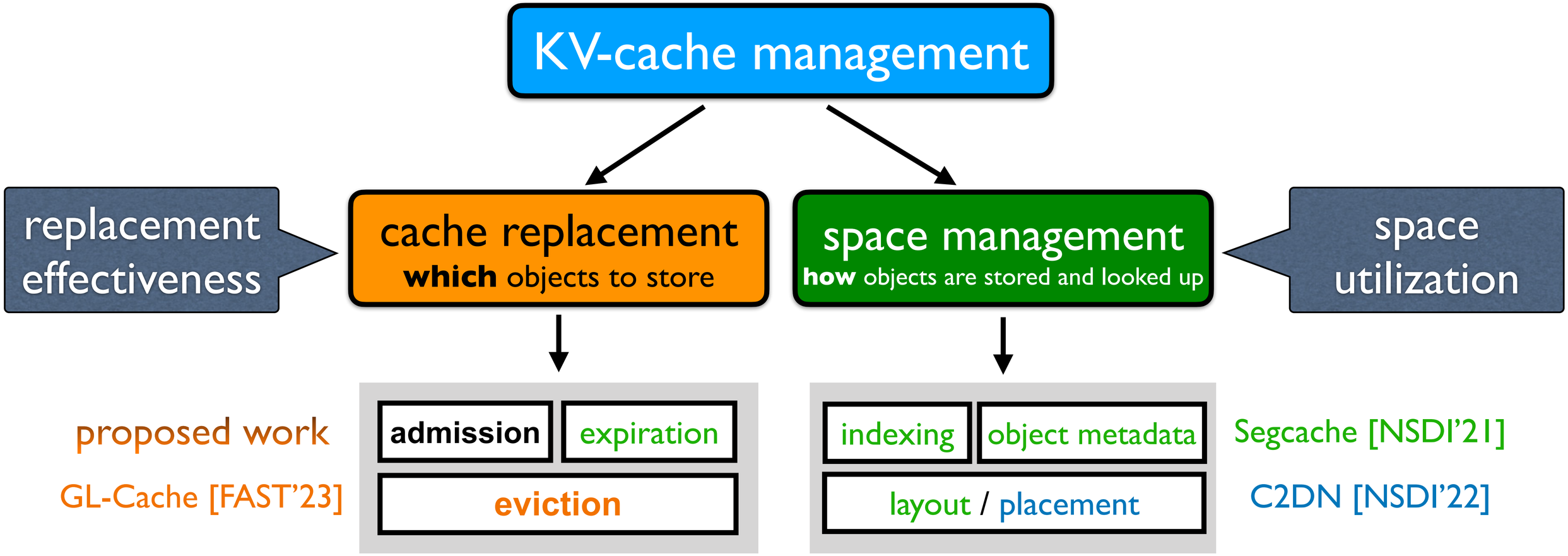
Agenda



Workload analysis [OSDI'20]

Cache management is no longer only about eviction

Agenda



Workload analysis [OSDI'20]

Cache management is no longer only about eviction

Part I. How are key-value caches used in production?

Object size distribution

Time-to-live (TTL) and working set

Cache use cases

Types of operations

Size distribution evolution

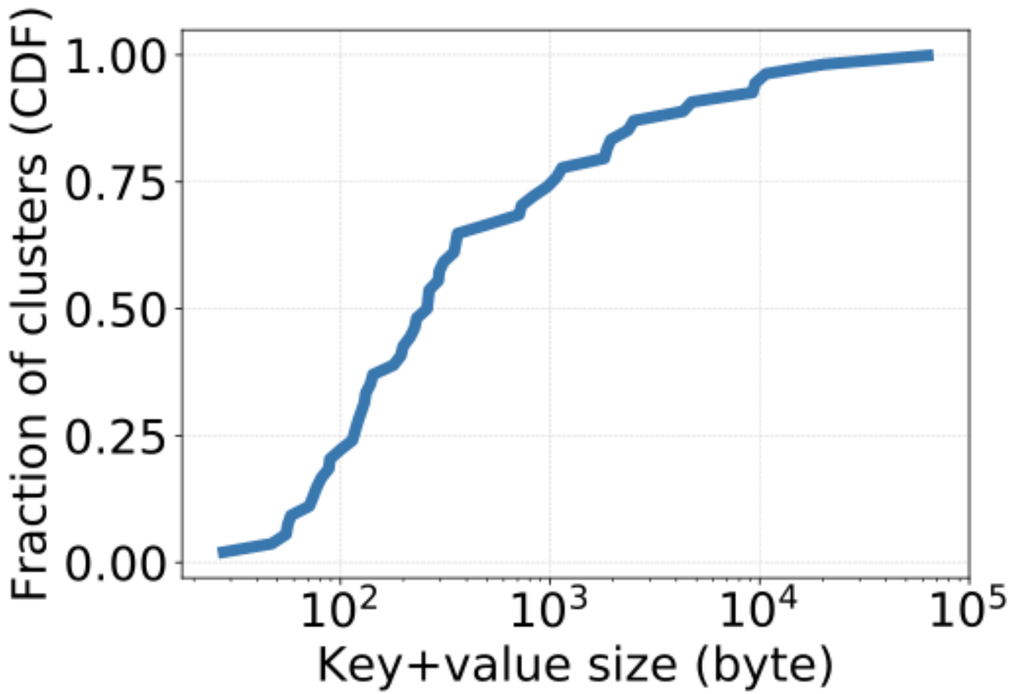
Object popularity distribution

In-memory key-value caches at Twitter

- Large-scale deployment
  - 100s cache clusters
  - 1s billion QPS
  - 100s TB DRAM
  - 100,000s CPU cores
- Single tenant, single layer
  - Container-based deployment
- Week-long unsampled traces from one instance of each cache cluster
  - 700 billion requests, 80 TB in size
  - Traces are open sourced: the largest public cache dataset so far

Object size

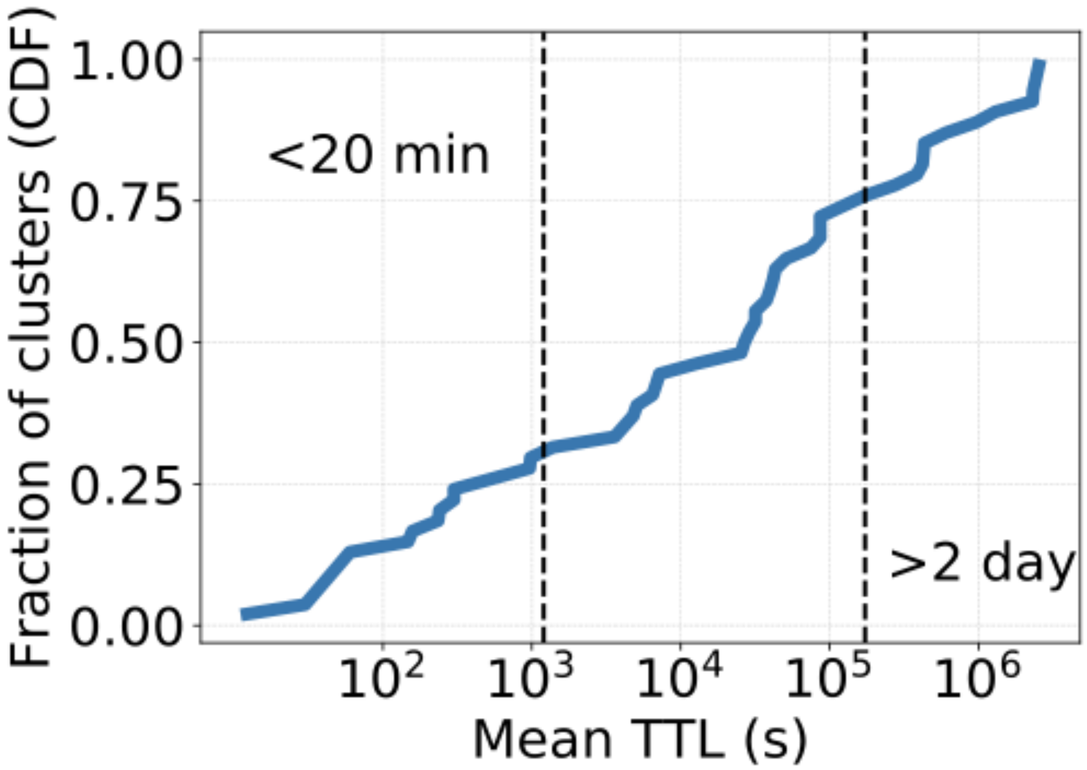
- Object sizes are small
  - 24% clusters: < 100 bytes
  - Median: 230 bytes
- Overhead of per-object metadata
  - Memcached uses 56 bytes per-obj metadata
  - Research systems often add more metadata



TTL use case and usage

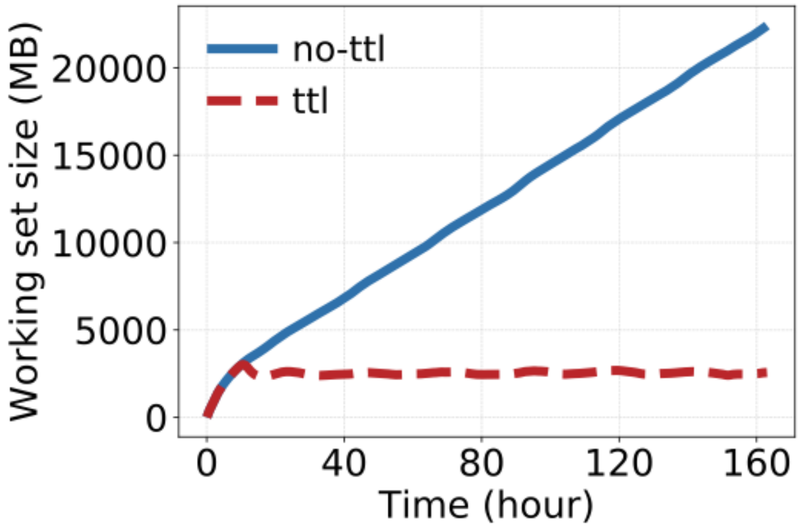
- Time-to-live (TTL)
  - Set during object write
  - Expired objects cannot be used
- TTL use case
  - Reduce stale data
  - Periodic refresh (e.g., ML predictions)
  - Implicit deletions (e.g., limiters, GDPR)
  - Signal objects' lifetimes

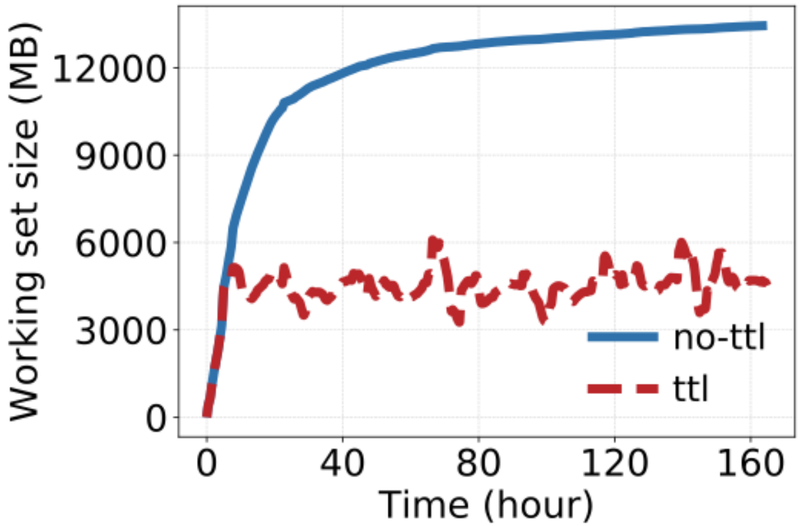
Short TTLs are widely used in production



TTL leads to a bounded working set size

There is no need for a huge cache size if expired objects can be removed in time





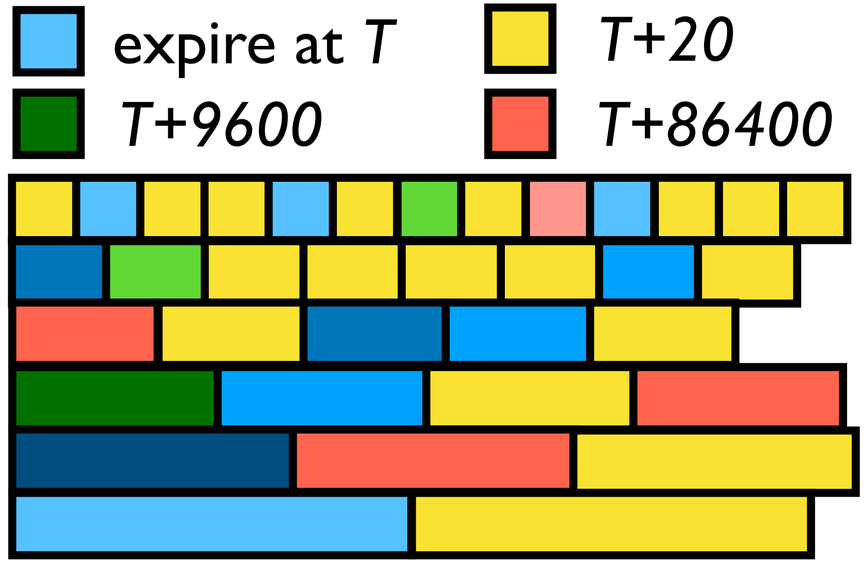
Timely removal of expired objects is critical for efficiency

- expiration: remove objects that **cannot** be used in the future
- eviction: remove objects that **could** potentially be used in the future

Existing approaches are not efficient or not sufficient

Efficient: small overhead
Sufficient: can remove expired objects in time

Category	Technique	Efficient	Sufficient
Lazy expiration	Delete upon re-access	✓	✗
	Check LRU tail	✓	✗
Proactive expiration	Scanning	✗	✓
	Sampling	✗	✗
	Transient object pool	✓	✗



How can I find expired objects

More in the thesis

- **Production statistics**
  - small miss ratio and small variations
  - request spikes are not always caused by hot keys
  - operations
- **Cache use cases**
- **Object popularity**
  - mostly Zipfian with large skewness
- **Object size**
  - key size and value size relationship
  - object size distribution is not static
- **More analysis of TTLs**
- **Eviction algorithms**
  - highly workload dependent
  - FIFO can be as good as LRU

Impact of the workload analysis

- Traces used by many works
- Observations and analysis inspired and helped many new designs
  - object size: DiffKV [ATC'21], REMIX [FAST'21], Bedrock [Usenix Security'22], Kangaroo [SOSP'21], Backdraft [NSDI'22]…
  - object popularity: Pegasus [OSDI'20], NAP [OSDI'21], Aurogon [FAST'22], Owl [OSDI'22], FrozenHot [Eurosys'23]…
  - operation ratio: Parallax [SOCC'21], SKYROS [SOSP'21], LogECMem [SC'21], FUSEE [FAST'23]…
  - wide use of TTLs, working set: Nyxcache [FAST'22], BigKV [Eurosys'23]…
  - others: NHC [FAST'21] (miss ratio), DataFreshness [IOT'23]…
- Used as reading materials for classes and reading groups

