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

Juncheng Yang, Carnegie Mellon University
Ziming Mao, Yale University
Yao Yue, Pelikan Foundation
K. V. Rashmi, Carnegie Mellon University

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.

1 Introduction

Large-scale cache deployments enable the success of today's Internet. Companies have deployed software caches throughout various layers of the data center infrastructure: local and remote storage block I/O caches, in-memory and on-flash key-value caches. Caches are the key to fast data serving and consume a vast amount of resources. For example, Twitter reports that TBs of DRAMs are used for caching [104], and Netflix reports 10s of PBs of storage in use for caching [70].

The main driving force of cache deployments is the cache's ability to serve data with high throughput and low latency. Retrieving data from a cache (e.g., in DRAM) is thousands of times faster than retrieving it from the backend (e.g., in spinning disks). Because caches are often deployed on expensive storage media with limited capacity, the cache sizes are often much smaller than the dataset sizes. Thus, deciding what data to store in the cache is critical. A more *efficient* cache stores *more useful* data and serves more requests without hitting backend storage systems. Cache efficiency is often measured by hit ratio — the fraction of requests served from the cache (termed "hits"). When a cache is full, it uses an eviction algorithm to decide what data to keep and what to evict, and thus, the eviction algorithm is critical to cache efficiency.

Over the years, many eviction algorithms have been proposed to leverage different object features to make better eviction decisions. For example, several LRU variants [41–43,69,76,85] use diverse notions of recency to choose eviction candidates; some algorithms combine frequency and recency to score objects in different ways [4, 15, 26, 28, 56, 92]; others use a composition of frequency and object size [17,20]. Since different features acquire varying degrees of importance for different workloads, using a specific way to combine one or two object features typically only achieves high efficiency on some workloads (§4.5). Recently, several works have employed machine learning to improve cache evictions. We call these designs "*learned caches*".

We classify learned caches into three categories. First, *"object-level learning"*, such as LRB [87], learns the next access time for each object using dozens of object features and evicts the object with the furthest predicted request time. Second, *"learning-from-distribution"* models 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. Third, *"learning-from-simple-experts"*, such as LeCaR [92] and Cacheus [82], performs evictions by choosing eviction candidates recommended by experts (e.g., LRU and LFU), and updates experts' weights based on their past performance on the workload.

Because object-level learning, such as LRB, leverages more

Table 1: Comparison of different learned caches (numbers describe the example systems).

Learning approach	Example system	Learning granularity	Features for eviction	Storage overhead (bytes per object)	Potential efficiency	Throughput relative to FIFO
Object-level learning	LRB [87]	object	44	189	high	0.001-0.01
Learning-from-simple-experts	Cacheus [82]	expert	2	32	low	0.2-0.25
Learning-from-distribution	LHD [7]	workload	2	24	medium	0.2-0.25
Group-level learning (this paper)	GL-Cache	object group	7	<1	high	0.3-0.8

object features, learns the relative feature importance, and performs fine-grained learning on each cached object, it has the highest potential for achieving high efficiency. However, predicting and ranking objects at each eviction incurs significant computation and storage overheads as we observe LRB suffers from a 775× slow down compared to LRU. Learning-from-distribution has a lower computation and storage overhead because it models request probability using fewer features at a coarser granularity. However, it still has a lower throughput compared to simple heuristics (e.g., LRU) because it has to randomly sample and compare many objects at *each eviction*. Moreover, the existing design (e.g., LHD [7]) does not leverage object features other than age and size, limiting its potential for high efficiency. Lastly, the performance of learning-from-simple-experts, which learns the weights of experts, highly depends on the choice of the experts. Existing systems use simple experts and cannot leverage features not considered by the experts (§4). We show the comparisons of the three types of learned caches in Table 1 and discuss each of these categories more in-depth in §2.2.

To overcome the challenges in the existing approaches to leverage learning in caching, we propose learning at the level of object groups (which we call group-level learning). Group-level learning leverages multiple group-level features to learn object-group utility for evictions. It reduces the computation and storage overheads of learning by hundreds of times through amortization compared to learning at the object level. Furthermore, object groups accumulate more "signals" for learning and can leverage a variety of features for prediction, enabling better eviction decisions.

