GL-Cache: Group-level learning for efficient and high-performance caching

21st USENIX Conference on File and Storage Technologies

Juncheng Yang; Ziming Mao; Yao Yue; K. V. Rashmi

Paper Notes

By JeongHa Lee

Abstract

Web applications rely heavily on software caches to achieve low-latency, high-throughput services. To adapt to changing workloads, three types of learned caches (learned evictions) have been designed in recent years: object-level learning, learning-from-distribution, and learning-from-simple-experts. However, we argue that the learning granularity in existing approaches is either too fine (object-level), incurring significant computation and storage overheads, or too coarse (workload or expert-level) to capture the differences between objects and leaves a considerable efficiency gap.

In this work, we propose a new approach for learning in caches ("group-level learning"), which clusters similar objects into groups and performs learning and eviction at the group level. Learning at the group level accumulates more signals for learning, leverages more features with adaptive weights, and amortizes overheads over objects, thereby achieving both high efficiency and high throughput.

We designed and implemented GL-Cache on an open-source production cache to demonstrate group-level learning. Evaluations on 118 production block I/O and CDN cache traces show that GL-Cache has a higher hit ratio and higher throughput than state-of-the-art designs. Compared to LRB (object-level learning), GL-Cache improves throughput by 228× and hit ratio by 7% on average across cache sizes. For 10% of the traces (P90), GL-Cache provides a 25% hit ratio increase from LRB. Compared to the best of all learned caches, GL-Cache achieves a 64% higher throughput, a 3% higher hit ratio on average, and a 13% hit ratio increase at the P90.

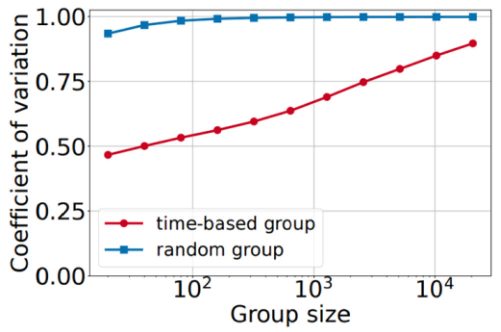
Problem Statement and Research Objectives

Learning in caching

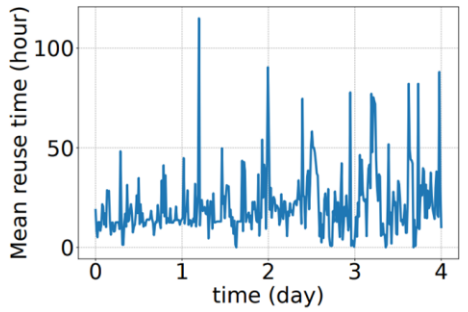
- Object-level learning
  - Learns the next access time for each object using dozens of object features and evicts the object with the furthest predicted request time.
  - Predicting and ranking objects at each eviction incurs significant computation and storage overheads;
    ➔ For example, LRB suffers from a 775× slow down compared to LRU.
- Learning-from-simple-experts
  - Learns the weights of experts and highly depends on the choice of the experts.
  - Existing systems use simple experts and cannot leverage features not considered by the experts.
- Learning-from-distribution
  - Use request probability distributions to inform eviction decisions. For example, LHD [7] measures object hit density using age and size, and evicts the object with the lowest hit density.
  - It still has a lower throughput compared to simple heuristics (e.g., LRU) because it must randomly sample and compare many objects at each eviction.
  - The existing design (e.g., LHD [7]) does not leverage object features other than age and size, limiting its potential for high efficiency

Proposed Method

1. How to cluster objects into groups? (§ 3.3)
   - Write-time based groups



(a)



(b)

Fig. 2: a) Objects grouped using write time have more similar (smaller coefficient of variation) mean reuse time than objects grouped randomly. As group size increases, write-time-based grouping become closer to random grouping. b) Different object groups written at different times exhibit a large variation in mean reuse time.

2. How to compare the usefulness of object groups? (§ 3.4)
   - The benefit of evicting one object *o* is proportional to its size s_o and time till next access $T_o(t)$ from current time *t* .

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

Proposed Method

3. How to learn the utility of object groups? (§ 3.5)

- object group features
  - Static features: request rate | write rate | miss ratio | mean object size
  - **Dynamic features**: **age** | **# of requests** | **# of requested object**
- Training : data features & utilities
  - generates new training data by sampling cached object groups, and it copies the features of the sampled groups into a pre-allocated memory region.
  - When an **object *o* from a sampled group is requested**, GL-Cache can **calculate the $T_o(t)$** (time till next request since sampling) and **object utility** using Eq. 1 and **add the object utility into the group utility**.
  - A sampled group may be evicted before being used for training. ➔ **keeps ghost entries** for objects.
  - GL-Cache **retrains the model every day**
- Inference: **performs an inference every $F_{eviction} \times N_{group}$ groups**.
  - $F_{eviction}$ is the fraction of ranked groups to evict using one inference.
  - N_{group} is the total number of groups.