Part II. Segment-structured cache (Segcache)

An efficient and scalable cache storage design

Motivations: workload and hardware trends

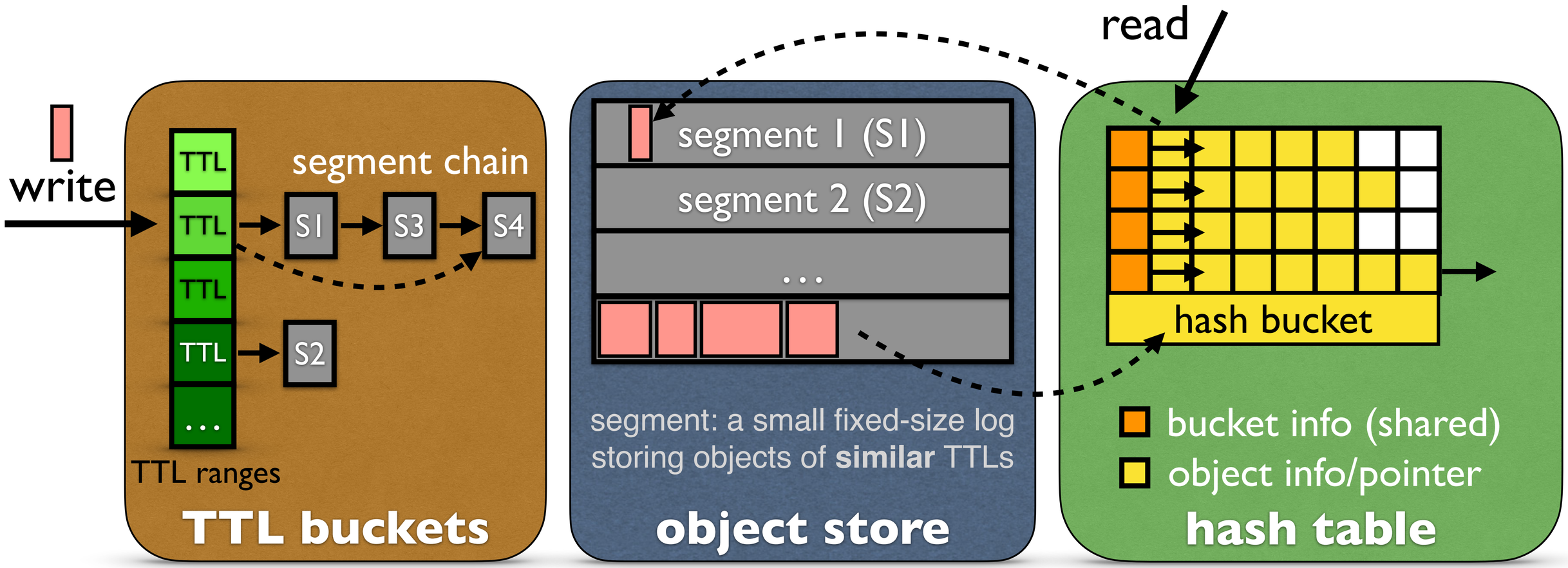
Workload trends

- Object sizes are **small**: *metadata overhead*
- Object sizes are **non-uniform**: *fragmentation*
- TTLs are widely used: *space wasted on expired objects*

Hardware trends

- An increasing number of cores per CPU: *lock contention*
- Poor random write performance on PMEM and flash: *low throughput*

Segcache design



Key idea

KV-cache ≠ KV-store + eviction

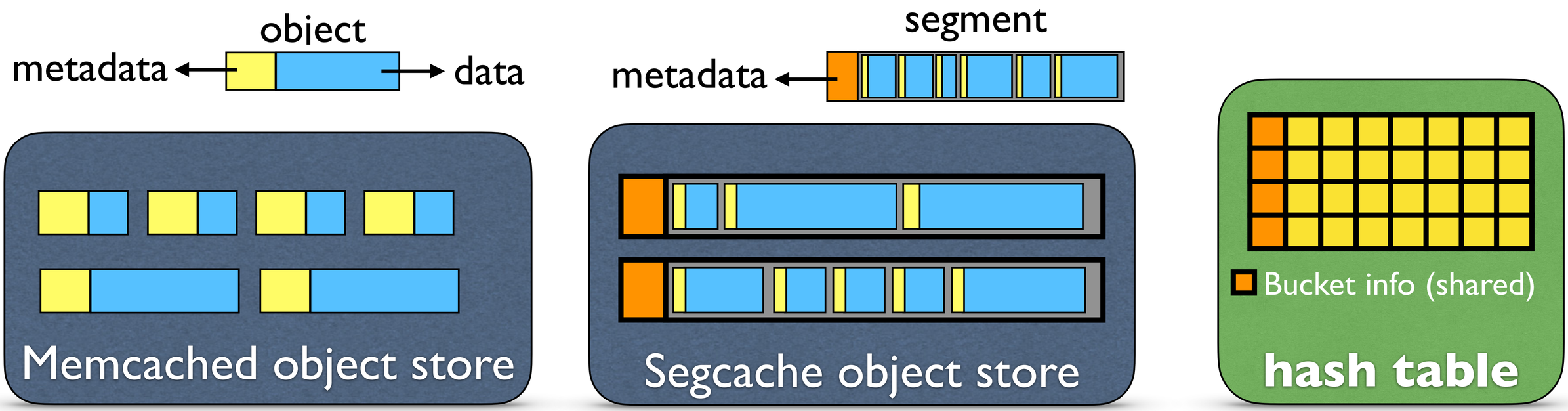
- Different design requirements
  - efficiency + scalability vs durability + strong consistency
- Different workloads
  - store: traffic filtered by the cache
  - cache: shorter TTLs + more skewed workloads

make the right trade-off for caching

Design 1

Maximize metadata sharing via approximation

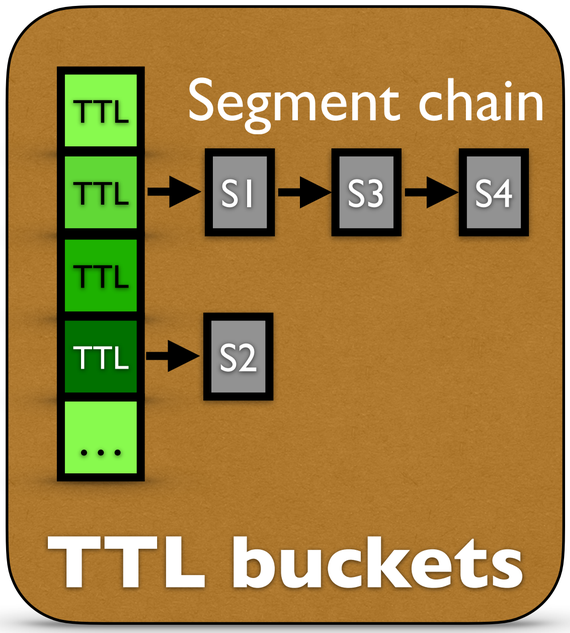
- **Insight**: many metadata can be approximate and/or shared
- **Design**: share metadata between objects in the same segment / hash bucket
- **Trade-off**: metadata precision vs space saving

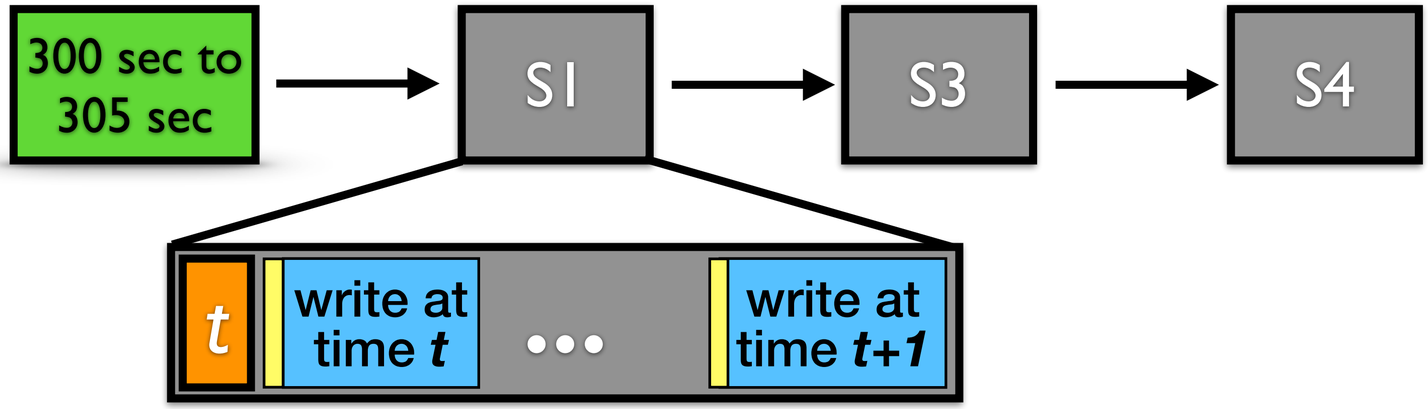


Design 2

Prioritize expiration over eviction

- **Insight**: expired objects cannot be used, but objects to be evicted may be used
- **Design**: approximate TTL indexing
- **Trade-off**: expire objects slightly early vs evicting an object



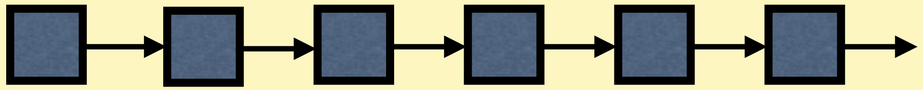


Design 3

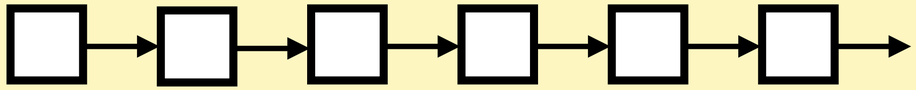
Macro management

- **Insight**: objects do not need to be promoted in the LRU queue before eviction
- **Design**: manage segments (groups of objects), not objects
- **Trade-off**: may increase miss ratio in some workloads vs high throughput/scalability

Object LRU chain



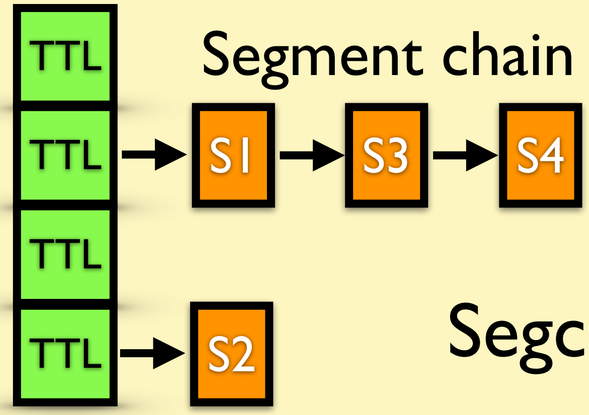
Free chunk chain



Every operation needs locking

Memcached

Expiration/eviction (locking) on the segment level



Segcache

In the thesis

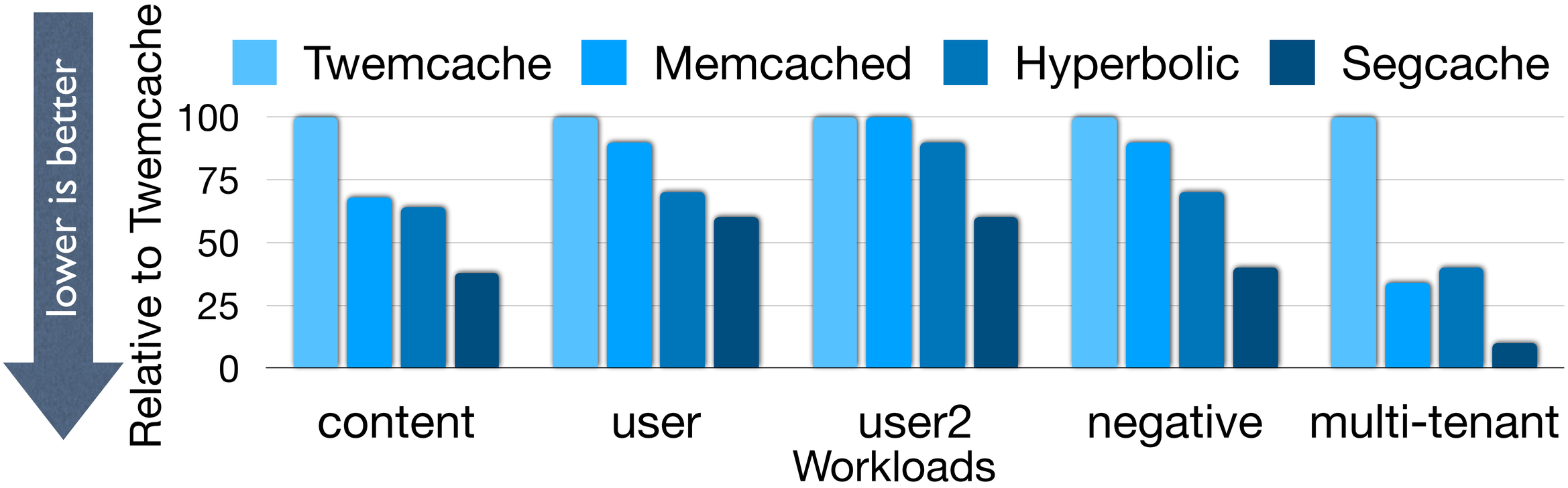
- Segment homogeneity
- Merge-based eviction
  - Approximate and smoothed frequency counter
    - ✓ Low overhead
    - ✓ Burst-resistant
    - ✓ Scan-resistant
    - ✓ Eviction-friendly

Evaluation setup

- Implemented as part of Pelikan
- Five systems: Twemcache, Memcached, LHD, Hyperbolic, Segcache
- Five week-long production traces
- Storage only (no RPC)

Evaluation

How much memory needed to achieve production miss ratios?