While group-level learning seems promising, it introduces several challenges: (1) How to group objects and perform evictions efficiently? (2) How to measure the usefulness of object groups (termed "utility") to determine the best eviction candidate? (3) How to learn and predict the object-group utility online?

We present **G**roup-level **L**earned **Cache** (GL-Cache) which leverages group-level learning by overcoming these challenges. GL-Cache clusters similar objects into groups using write time (§3.3) and evicts the least useful groups using a merge-based eviction (§3.6). GL-Cache introduces a group utility function (§3.4) to rank groups, which enables group-based eviction to achieve similar efficiency as object-based eviction (§4.2). GL-Cache uses a hybrid approach for eviction: it performs the heavyweight learning at the group level (thus amortizing the overheads) to identify the best groups to evict. And it leverages lightweight object-level metrics to retain a few highly useful objects from evicted groups. This two-level eviction enables GL-Cache to achieve a superior trade-off between learning overhead and cache efficiency.

We implemented GL-Cache in an open-source production cache and also developed a storage-oblivious implementation for running microbenchmarks. We compare GL-Cache with state-of-the-art designs on 118 production block I/O and CDN cache traces. Compared to object-level learning (LRB), group-level learning allows GL-Cache to achieve a 228× higher throughput on average. Moreover, GL-Cache achieves a slight improvement in hit ratio compared to LRB, with a 7% increase on average and 25% at P90 (10% of the traces) compared to LRB. Compared to the learned cache with the highest hit ratio, GL-Cache increases the hit ratio by 3% on average and 13% at the P90 tail, with a 64% higher throughput. Varying group sizes allow GL-Cache to change learning granularity, leading to a spectrum of algorithms. Along with two other system parameters, this spectrum enables users to navigate the trade-off between efficiency and throughput.

This paper makes the following contributions.

- We classify existing learned caches into three categories based on learning granularity and propose a new approach for learning in caching — group-level learning. Group-level learning amortizes overheads over objects in the group to achieve high throughput. By leveraging multiple group features and accumulating more training signals, group-level learning also achieves a high hit ratio.
- We design and implement GL-Cache, which overcomes the challenges of using group-level learning to achieve high cache efficiency with low-overhead learning. For the first time (to the best of our knowledge), a group-level utility function is defined and used for cache eviction.
- We evaluate GL-Cache using a diverse set of 118 production traces to illustrate and understand the high efficiency and high throughput of group-level learning.

2 Background and motivation

2.1 Software caches in data centers

Applications rely heavily on caching to speed up data access and increase system throughput. The two most important metrics of cache are efficiency measured using *hit ratio* and performance measured using *throughput*. Hit ratio is the fraction of requests fulfilled by the cache without fetching from the backend, and it measures the effectiveness of an eviction algorithm. A cache is more *efficient* if it achieves a higher hit ratio. Throughput measures the volume of requests a cache

can handle in a given duration. Higher throughput means serving the workload consumes less CPU resources and reduces expenses.

Over the years, many algorithms have been designed to improve cache hit ratio under different types of workloads [4,7,10,12,13,15,17,21,22,26,28,41–43,45,56,58,59, 68,69,76,79,82,85,87,92,98,103,109,110]. However, most of the algorithms make eviction decisions based on one or two object features, such as recency in LRU variants [43,76,85], and frequency in LFU variants [4,48], or a combination of two features [7,15,28,92]. However, cache workloads are often too complex to be captured by one or two features, and different features may acquire different importance across workloads. Furthermore, the feature importance can be different when the same workload is served at different cache sizes, as we show in §4.5. As a simplified example, assume a workload is composed of Zipf and repeated scans. When the cache size is very small, frequency is more important in selecting popular objects from the Zipf distribution. However, when the cache size is large enough to store both popular objects and repeated scans, recency may become more important in choosing objects to cache. In addition, prior works [10,87] reveal a large hit ratio gap between the state-of-the-art designs and the upper bound (e.g., Belady's algorithm [8] or flow-based offline optimal [11]), illustrating the possibility of improving the cache efficiency further.