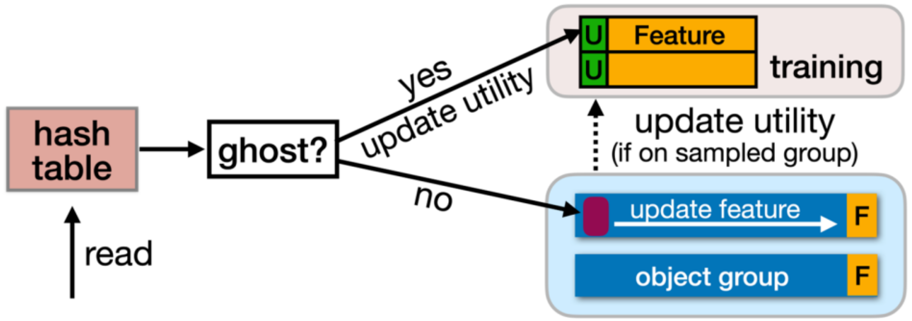


Fig. 3: The read flow in GL-Cache.

Proposed Method

4. How to perform evictions at group level? (§ 3.6)

- Upon each eviction, GL-Cache **picks the least useful object group** and **merges it with the N_{merge} –1 object groups that are closest with respect to write time**.
  - N_{merge} : Number of object groups to merge each eviction
- Unlike group selection, which uses ranking, **object selection** uses a simple metric based on object age and size: ***1 / (size·age)***
  - age is the time since the last access.

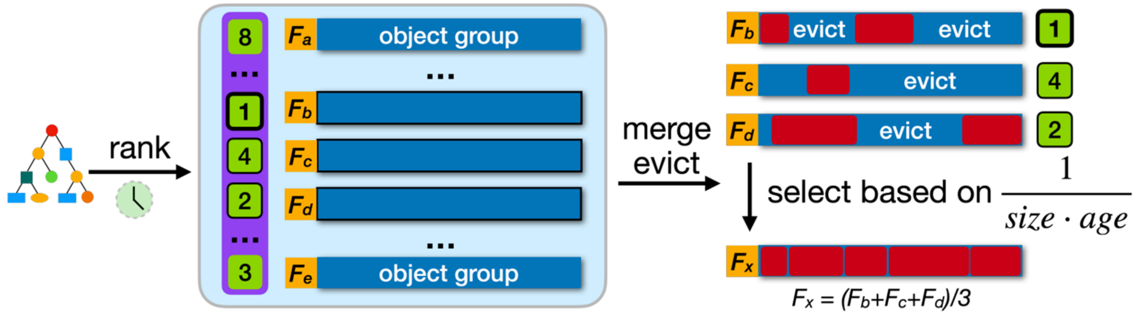


Fig. 4: Object group utility prediction and merge-based group eviction in GL-Cache.

Table 2: Parameters used in the design.

Para	Meaning
S_{group}	Size of an object group (in number of objects or bytes)
N_{merge}	Number of object groups to merge each eviction
$F_{eviction}$	Each inference evicts $F_{eviction}$ fraction of ranked groups

Evaluation and Results

1. Will group-based eviction limit the efficiency upper bound when compared to object-based eviction?

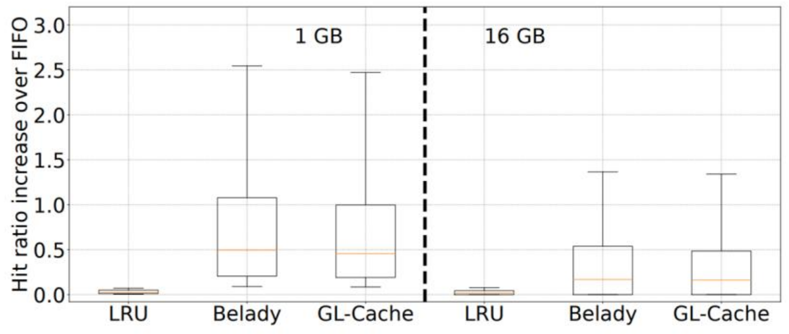
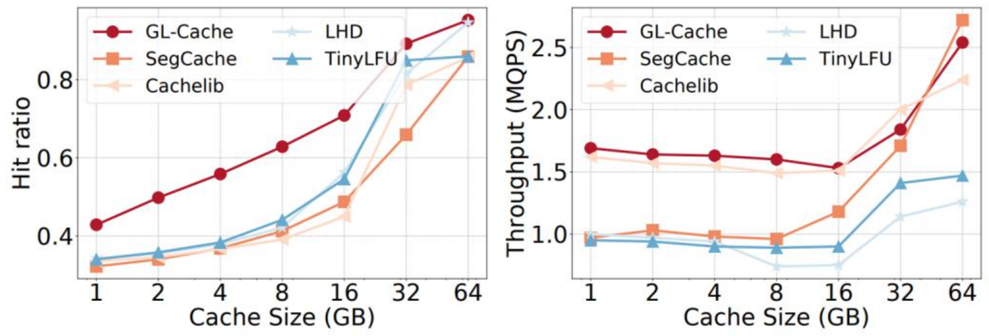


Fig. 5: With oracle assistance, group eviction can achieve a similar hit ratio improvement as object eviction.