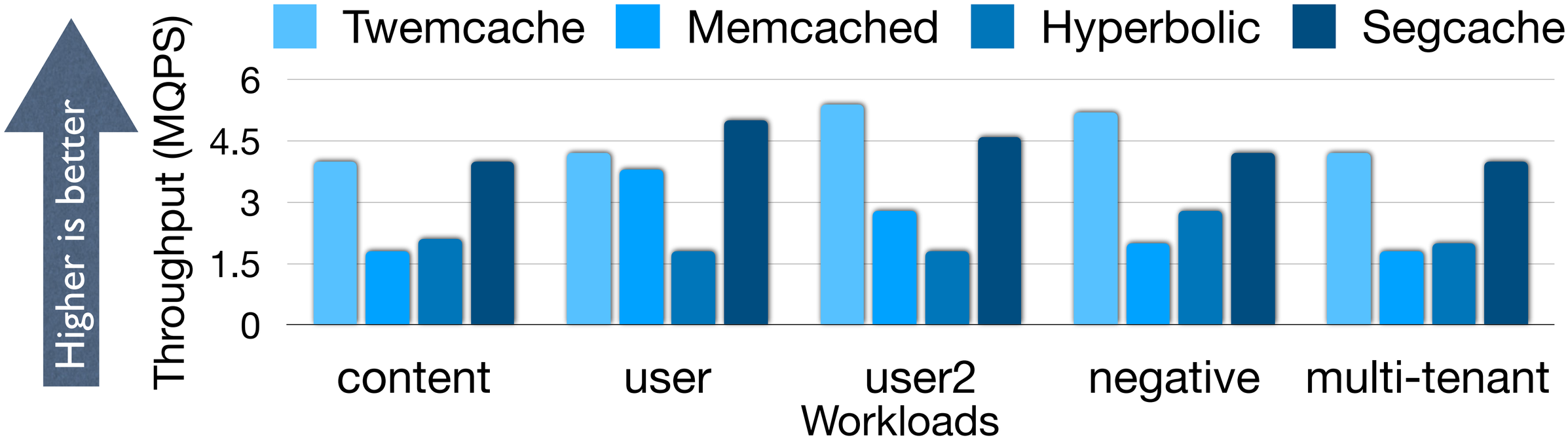


40-90% compared to Twemcache (60% on the largest cluster)

22-60% compared to state-of-the-art

Evaluation

Single-thread throughput performance

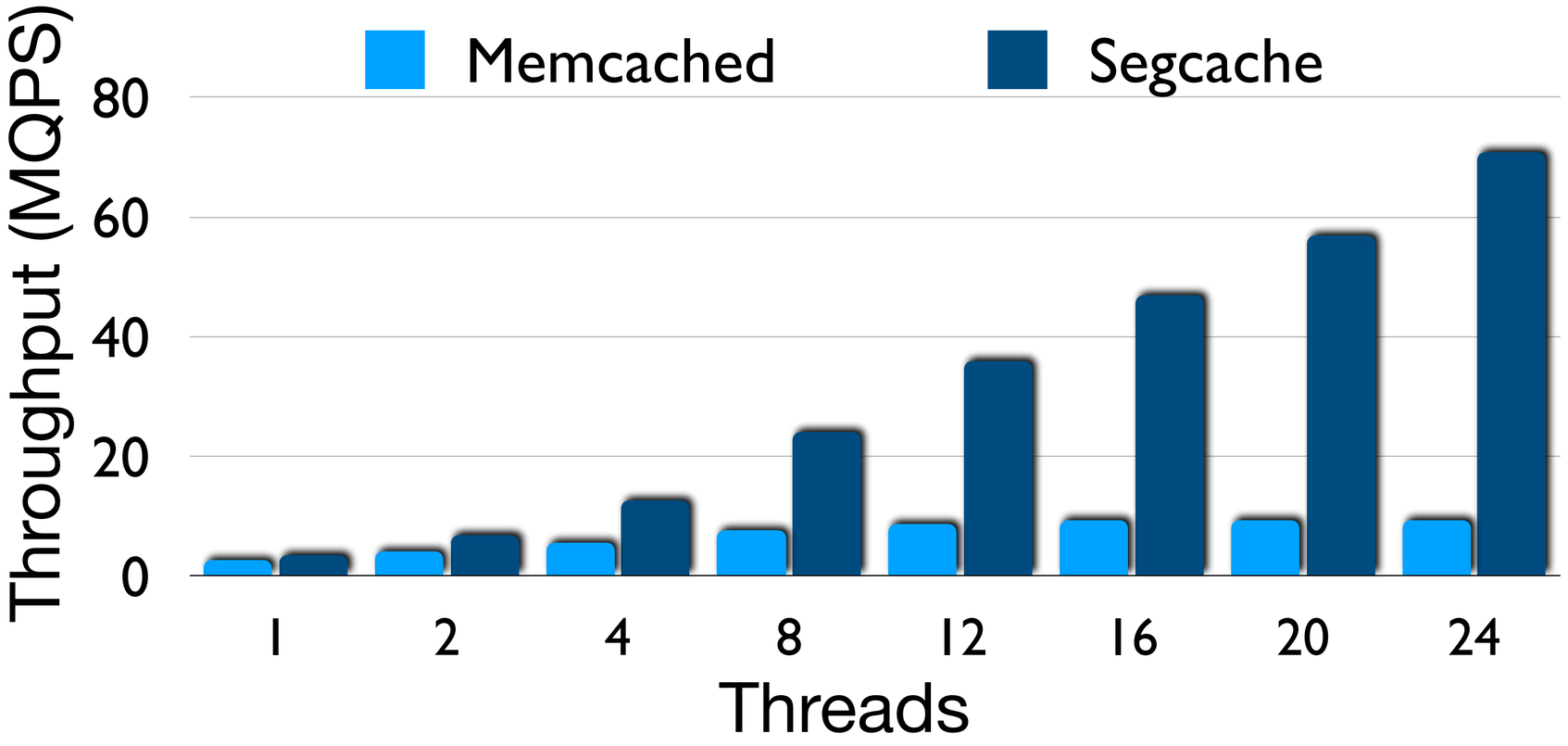


Close to Twemcache

Up to 40% higher than Memcached

Evaluation

Multi-thread throughput performance



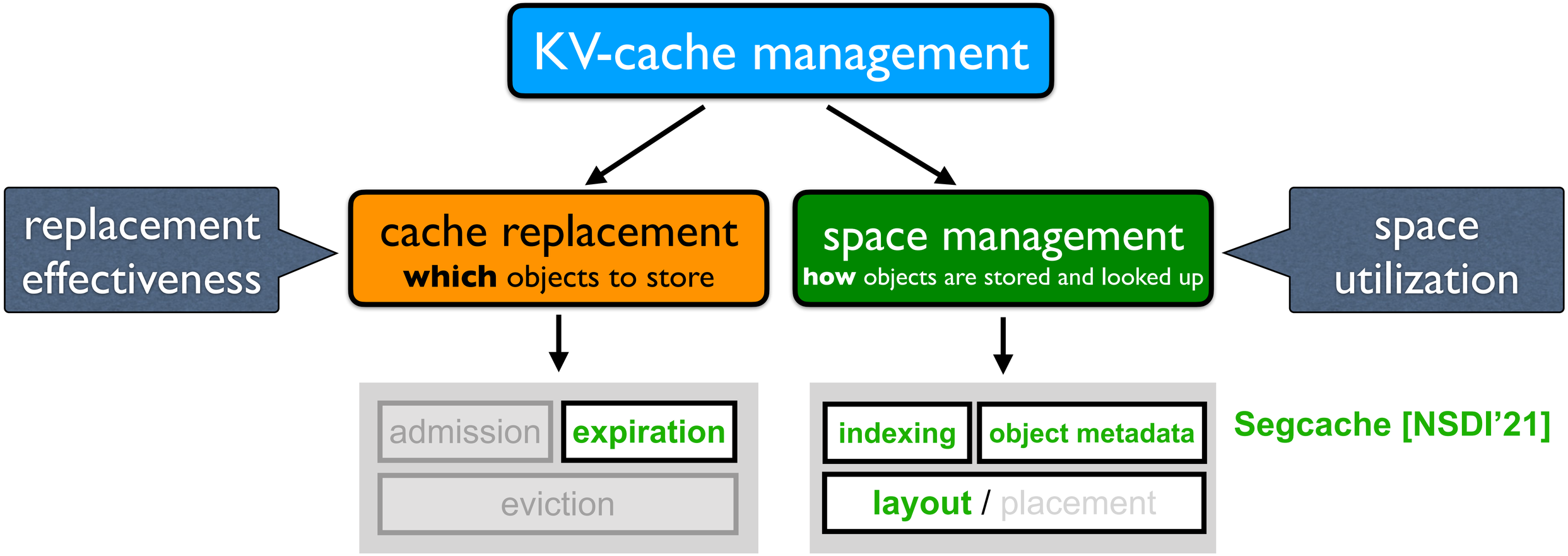
8x improvement with 24 threads

Summary

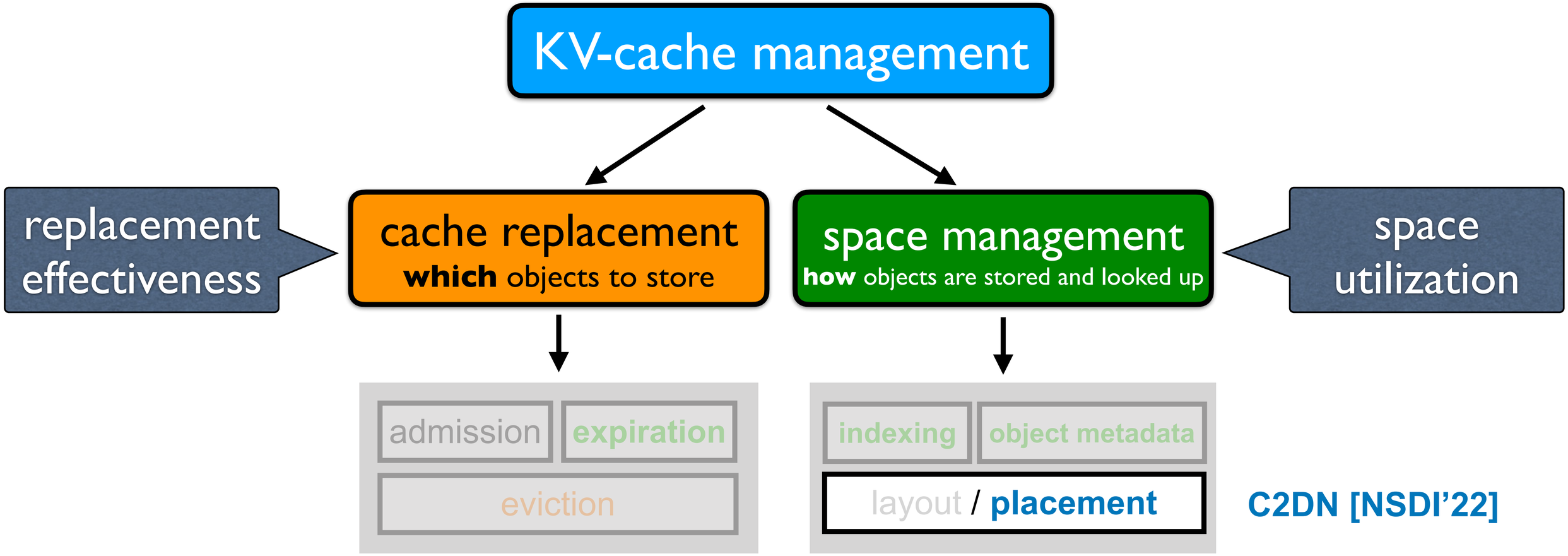
- Segcache: an approximate-TTL-indexed segment-structured cache
  - efficient TTL expiration
  - tiny object metadata (5 bytes)
  - almost no fragmentation
  - close-to-linear scalability

- Segcache impact
  - NSDI'21 community (best paper) award
  - in production at Twitter and Momento

Key-value cache management system



Key-value cache management system



Part III. Coded Content Delivery Network (C2DN)

How to design an efficient fault tolerant CDN

Motivation: unavailabilities at the edge are common

A month-long trace of 2190 clusters

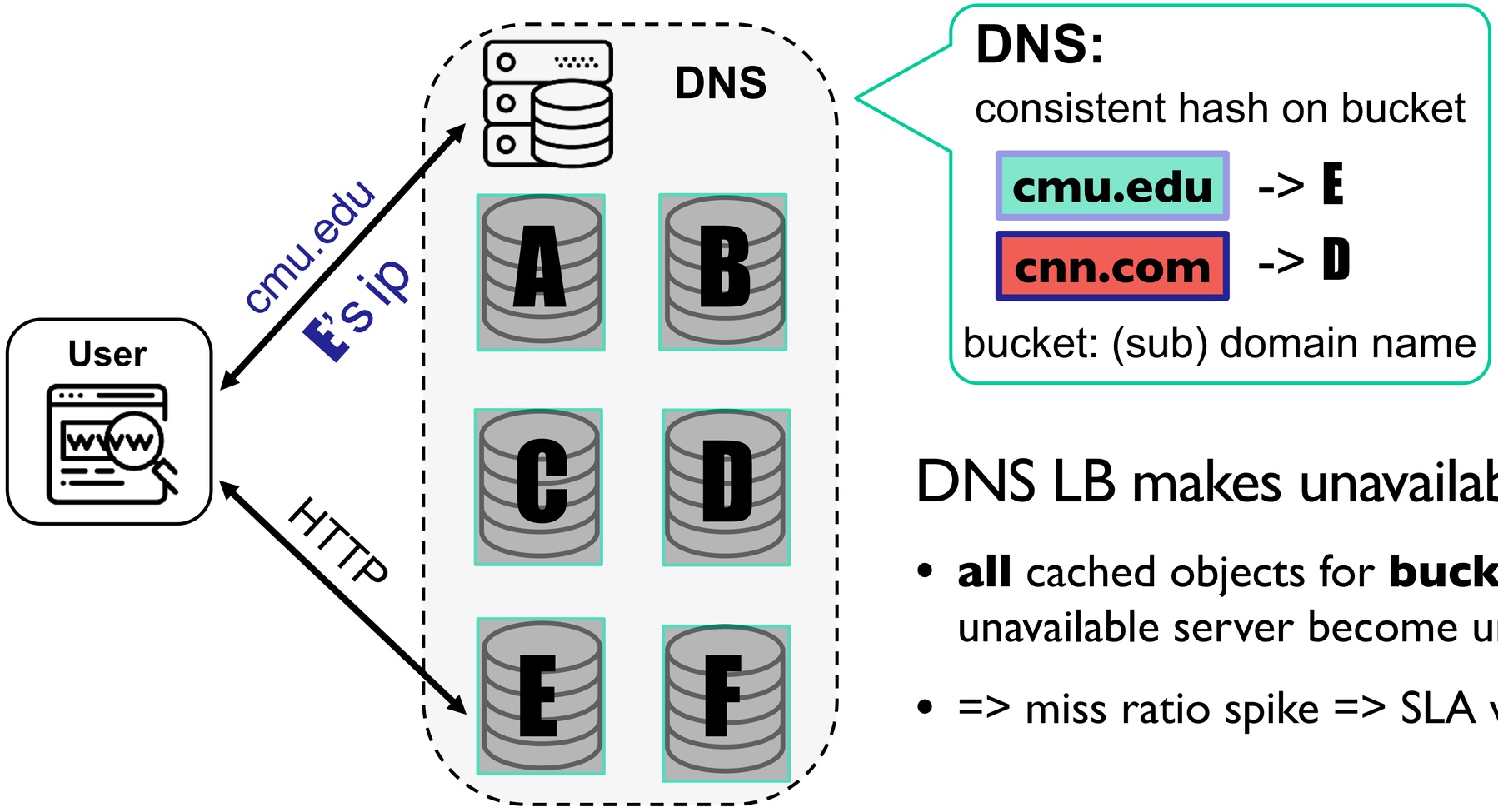
All clusters: unavailability in 45.2% of observations