2.2 Learning in caching

To make cache eviction algorithms adaptive across workloads, cache size, and over time, recent works have explored the idea of using machine learning in caching [7,10,29,82, 87,93,102]. These approaches can be broadly classified into three classes, which come with their pros and cons, as discussed below and summarized in Table 1.

2.2.1 Object-level learning

Object-level learning performs learning on each object. Multiple works have studied the prediction of object reuse distance [10, 14, 32, 63, 65, 86, 87, 99, 100] and popularity [19,31,71,107]. By predicting reuse distance, a learned cache can mimic Belady's algorithm [8], which evicts the object requested the furthest in the future using an oracle. However, predicting reuse distance is challenging [87] because an object's reuse distance is not only inherent to the object but is also affected by the access patterns of the workload. For example, the reuse distance will increase if a request-burst or scan happens between the two requests to the same object. Moreover, cache workloads often follow Zipf distributions [5,9,18,104]. Thus, most objects only get a limited number of requests. This leads to *limited object-level information for learning*. Meanwhile, it is these less popular objects that often affect cache efficiency [102]. As a result, existing works introduce approximations and proxies for learning reuse distance. For example, LRB [87] introduces Belady Boundary to reduce the range of reuse distance. While learning reuse distance is challenging, with careful feature engineering, large enough data, and a complex model, object-level learning may have the potential to achieve the highest hit ratio among all learned caches. However, object-level learning incurs prohibitively high storage and computation overheads.

Storage overhead. Both training and inference require extra storage. While the storage overhead of training data is often negligible with optimizations such as sampling and offloading to cheaper storage, inference data pose a significantly higher storage overhead. To make predictions on the object level, the cache needs to track features for each object. For example, LRB [87] stores 44 features (189 bytes) per object. Moreover, this large per-object metadata overhead is prohibitively high because it needs to reside in DRAM for frequent updates. Using fewer features is possible, but it leads to worse performance (§4).

Computation overhead. Both training and inference add computation overhead. While training data collection and frequent re-trainings consume CPU cycles, inference is the major source of computation overhead. The prediction in object-level learning uses dynamic features (e.g., object age), and the prediction results cannot be reused over time. Therefore, object-level learning needs to sample objects and perform inference at each write (eviction). For example, LRB samples 32 objects and copies their features to a matrix for inference for *each* eviction. In our measurement, each eviction (including feature copy, inference, and ranking) takes 200 μs on one CPU core, indicating that the cache can evict at most 5,000 objects on a single core per second. As a comparison, a production server achieves over 100,000 requests per second [75].

2.2.2 Learning-from-simple-experts

Several works use reinforcement learning to choose between multiple simple experts (eviction algorithms). For example, LeCaR [93] uses two experts (LRU and LFU). At each eviction, LeCaR chooses one expert to make an eviction decision based on the experts' weights. Similar designs can be found in ACME [2], FRD [80], and Cacheus [82], which use different experts and weight adjustment methods.

By using more than one algorithm for eviction, learning-from-simple-experts can adapt to changing access patterns. The overhead and efficiency of learning-from-simple-experts depend on the experts. Existing systems use simple experts and thus incur lower overhead than object-level learning. However, existing systems suffer from two problems. First, a delay exists between a bad eviction and an update on the expert's weight. The cache only discovers a bad prior eviction when the evicted object is requested again. This challenge, commonly known as "delayed rewards" in reinforcement learning [3,36,47,90], limits the efficiency of caches that use learning-from-simple-experts. Second, the cache efficiency is bounded by the experts selected; an efficient policy requires a good understanding of the workload. Learning-from-simple-experts cannot leverage features that the experts

do not consider. If a feature is important to the workload and not considered by any of the experts, then learning-from-simple-experts will not provide a high hit ratio. Some works used more experts [34] to capture more features. However, using more experts incurs higher overheads because it needs more computation and space to evaluate expert performance and update experts' weights.