2. Can GL-Cache improve hit ratio and efficiency over other learned caches?



(a) Hit ratio (b) Throughput

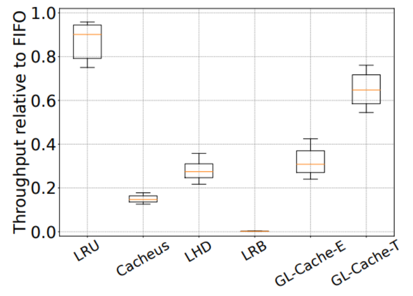
Fig. 6: Prototype evaluation of a CloudPhysics trace.

Table 4: Comparing LRB and GL-Cache-E on the Wikimedia trace used in LRB paper [87]. We use miss ratio because it is more commonly used in web caches.

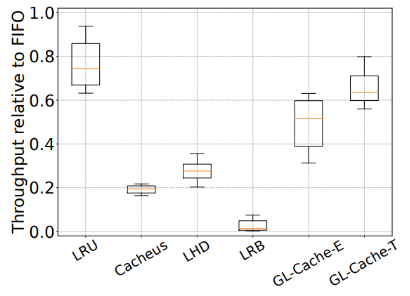
Algorithm	Miss ratio			Throughput (MQPS)		
Size (GB)	20	200	2000	20	200	2000
FIFO	0.39	0.16	0.025	7.62	7.91	9.68
LRB	0.24	0.048	0.016	0.01	0.04	0.07
GL-Cache-T	0.24	0.065	0.017	4.97	6.53	4.89
GL-Cache-E	**0.20**	**0.041**	**0.013**	2.55	3.91	4.20

Evaluation and Results

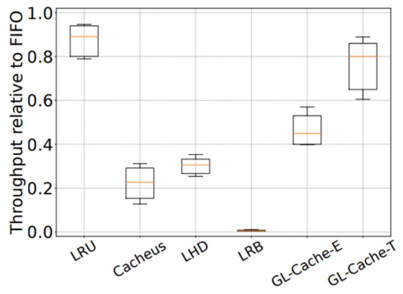
3. Can GL-Cache meet production-level throughput requirements and how much overhead does GL-Cache add?



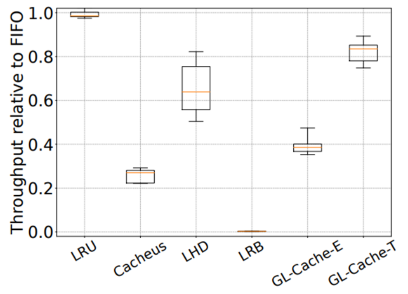
(a) CloudPhysics, small cache size



(b) CloudPhysics, large cache size



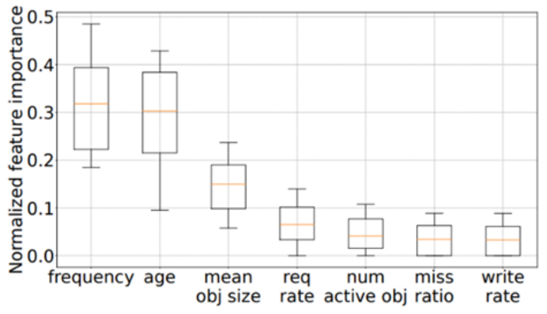
(c) MSR, small cache size



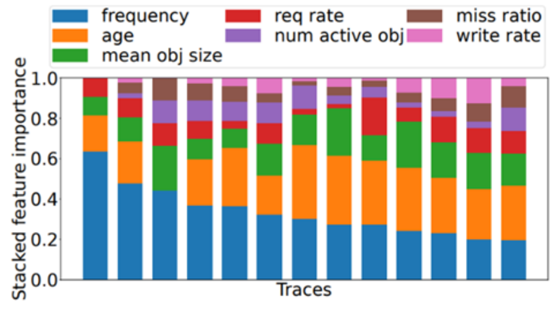
(d) MSR, large cache size

Fig. 8: Throughput relative to FIFO.

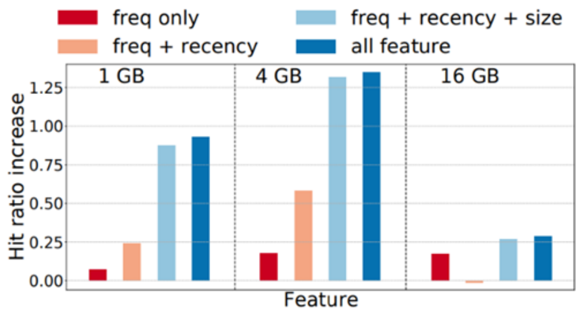
4. How does GL-Cache improve efficiency without compromising throughput?



(a) Feature importance across traces



(b) Feature breakdown across example traces.



(c) Feature utility case study.

Fig. 9: Feature case study.