10-server clusters: unavailability in 30.5% of observations

Unavailabilities are more common than in hyper-scale data centers

Reasons: server overload, hardware failure

Bucket-based routing and coarse load balancing (LB)



DNS:

consistent hash on bucket

cmu.edu -> **E**

cnn.com -> **D**

bucket: (sub) domain name

DNS LB makes unavailability worse

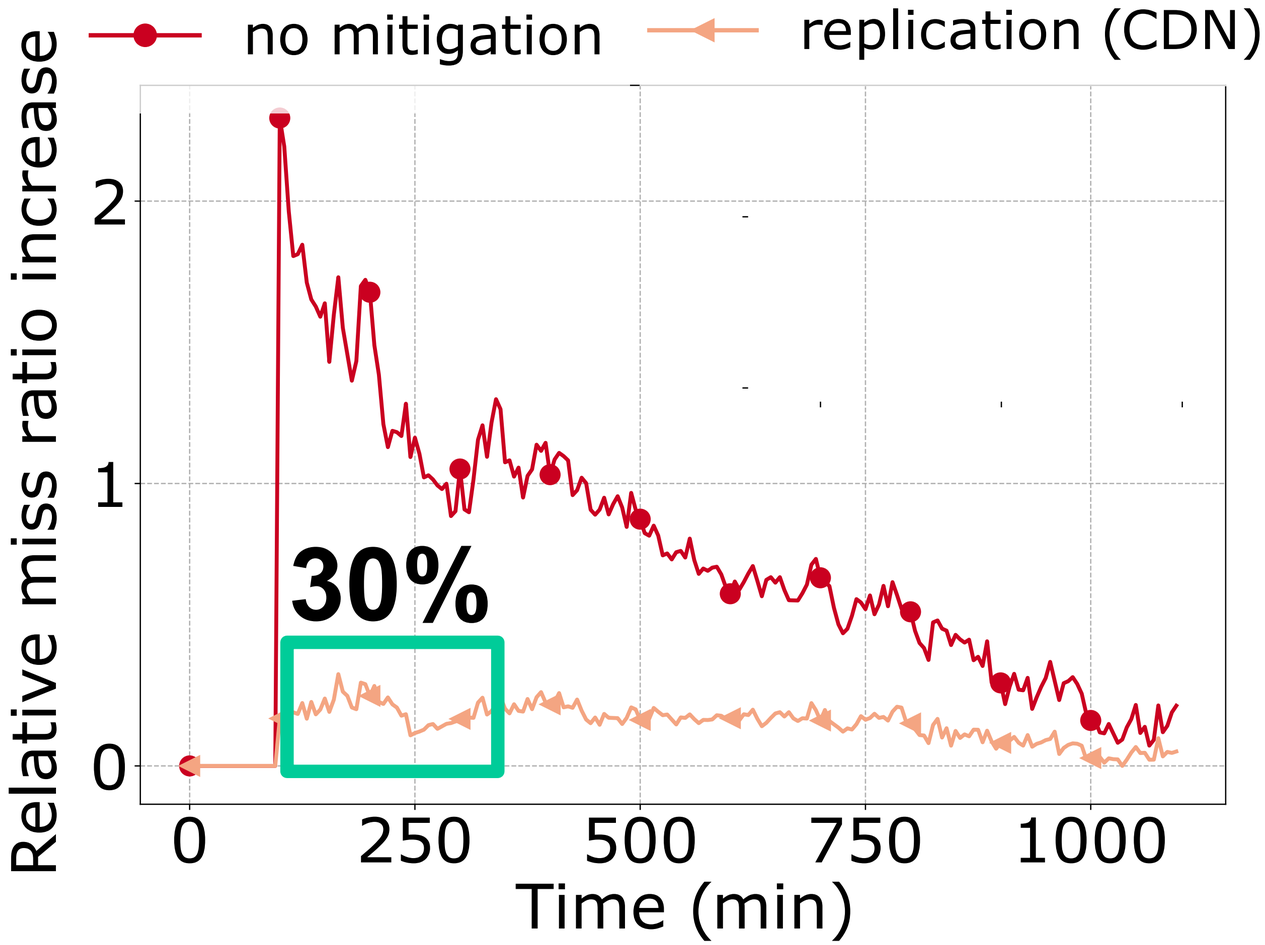
- **all** cached objects for **buckets** mapped to the unavailable server become unavailable
- => miss ratio spike => SLA violation

State-of-the-art solution

Replication

Limitations

- cannot remove spike



State-of-the-art solution

Replication

Limitations

- cannot remove spike

Problem: write load imbalance

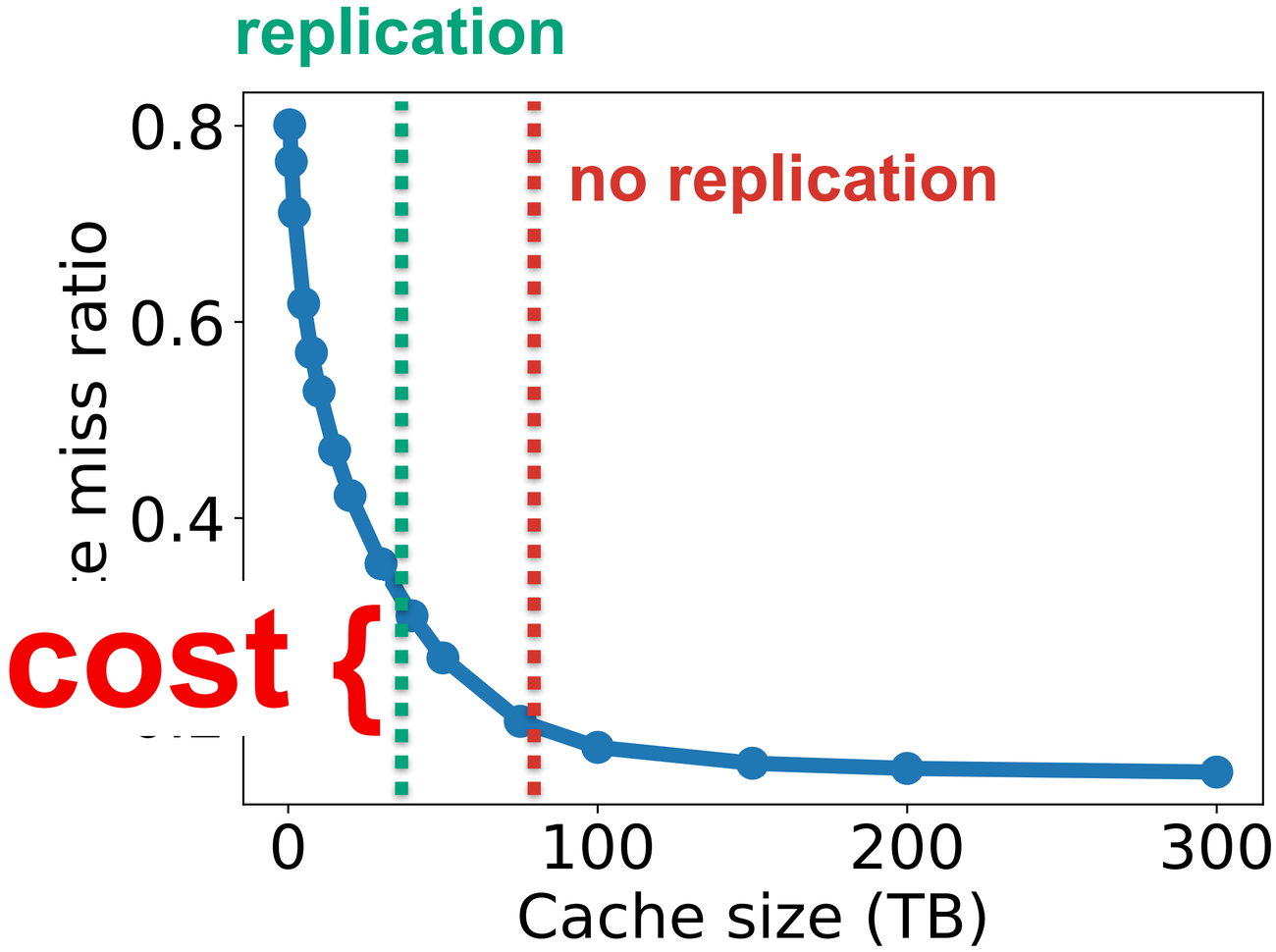
- Cause: coarse-grained load balancing
- Production: max/min server load = 2.5
  - replicated objects are evicted at different times
  - SSDs wear out at different rates

State-of-the-art solution

Replication

Limitations

- cannot remove spike
- waste space and cache fewer data



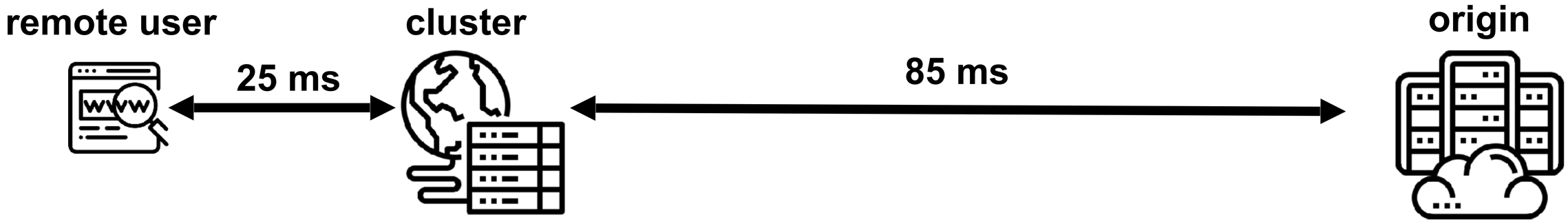
Coded CDN (C2DN) design overview

- **Erasure coding** with **parity rebalance**
  - reduce storage overhead and cache more objects (lower miss ratio)
  - reduce write load imbalance and eliminate miss ratio spike
    - data chunk: consistent hashing, parity chunk: solved as a MaxFlow problem
- Several other optimizations, e.g., hybrid redundancy, transparent coding, sub-chunking

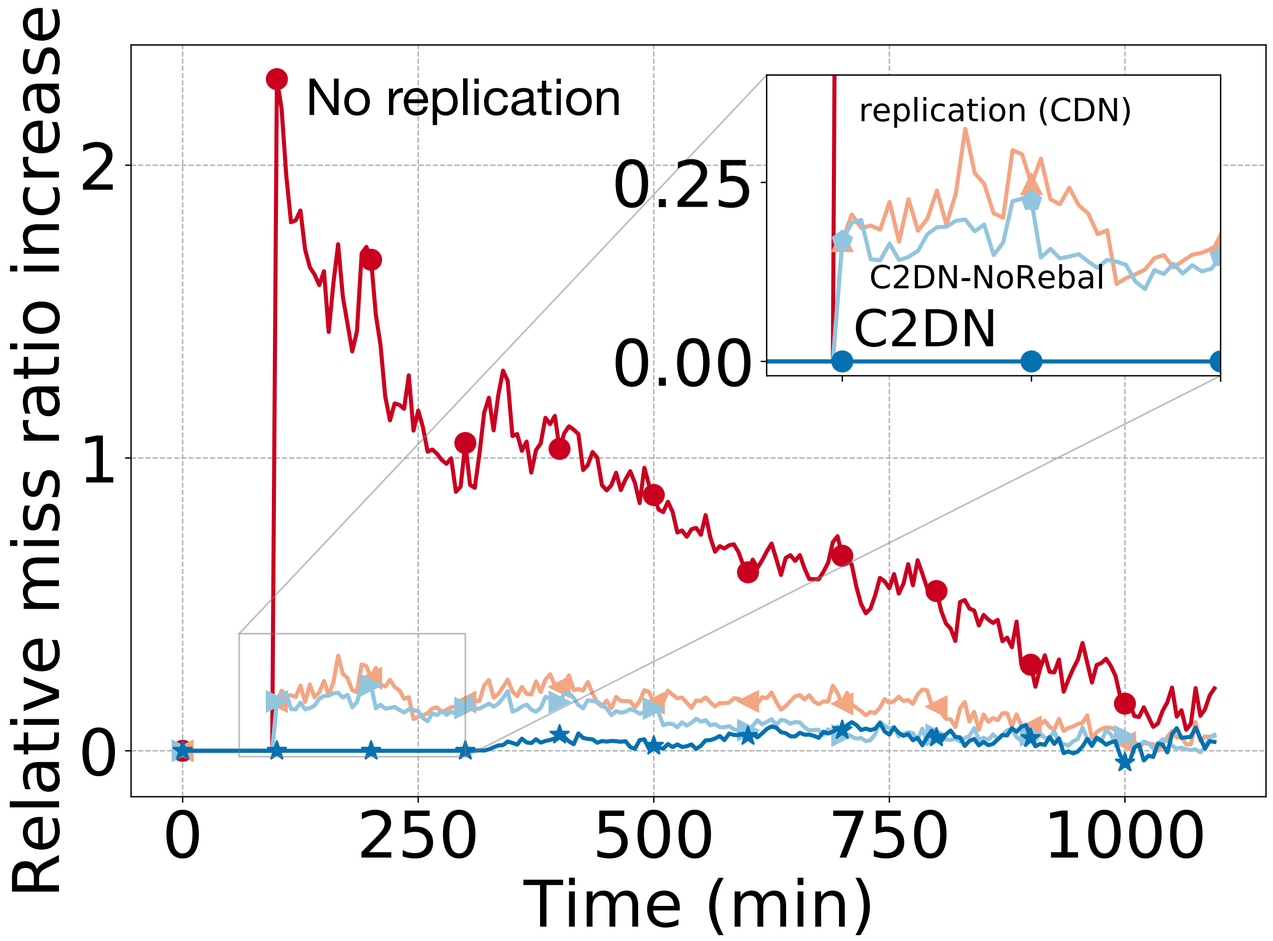
Evaluation setup

- Built C2DN using Apache Trafficserver
- Replayed week-long production traces from Akamai
- Evaluated using three AWS regions
- Simulated more scenarios