2.2.3 Learning-from-distribution

The third type of learned cache models the request probability distribution and makes decisions based on the distribution. For example, LHD [7] uses the request probability distribution to calculate *hit density* (hits-per-space-consumed) as a metric for eviction. Specifically, LHD learns the request probability as a function of ages and then modulates it with size to arrive at hit density. LHD is simple yet effective and does not require expensive inference computation to compare objects. However, LHD's hit density is calculated based only on two features: age and size, and it is non-trivial to track probability with more features. Besides, LHD cannot change relative feature importance (how features are composed). Furthermore, because hit density does not change monotonically over time, LHD must sample objects to rank at each eviction, limiting its throughput due to slow random memory access.

Takeaways. We summarize the potential efficiency and overhead of the three types of learned caches in Table 1. We observe that object-level learning has a high potential to achieve high efficiency, but it incurs huge storage and computation overheads. Learning-from-distribution only considers a limited number of features and has lower overhead with lower potential for high efficiency. Although having a lower learning overhead, learning-from-distribution requires random sampling during *each* eviction, which limits its throughput. Learning-from-simple-experts highly depends on the experts used. Existing systems such as LeCaR and Cacheus achieve a higher hit ratio than a single expert but still leave a large hit ratio gap compared to other learned caches (§4.3).

3 GL-Cache: Group-level learned cache

To enable a better trade-off between learning granularity and learning overhead, we propose learning at the level of object groups (which we term "group-level learning"). The key idea behind group-level learning is to learn the usefulness of groups of objects (called "utility"). Based on this idea, we designed **G**roup-level **L**earned **C**ache (GL-Cache), which learns the object-group utility and evicts the least useful object groups. We first give a high-level overview of GL-Cache's design and then go into the details of each component.

3.1 Overview of GL-Cache

Fig. 1 shows an overview of GL-Cache. In GL-Cache, objects are clustered into fixed-size groups when writing to cache (§3.3). The training module in GL-Cache collects training data online and periodically trains a model to learn the utility of object groups (§3.5). The inference module predicts object-group utility and ranks object groups for eviction.

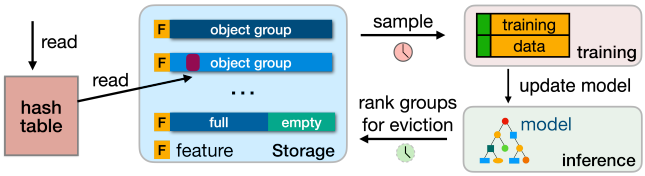


Fig. 1: Overview of GL-Cache. Objects are clustered into groups for learning: feature tracking, model training, and inference are performed on the group level.

Group-level learning requires group-level eviction: when the cache is full, object groups are evicted using a merge-based eviction which merges multiple groups into one, evicts most objects, and retains a small portion of popular objects (§3.6).

3.2 Group-level learning

Group-level learning has several advantages over existing learned caches:

Grouping amortizes overheads. Learning in caching incurs both computation and storage overheads. In group-level learning, these overheads are amortized over multiple objects in the group. In terms of storage, instead of adding huge per-object metadata, the metadata overhead is only added for each group. As a result, each object only incurs a tiny overhead on average (less than one byte in our implementation). The cost of inference computation is also amortized over objects. Compared to object-level learning, which performs one inference per eviction, each inference in group-level learning is used to evict a group of objects.

Grouping accumulates more signal. Many cache workloads follow a Zipf distribution [16, 104], and most of the objects receive very few requests. Because an object group has many objects, it often receives more requests than an individual object. More requests lead to more information on the group level compared to the object level, which makes it easier to learn and predict.

While group-level learning is promising, several challenges need to be addressed to leverage the power of learning:

- How to cluster objects into groups (§3.3)?
- How to compare the usefulness of object groups (§3.4)?
- How to learn the utility of object groups (§3.5)?
- How to perform evictions at group level (§3.6)?