- Metrics:
  - miss ratio spike
  - average miss ratio
  - write load balance



No more miss ratio spike



Reducing normal case miss ratio

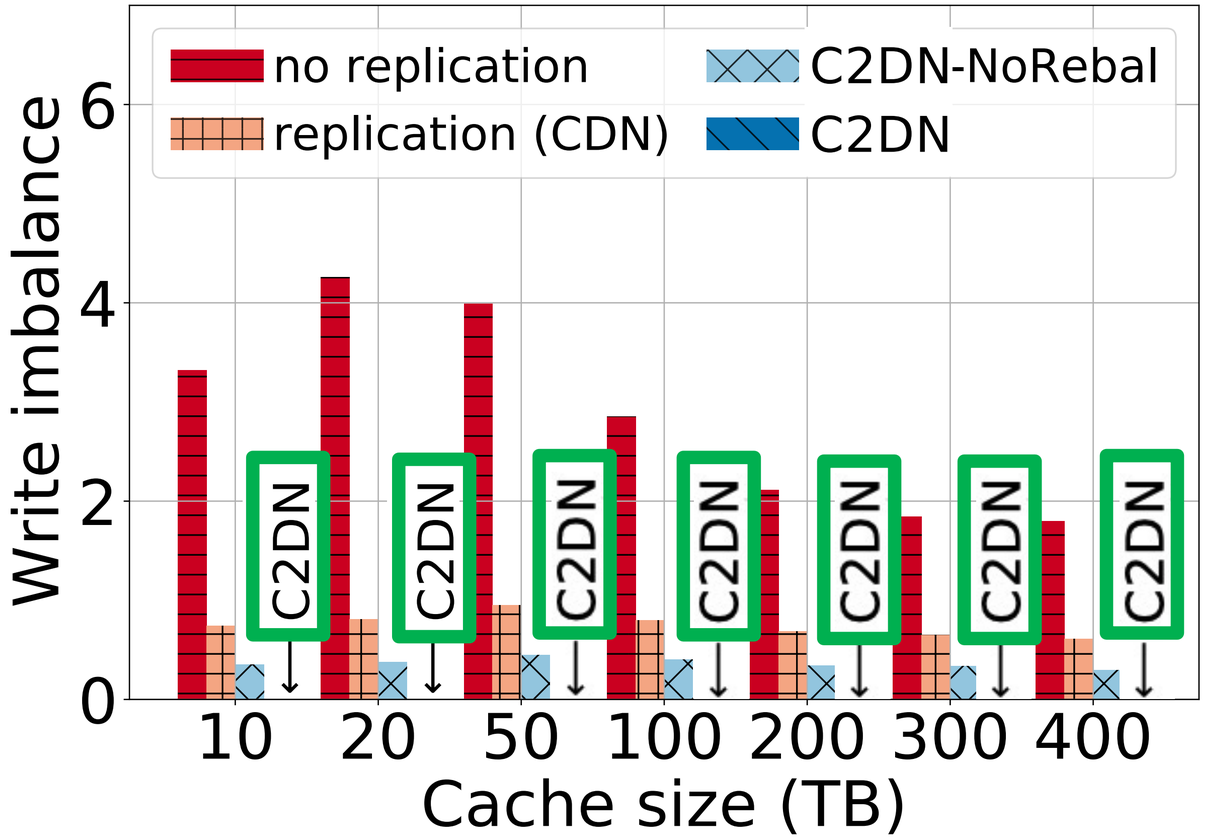
The benefits of erasure coding

Cache size	Byte miss ratio (bandwidth) reduction
production	21%
2x production	16%
4x production	5%

Erasure coding reduces storage overhead

Parity rebalance allows chunks to be evicted at a similar time

Near-perfect write load balance



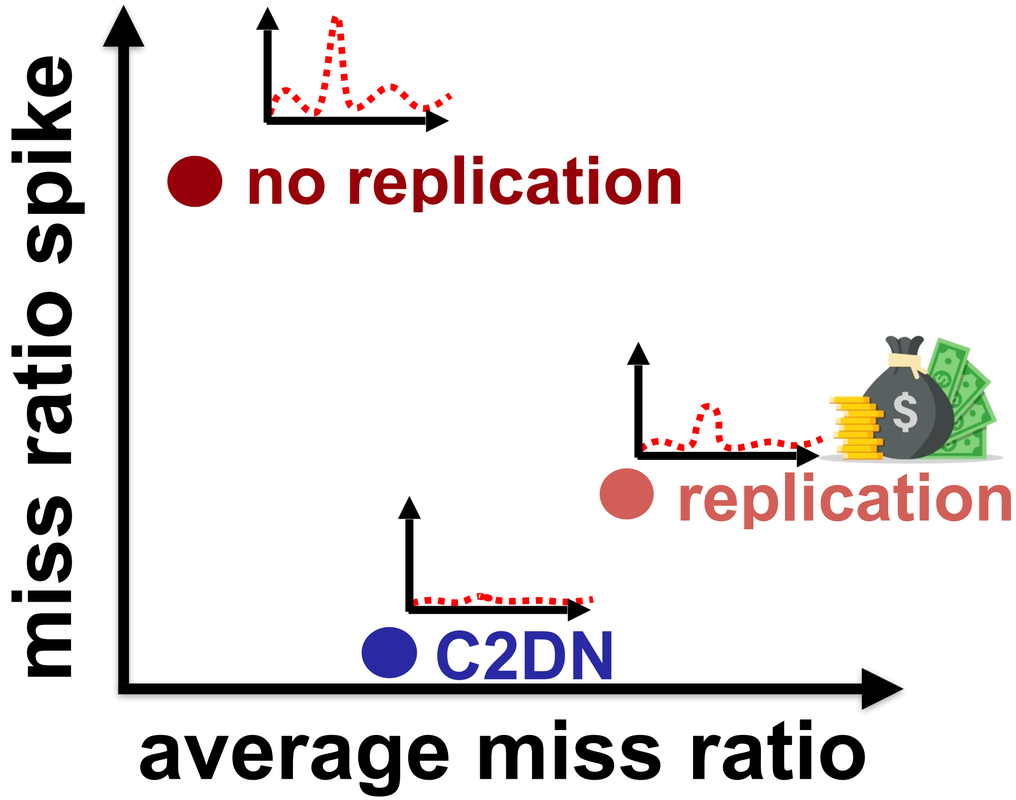
Write imbalance: max server write / min server write

In the thesis

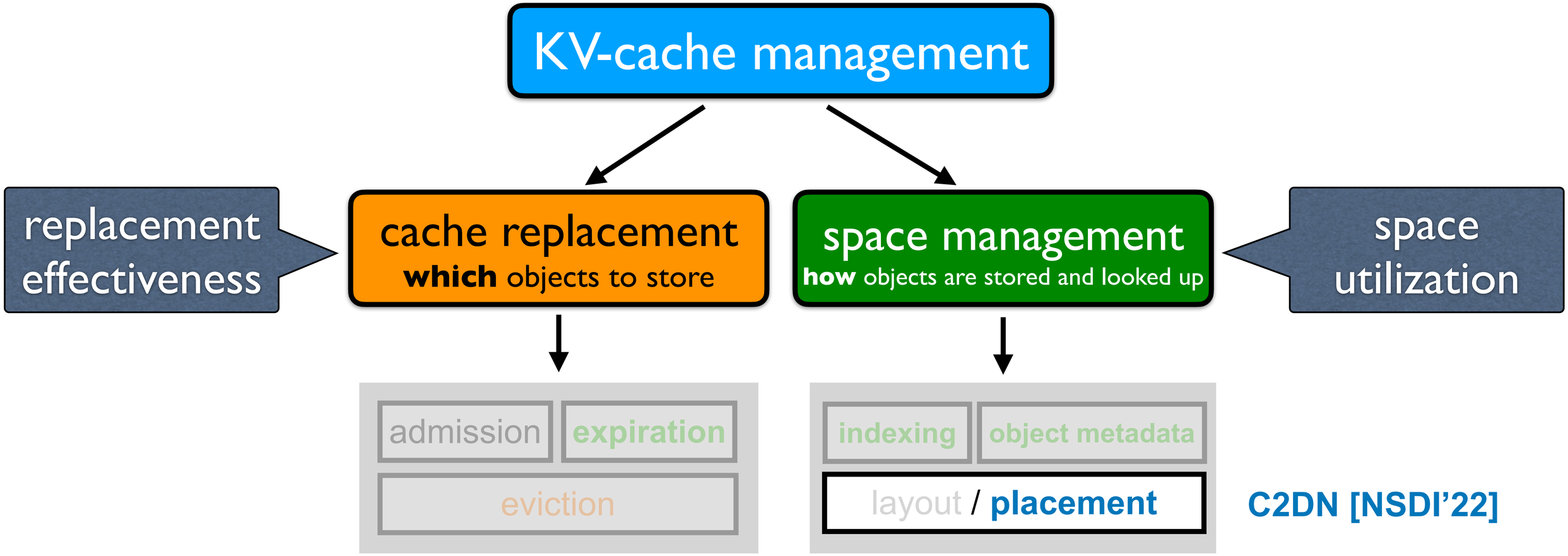
- Time-to-first-byte latency and content download time
  - no noticeable latency change
- CPU and disk usage
  - manageable increase
- Different number of unavailabilities, different workloads, different parameters…

Summary

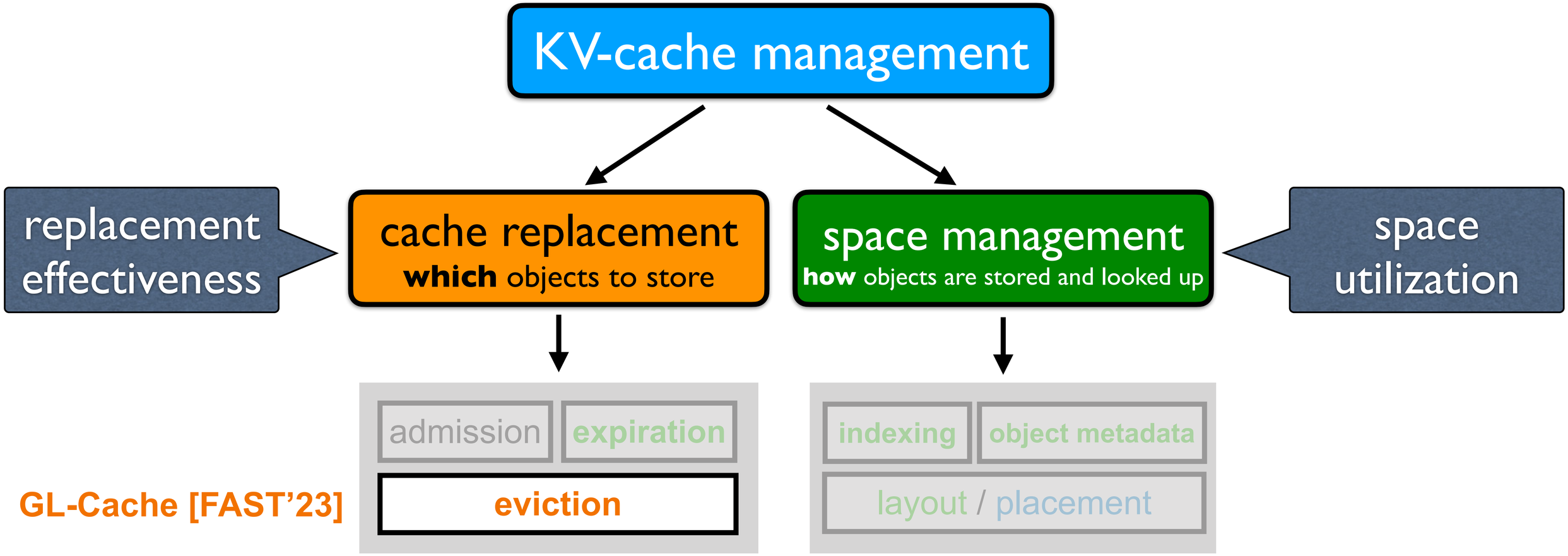
- Unavailability and write load imbalance are common in CDN edge clusters
- Traditional approach (replication) is not effective nor efficient for CDN caches
- C2DN reduces both normal case miss ratio and miss ratio spike



Key-value cache management system



Key-value cache management system



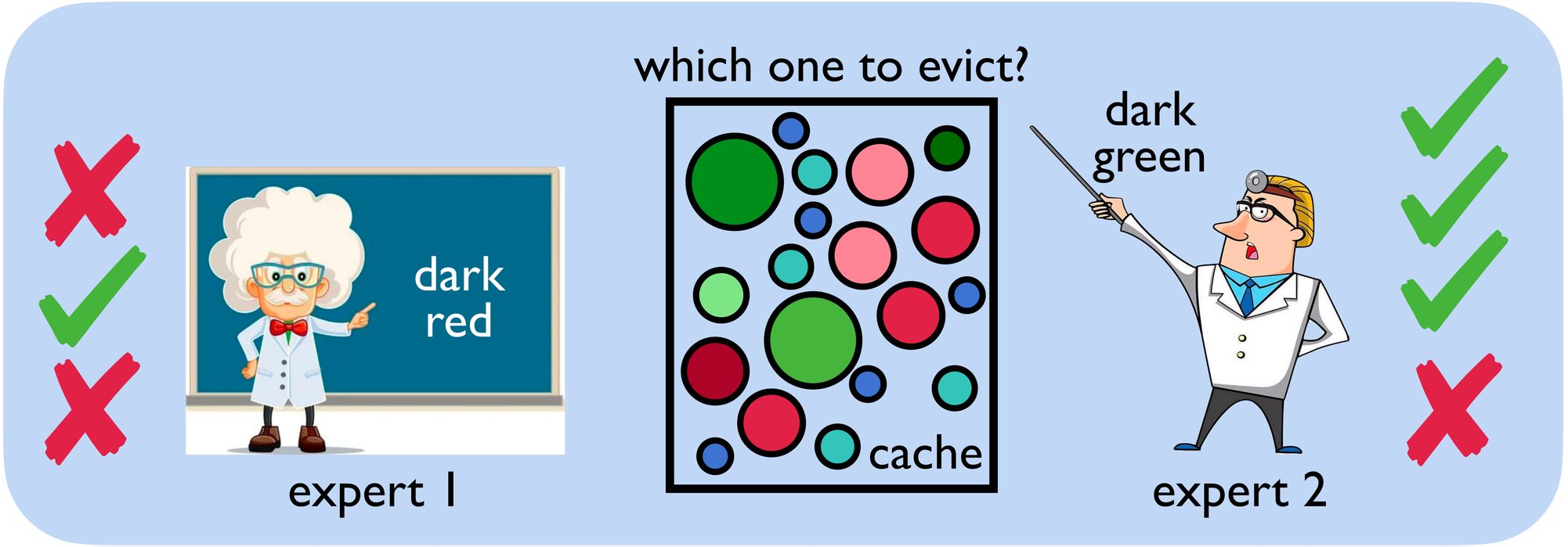
Part IV. Group-level Learned Cache (GL-Cache)

How to make machine learning practical for caching

Introduction

Learned caches

Learning from simple experts (e.g., LeCaR[1])



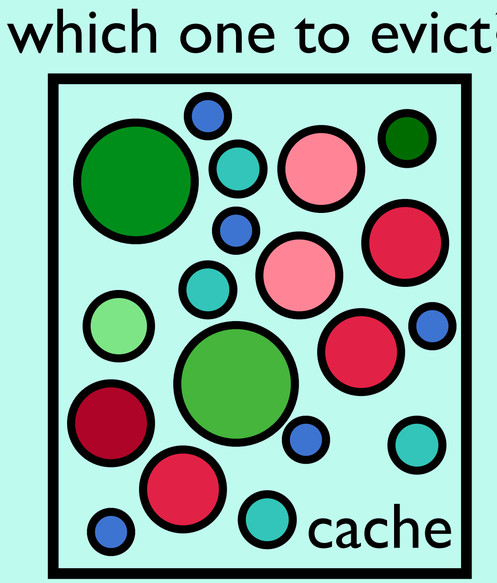
maintain two sets of metadata is expensive and complex
delayed reward

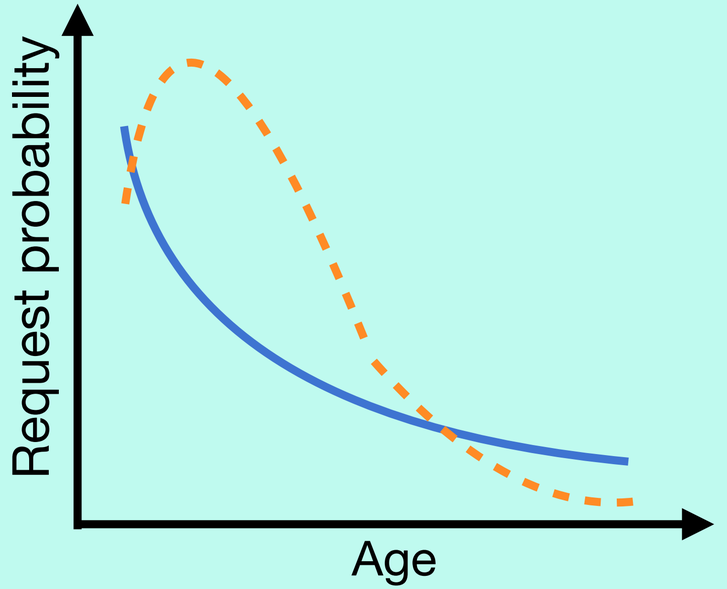
[1] Vietri, Giuseppe, et al. "Driving Cache Replacement with ML-based LeCaR." *HotStorage*. 2018.

Introduction

Learned caches

Learning from distribution (e.g., LHD[2])





can only use a limited number of features => low efficiency upper bound

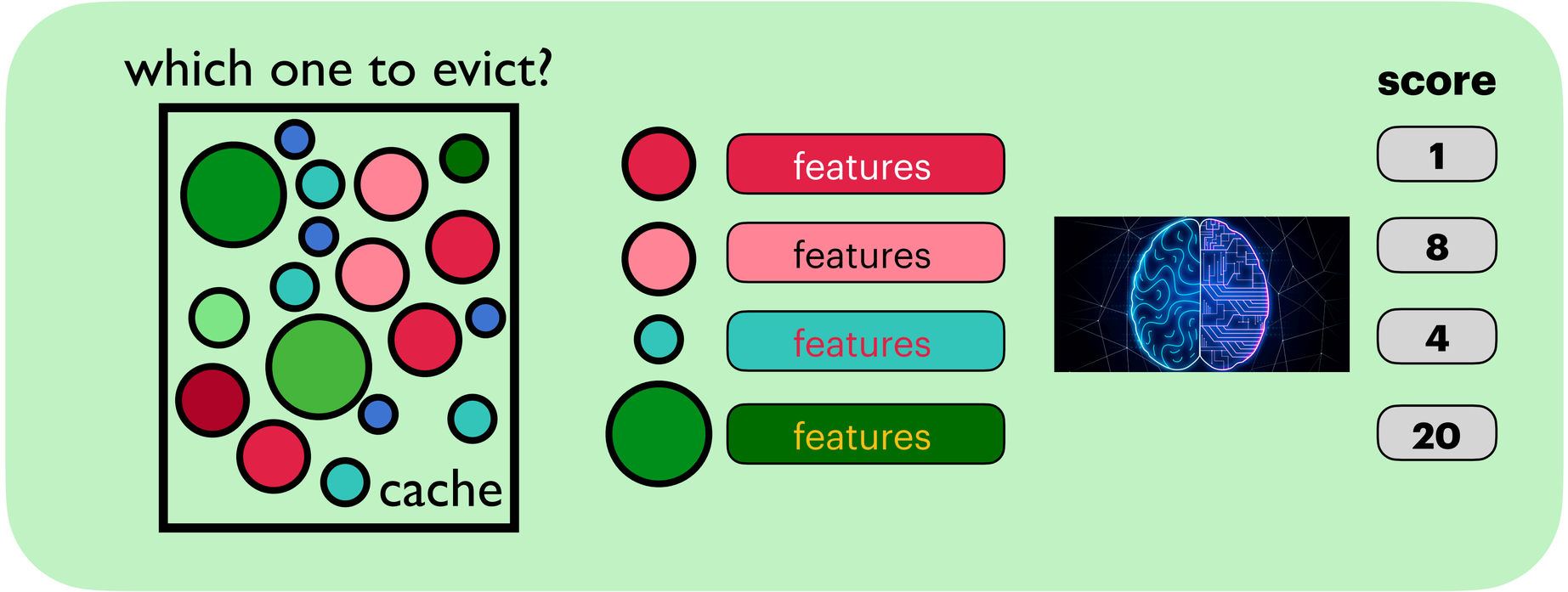
require sampling many objects to compare at each eviction => low throughput

[2] Beckmann, Nathan, et al. "LHD: Improving Cache Hit Rate by Maximizing Hit Density." *NSDI*. 2018.

Introduction

Learned caches

Object-level learning (e.g., LRB[3])

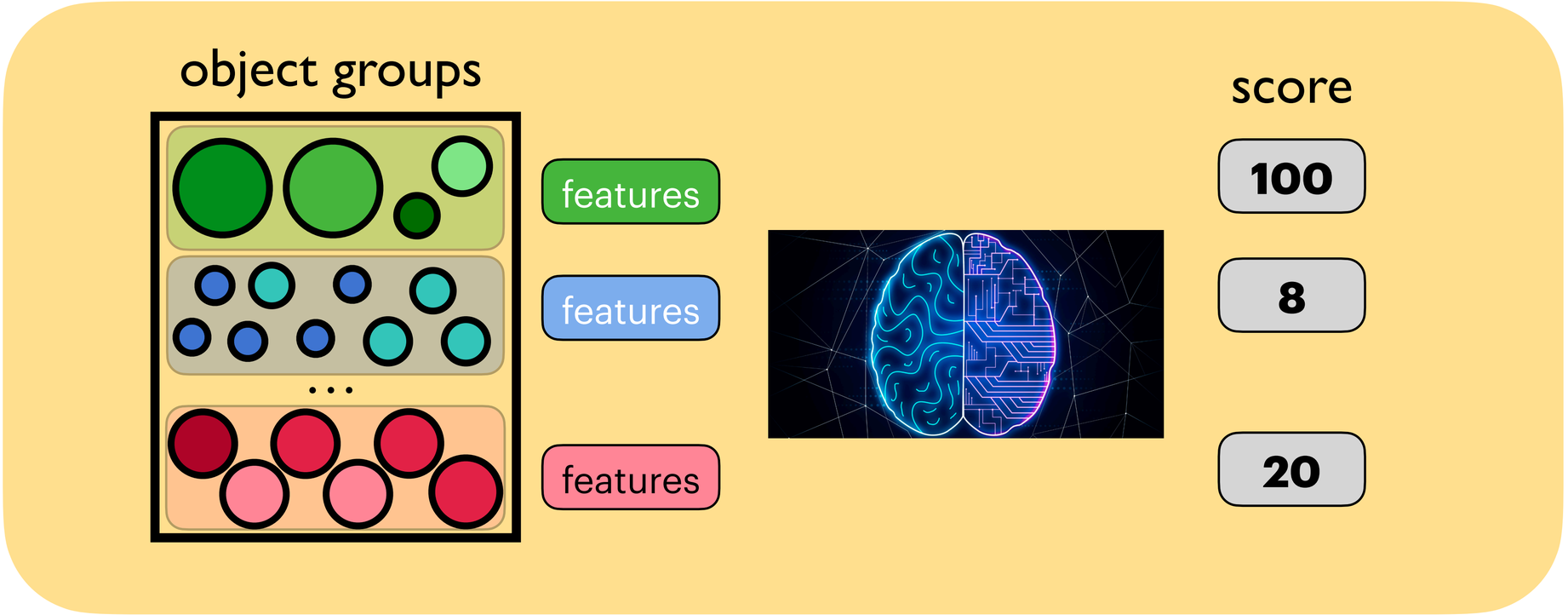


leverage more features than other learned caches
sampling and inference at each eviction => very very very **slow**

[3] Song, Zhenyu, et al. "Learning relaxed belady for content distribution network caching." *NSDI* 2020.

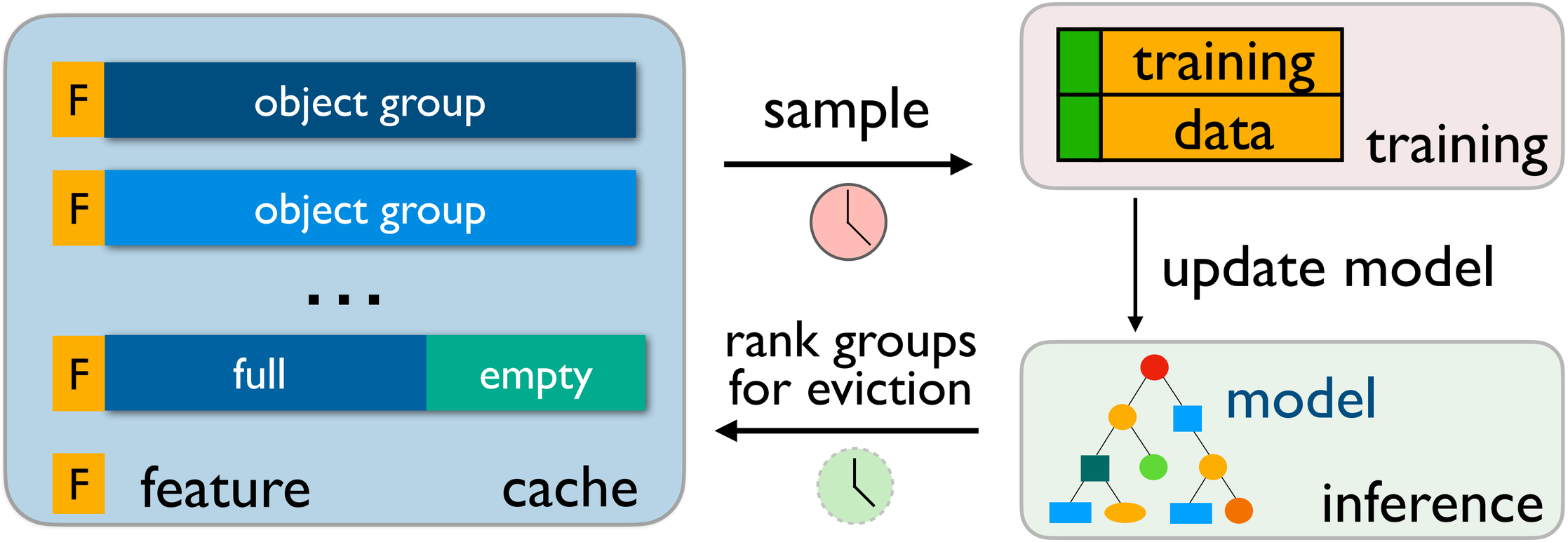
New idea

Group-level Learning (this work)



utilizes multiple features, while amortizes overheads
groups accumulate more information and are easier to learn

GL-Cache architecture



Design decision

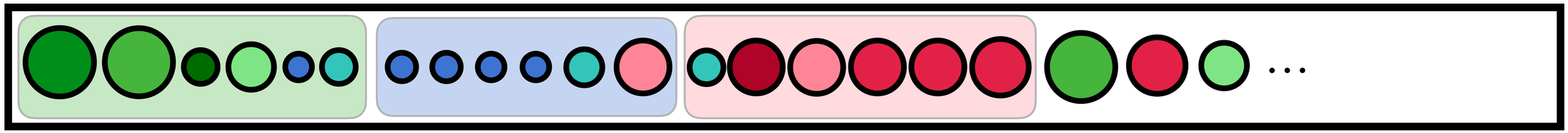
- **How** does GL-Cache **group** objects
- **What** does GL-Cache **learn**
- **How** does GL-Cache **learn**
- **How** does GL-Cache **evict**

Design decision

- **How** does GL-Cache **group** objects
- **What** does GL-Cache **learn**
- **How** does GL-Cache **learn**
- **How** does GL-Cache **evict**

How does GL-Cache group objects

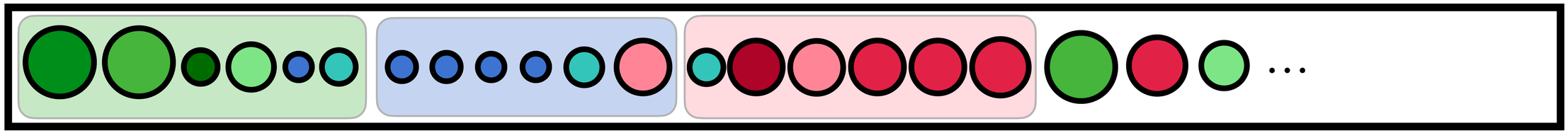
Insertion-time-based grouping



- Why?
  - objects inserted at similar time are similar
  - simple and generally applicable
  - can be implemented on Segcache to harness its benefits

What does GL-Cache learn

A new utility function



Which group is a better eviction candidate?