While the ideas of grouping [105] and learning [87] have been studied independently in the context of caching, the combination of the two ideas in group-level learning leads to the unique challenges of *understanding, defining, and learning group utility*. We discuss these challenges and how GL-Cache overcomes them in this section.

3.3 Object groups

Using group-level learning, both learning and eviction are performed at the granularity of an object group, which usually contains tens to thousands of objects. Object grouping happens when an object enters the cache, and an object should not switch groups for two reasons. First, changing groups

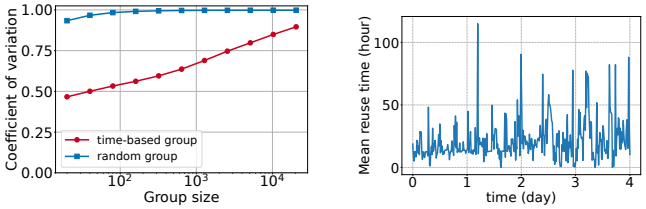


Fig. 2: a) Objects grouped using write time have more similar (smaller coefficient of variation) mean reuse time than objects grouped randomly. As group size increases, write-time-based grouping become closer to random grouping. b) Different object groups written at different times exhibit a large variation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

invalidates the learning pipeline. When an object is added to or removed from a group, the accumulated group information becomes stale and cannot be used for learning. Second, in implementation, changing groups often requires copying data on the storage device. Therefore, the grouping of an object is decided when entering the cache using simple static object features. Depending on workload types, such features include time, tenant id, content type, object size, etc. In this work, we focus on grouping based on write time, which is available in all systems and hence more generalizable.

Similar to observations made in prior works [82, 105], we observe that objects written at a similar time exhibit similar behaviors. Using traces from the evaluation, we measure the mean reuse time variation of objects in (1) write-time-based groups and (2) random groups. Fig. 2a plots the mean coefficient of variation (standard deviation over mean) of 100,000 groups for the two grouping methods at different group sizes. Compared to random groups, write-time-based groups aggregate objects with closer mean reuse time. Besides reuse time, we have similar observations on the frequency and the group utility defined below (not shown due to the space limit).

While objects *within* each write-time-based group have similar reuse, object groups created at different times exhibit dramatically different mean reuse times. Using a group size of 100 objects on the same trace, Fig. 2b shows that some groups exhibit more than 10× higher mean reuse time than others. These high-reuse-time groups are potentially good candidates for eviction. The two observations illustrate the feasibility of group-level learning using write-time-based grouping: objects inside groups are similar. Grouping by write time also allows an efficient implementation using a log-structured cache.

3.4 Utility of object groups

Identifying a good eviction candidate in object-based eviction has been well-studied. When object size is uniform, Belady [8] algorithm evicts the object that is requested the furthest in the future. When object size is not uniform, identifying the optimal candidate is NP-hard [11]. A common approximation is to evict the object that has the largest time till the next request over object size (called "size-aware Belady"). However, no metric exists that applies to object groups, and it is not trivial to adapt object-level metrics to the group level. In this section, we define an *object-group utility* function to measure object-group usefulness. A group with a lower utility is less useful and hence should be preferred for eviction. Because identifying the optimal *object* for eviction (when objects do not have the same size) can be reduced to identifying the optimal *group* for eviction, and the former is NP-hard [11], finding the optimal group for eviction is also NP-hard. Therefore, we define an empirical group utility that satisfies several properties.

3.4.1 Desired properties

(1) Because large objects occupy more space, the utility should consider object sizes. Groups composed of larger objects should have lower utilities.

(2) Similar to Belady, the utility should consider the time till the next access of objects in the group. A group of objects that are requested further in the future should have a lower utility. Importantly, the utility definition should properly handle objects with no future requests.

(3) When the group size is one object, group-level learning becomes object-level learning. In this case, ranking using the defined utility should produce the same result as Belady.

(4) The utility should be easy and accurate to track online. Calculating the ground truth (used for training) requires future information, but the cache cannot wait indefinitely to calculate it. This property requires that within a limited time horizon, the online tracked utility should be close to the utility calculated with complete future information. In other words, objects requested further in the future, including the ones with no future requests, should contribute less to the utility.