- Quantify the usefulness of object groups
- Properties desired
  - smaller object => larger utility
  - sooner-to-be-accessed => larger utility
  - group size one => Belady's MIN (weighted by size)
  - easy and accurate to track online

object utility at time t

$$U_o(t) = \frac{1}{T_o(t) \times s_o}$$

group utility

$$U_{group}(t) = \sum_{o \in group} \frac{1}{T_o(t) \times s_o}$$

$T_o(t)$ time till next request since t

s_o object size

* requires future information

In the thesis

- GL-Cache features
  - static and dynamic features
- GL-Cache training
  - once a day
- GL-Cache inference
  - periodically
- GL-Cache eviction
  - merge-based and retains popular objects during eviction

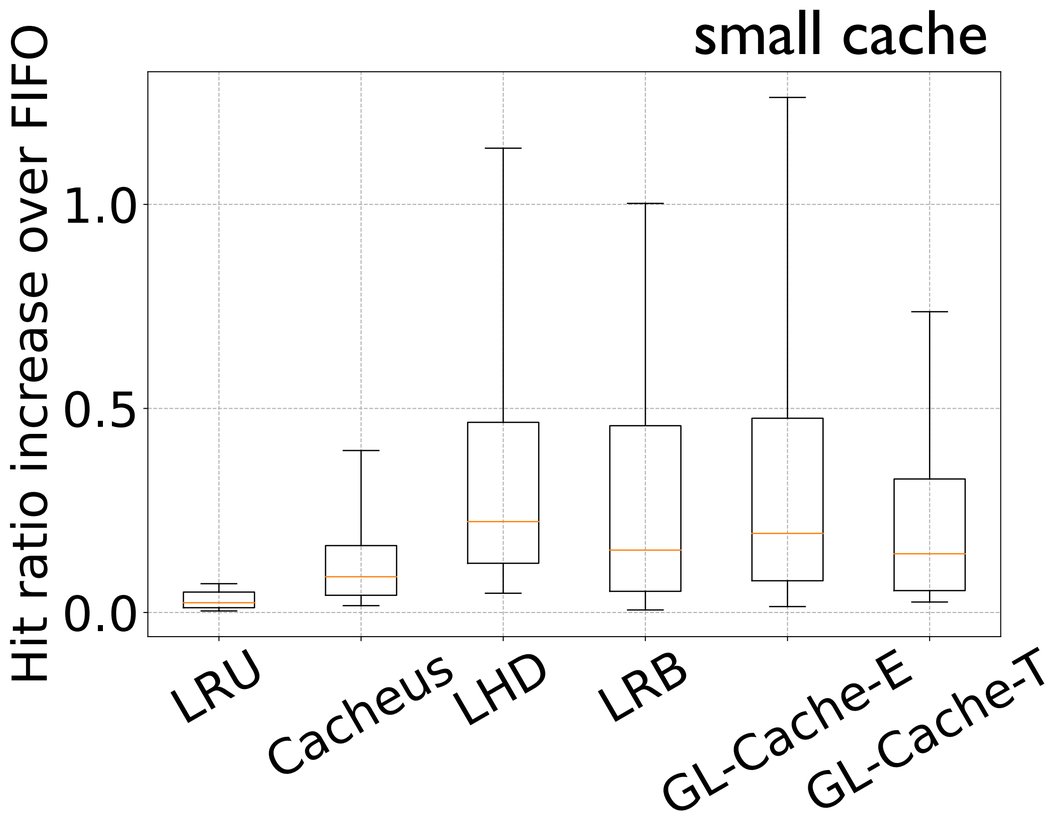
Evaluation setup

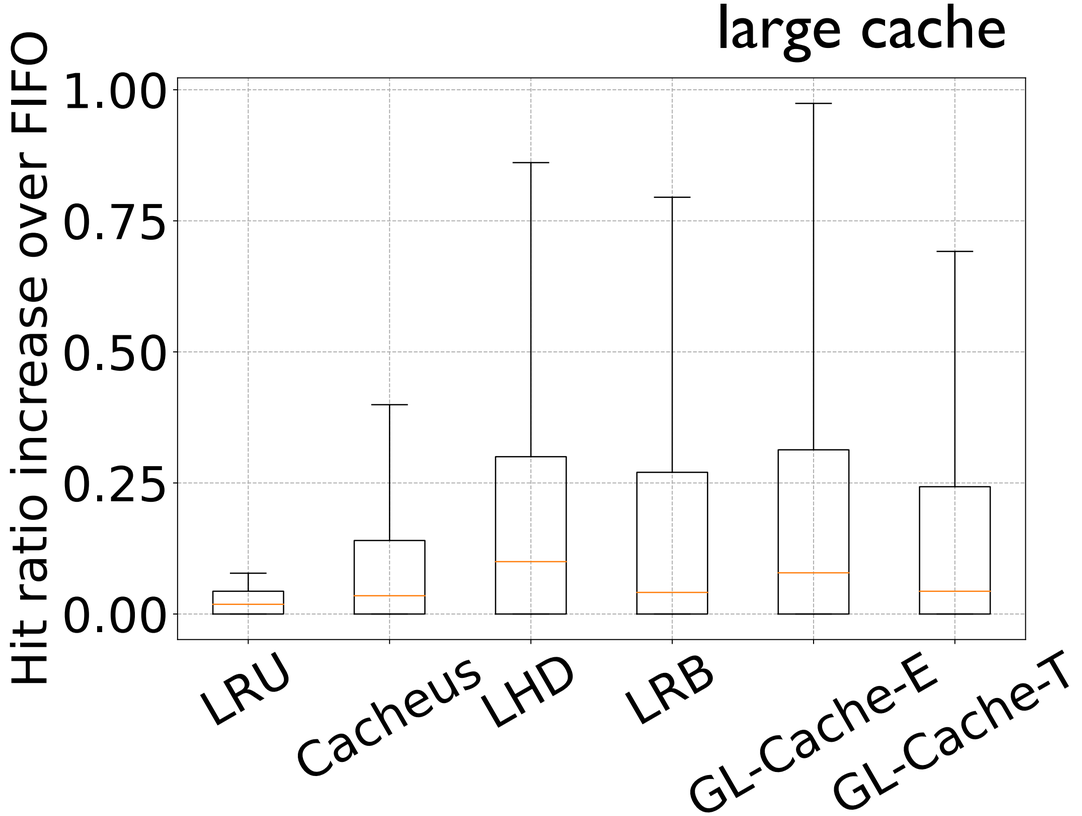
- Traces
  - 103 Cloudphysics traces
  - 14 MSR traces
  - 1 Wikipedia trace
- Two modes of GL-Cache
  - GL-Cache-E, GL-Cache-T
- Micro-implementation based on libCacheSim
  - LRU, CACHEUS, LHD, LRB
- Prototype implemented from Segcache

Evaluation setup

- Traces
  - 103 Cloudphysics traces
  - 14 MSR traces
  - 1 Wikipedia trace
- Two modes of GL-Cache
  - GL-Cache-E, GL-Cache-T
- Micro-implementation based on libCacheSim
  - LRU, CACHEUS, LHD, LRB
- Prototype implemented from Segcache

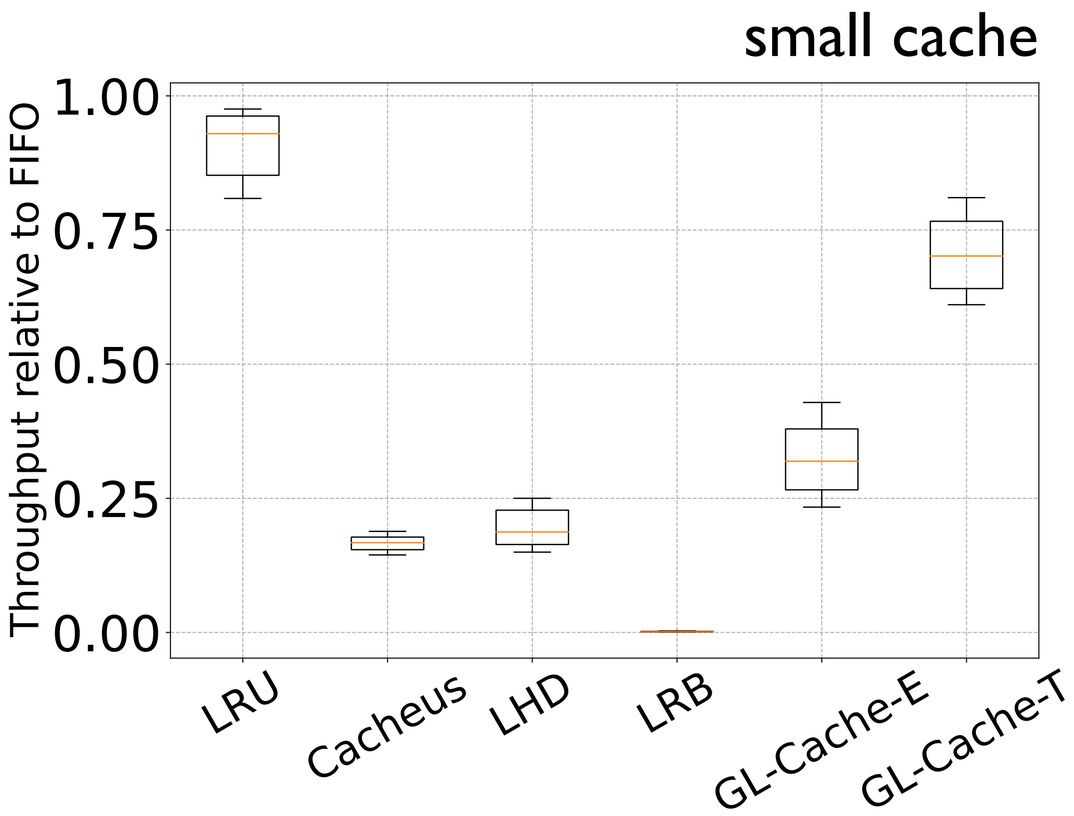
Efficiency

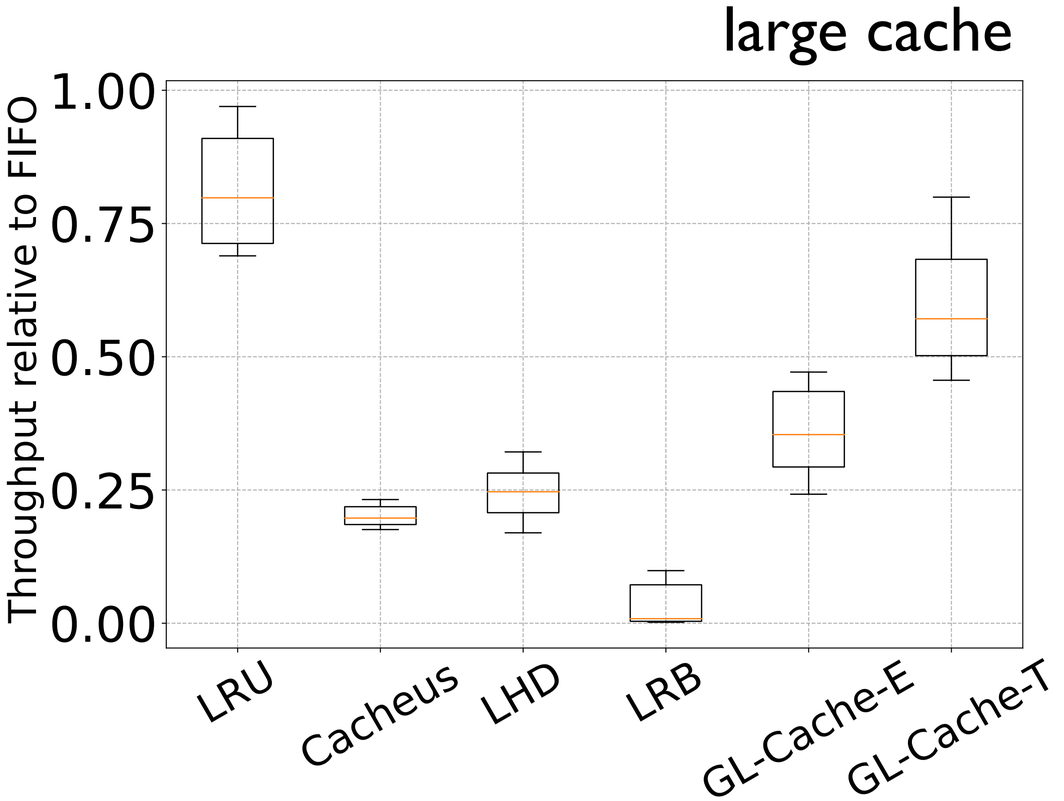




GL-Cache-E is slightly better than state-of-the-art algorithms
GL-Cache-T is close to LRB

Throughput

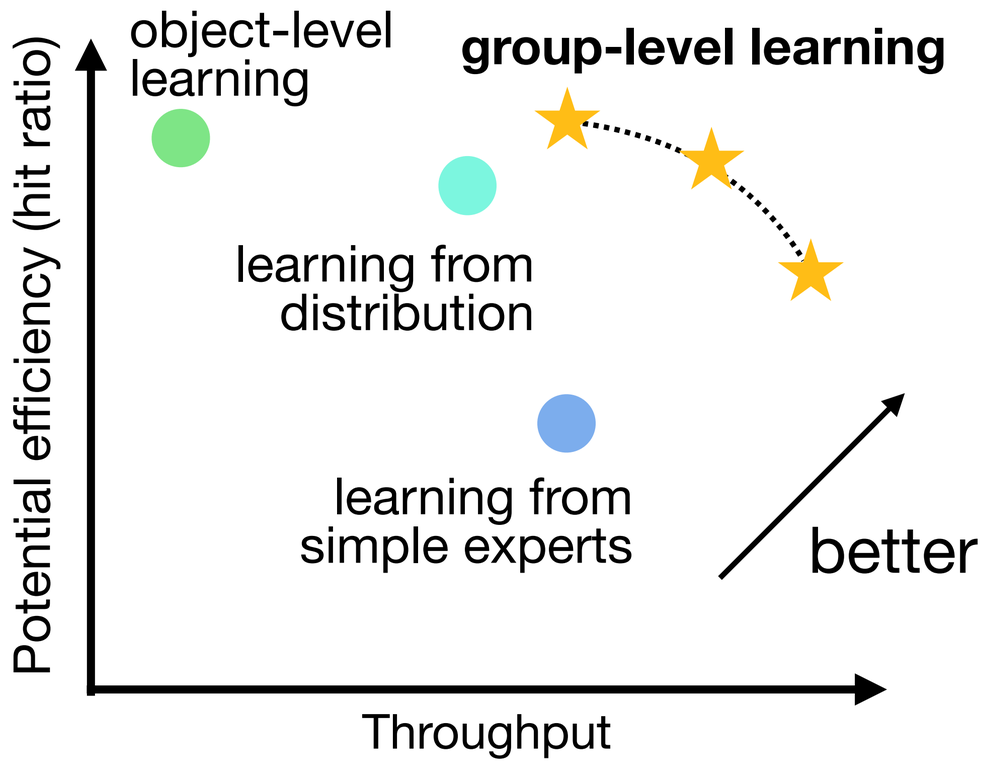




GL-Cache-E is faster than all state-of-the-art learned caches
GL-Cache-T is significantly faster

Summary

- Group-level learning
  - lower overhead
  - more information
  - a spectrum of algorithms



Part V. Proposed work

- An interpretable learned cache eviction algorithm
- A simple yet efficient FIFO-based cache eviction algorithm

Problems with existing learned caches

- Using machine learning as a black box
  - not interpretable and hard to debug
  - misalignment between miss ratio and the loss function
- Complexity
  - GL-Cache uses 3000 LOC vs LRU uses 300 LOC
  - intimidating for production engineers
  - error-prone

Proposed work 1: an interpretable learned cache

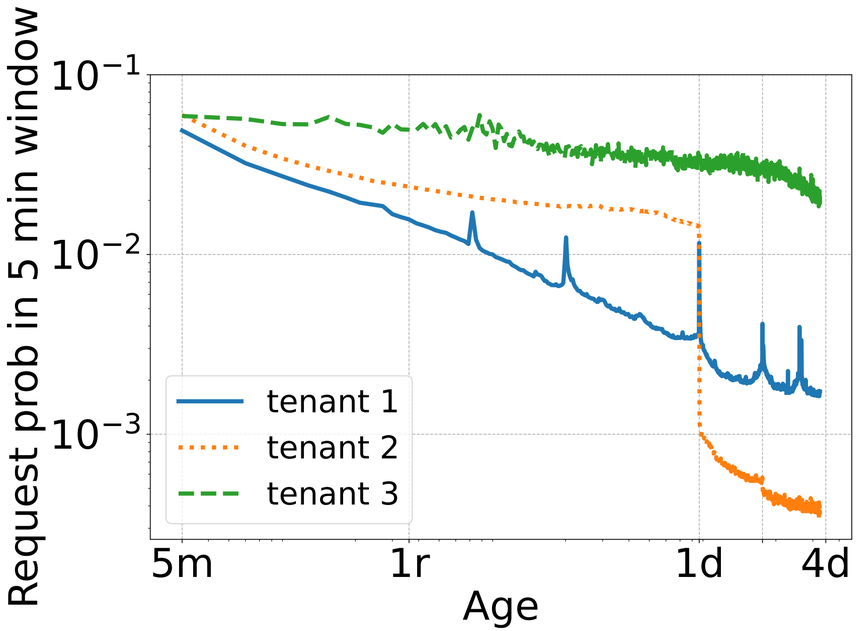
Goals

- Produce human-examinable and explainable predictions
  - fewer predictions
  - prediction is measurable
- Align the loss function directly with the cache metrics, e.g., miss ratio

Proposed work 1: an interpretable learned cache

Proposed approach

- Predict the object lifetime of each tenant
  - tenant: namespace, content type
  - different tenants often have different access patterns, e.g., streaming video, API calls vs images
- Why and how
  - more interpretable
  - fewer predictions
  - facilitate better data placement
  - can formulate as an optimization problem to minimize miss ratio



Proposed work 2: rethinking FIFO-based eviction algorithms

Background

- The evolution of eviction algorithms
  - mostly LRU-based
    - most algorithms reducing miss ratios are built upon LRU
    - most throughput-oriented systems use “weak LRU”, e.g., CLOCK
  - more and more complex
- Is LRU still the to-go algorithm?
  - many workloads today exhibit popularity decay

Proposed work 2: rethinking FIFO-based eviction algorithms

Preliminary work

- Why FIFO? Many benefits
  - fewer metadata
  - less computation
  - more scalable
  - flash friendly
  - popularity decay
- FIFO cannot keep popular objects in the cache
  - FIFO-Reinsertion (CLOCK)

Proposed work 2: rethinking FIFO-based eviction algorithms

Preliminary work

Common Wisdom:
FIFO-Reinsertion, CLOCK are **less efficient** than LRU

We observe:
FIFO-Reinsertion, CLOCK are **more efficient** than LRU

- A large-scale eviction algorithm study
  - 10 datasets, 5307 traces from 2007-2020
  - Block, key-value, object
  - 814 billion requests , 55.2 billion objects

Moreover,
Quickly removing new objects makes eviction algorithms more efficient

Proposed work 2: rethinking FIFO-based eviction algorithms

Proposed work

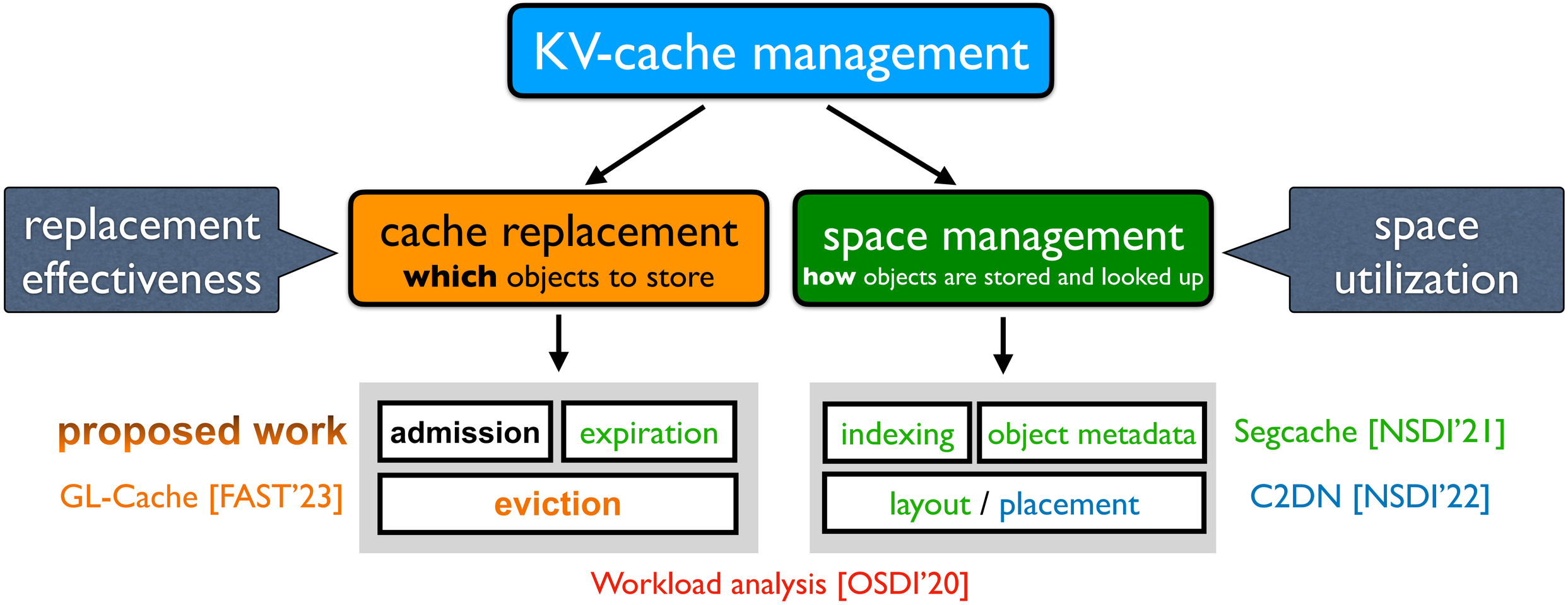
Common Wisdom:
FIFO-Reinsertion, CLOCK are **less efficient** than LRU

We observe:
FIFO-Reinsertion, CLOCK are **more efficient** than LRU

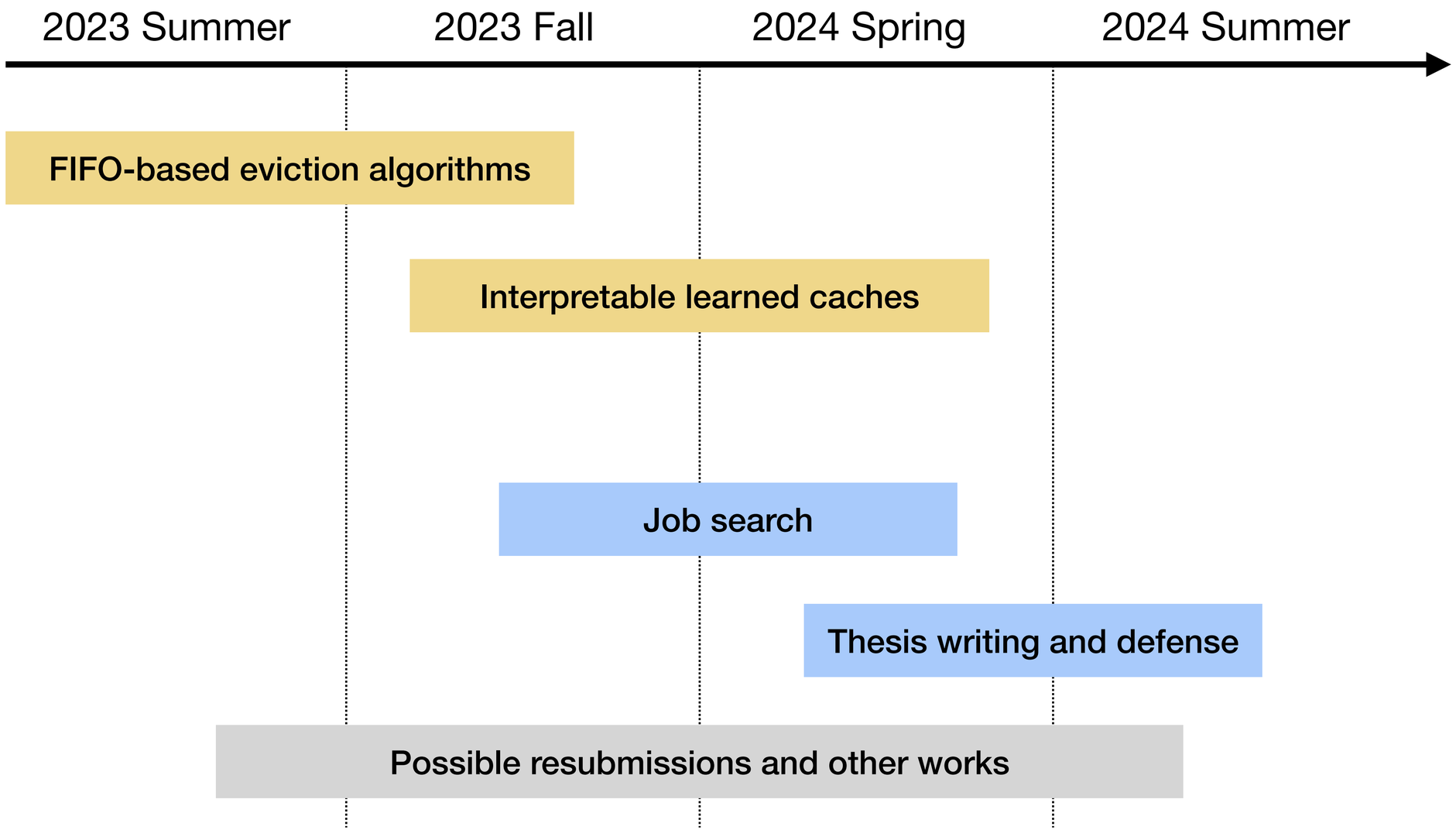
- Design a FIFO-based eviction algorithm
  - simple
  - efficient
  - scalable

Moreover,
Quickly removing new objects makes eviction algorithms more efficient

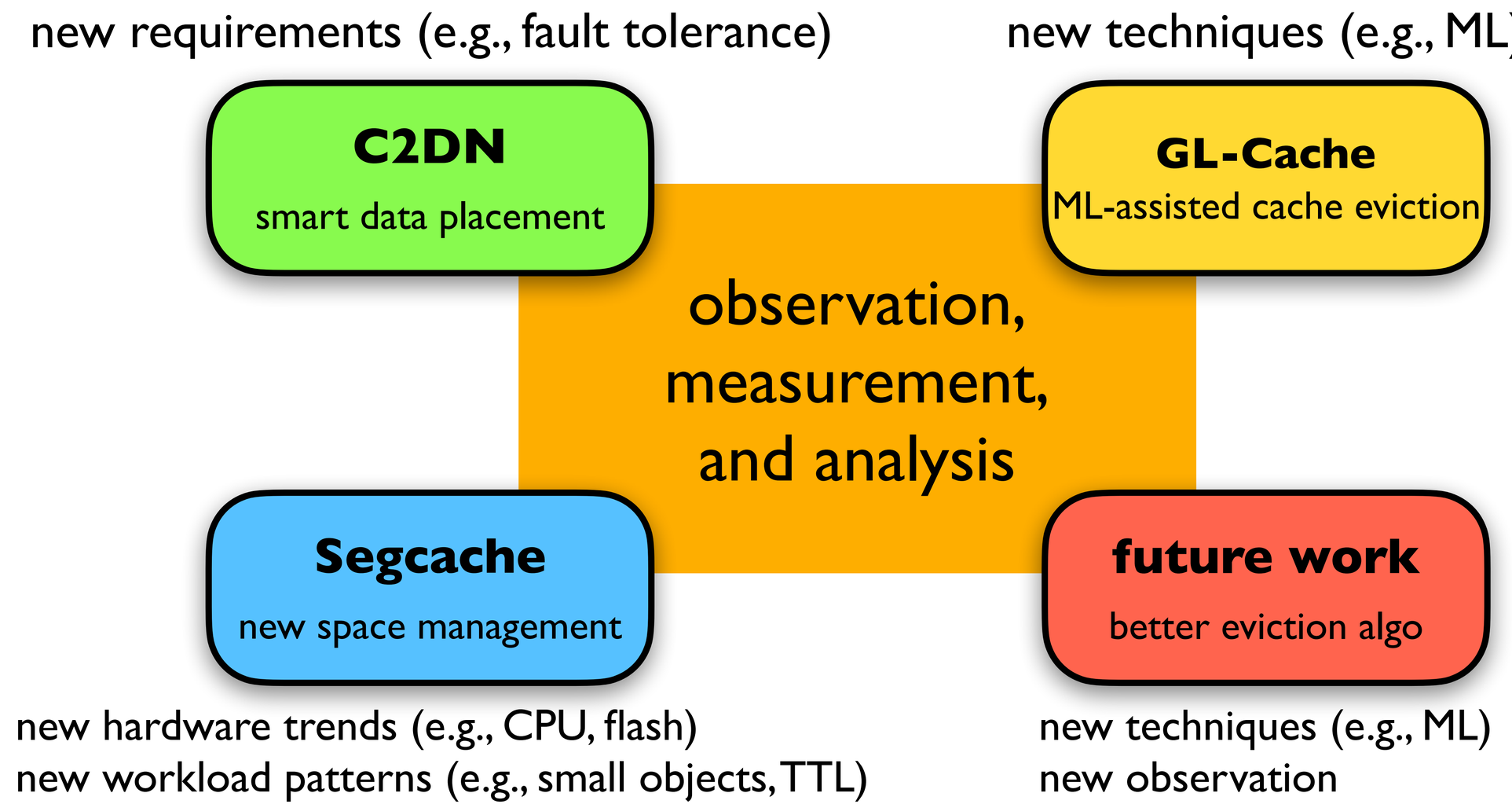
The full picture of KV-cache management system



Timeline


2023 Summer
2023 Fall
2024 Spring
2024 Summer
FIFO-based eviction algorithms
Interpretable learned caches
Job search
Thesis writing and defense
Possible resubmissions and other works

Summary


new requirements (e.g., fault tolerance)
new techniques (e.g., ML)
C2DN
smart data placement
GL-Cache
ML-assisted cache eviction
observation, measurement, and analysis
Segcache
new space management
future work
better eviction algo
new hardware trends (e.g., CPU, flash)
new workload patterns (e.g., small objects, TTL)
new techniques (e.g., ML)
new observation