3.4.2 Utility definition

We observe that the cost of evicting one object is *always only one miss*. After a cache miss, the evicted object will be inserted into the cache. Meanwhile, 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. Therefore, similar to the cost-benefit analysis in LFS [83] and RAMCloud [77, 78], we define the utility of an object as its cost (one miss) over benefit (freed space multiplied by time till its next request).

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

Because GL-Cache evicts object groups, we further define the group utility as the sum of object utilities.

$$U_{group}(t) = \sum_{o \in \text{group}} \frac{1}{T_o(t) \times s_o} \tag{2}$$

The utility of a group measures the penalty of evicting the group or the benefit of keeping the group. Groups with lower utilities are thus better candidates for eviction. We remark that this is one definition of group utility that both satisfies the desired properties and performs well in our experience (§4). With this definition, we compare object-group utility and evict the group with the lowest utility. Since the true utility relies

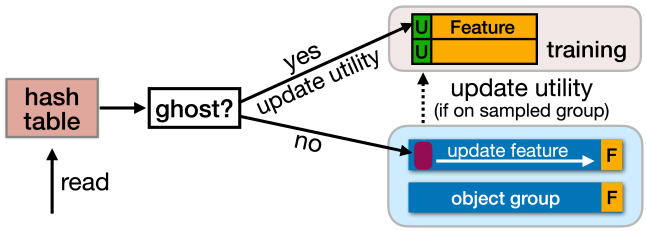


Fig. 3: The read flow in GL-Cache.

on the time till the next request and can only be calculated with future information, we design GL-Cache, which learns a model that can predict a group's utility based on its features.

3.5 Learning object-group utility in GL-Cache

GL-Cache learns a function $\mathcal{F}$ that calculates a group's utility given its features: $\mathcal{F}(X_{group}) = U_{group}$ where X_{group} is the features of an object group.

Object-group features. Features play a crucial role in learning [24, 38]. We consider two types of features in GL-Cache. The first type is *static features*, which includes request rate, write rate, miss ratio in the time window when the group was created (the write time of the first object), and mean object size. The second type is *dynamic features*, which includes age (in seconds), the number of requests, and the number of requested objects. Dynamic features increase over time. Static features do not change after creating a group and capture the workload and cache states (e.g., daily scan, request spike) during group creation time. We focus on these states because access pattern changes are often reflected in these metrics. For example, object groups created from scans are good candidates for evictions, and they often co-appear with increased request rates, write rates, and miss ratios. Compared to many of the existing works [87, 100], which mostly use dynamic features, GL-Cache uses far fewer dynamic features because tracking dynamic features is computationally expensive. We observe that adding more dynamic features only brings marginal hit ratio improvement, which does not justify the added computation overhead.

In total, GL-Cache uses seven features occupying 20 bytes for each group or 28 bytes if mean object size and creation time are not already tracked.

Learning model and objective function. GL-Cache uses gradient boosting machines (GBM) because tree models do not require feature normalization, and they have been shown to work well in previous works [10, 87] as well as many production environments [84, 96]. We formulate the learning task as a regression problem that minimizes the mean square loss (L2) of object-group utilities. We also explored the ranking objective function without observing a significant difference.

Training. GL-Cache trains a model using online collected training data, which consists of features and utilities of object groups. GL-Cache generates new training data by sampling cached object groups, and it copies the features of the sampled groups into a pre-allocated memory region. The utilities of the sampled groups are initialized to zero at the beginning and calculated over time. 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. GL-Cache then marks the object to ensure that it only contributes to the group utility *once*. It is possible that some objects may not be requested before training, and the online calculated group utility may be lower than the true utility. However, as mentioned in §3.4, these objects contribute marginally to the group utility due to their large reuse time.

In addition, a sampled group may be evicted before being used for training. Such evictions halt the tracking of group utility. Inspired by prior works [69, 82], GL-Cache keeps ghost entries for objects which have not been factored into group utility. A future request on the ghost entry will update the group utility, bringing it closer to the true utility.

Fig. 3 shows the read flow in GL-Cache. A successful hash table lookup may find two types of entries: a pointer to the object or a ghost entry. If it is a regular object, GL-Cache first updates the group features. Further, if the object is on a sampled group and has not contributed to the group utility, GL-Cache also updates the group utility before returning the data to the user. If it is a ghost entry, GL-Cache updates the corresponding utility and removes the ghost entry from the hash table, then returns a cache miss.

Given the access patterns change over time, the model needs to be retrained regularly. GL-Cache retrains the model every day (i.e., using wall clock time as a reference) because many real-world events that trigger requests repeat on a daily basis, such as cron jobs. In contrast, the other option of retraining every certain number of requests may cause the system to enter metastable failure [40] when an access pattern change increases the system load. Besides, GL-Cache chooses to retrain from scratch each time because tree models do not benefit from continuous training. Moreover, the inference overhead grows with training iterations because a new tree is added to the model in each iteration.

Inference. When GL-Cache needs to perform evictions, it predicts the utilities of all object groups and ranks them. GL-Cache uses the inference/ranking result for multiple evictions, which reduces the frequency of inference and thus the computation overhead. We denote *eviction fraction* $F_{eviction}$ as the fraction of ranked groups to evict using one inference. That is, GL-Cache performs an inference every $F_{eviction} \times N_{group}$ groups where N_{group} is the total number of groups. In our evaluation, $N_{ranked-group}$ is the total number of groups, but we remark that one can also sample some groups for inference if the total number of groups is too large. Also, the groups are evicted over time on demand rather than all at once, and neither training nor inference need to be on the critical path of request serving. In summary, GL-Cache only needs to perform $\frac{1}{F_{eviction}}$ inferences to write a full cache of objects.

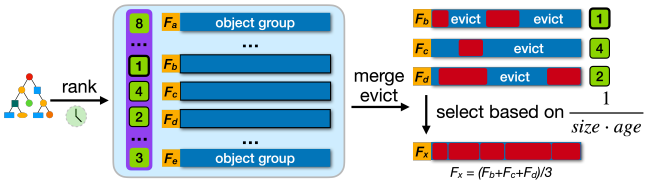


Fig. 4: Object group utility prediction and merge-based group eviction in GL-Cache.

3.6 Evictions of object groups

Learning at the object-group level introduces an interesting challenge to cache eviction: unlike most caches which evict one object each time, GL-Cache evicts a group of objects. Although evicting object groups leads to lower overhead due to batching and amortization, it may evict objects that are still popular. GL-Cache optimizes the group eviction by using a merge-based eviction, similar to Segcache [105]. 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*. The merge process retains S_{group} objects from the merged groups and evicts all other objects. The retained objects form a new group, and the original N_{merge} groups are evicted. This is the only time that an object changes its group membership in GL-Cache. Unlike group selection, which uses ranking, object selection uses a simple metric based on object age and size: $\frac{1}{size \cdot age}$ where *age* is the time since the last access. We choose to use this metric because recency and size are the two most common metrics used in other eviction algorithms (§2). GL-Cache performs the heavyweight online learning at the group level to identify the best groups to evict. It leverages lightweight object-level metrics to retain a few highly useful objects. This two-level eviction approach enables GL-Cache to achieve a superior tradeoff between learning overhead and cache efficiency.

In summary, each eviction evicts N_{merge} groups of objects and retains one group of objects, as illustrated in Fig. 4. The features (except mean object size) of the merge-produced group take the mean values of the N_{merge} merged groups. Note that only the first object group is picked based on the group utility; the next $N_{merge} - 1$ object groups are chosen as ones with write time close to the first group. This ensures that objects in the new group after a merge-based eviction are still close in write time and similar. In contrast, objects from the N_{merge} least useful groups may not be similar. Clustering similar objects into groups is critical for effective group-level learning. In our experience, merging the N_{merge} least useful groups shows lower efficiency with up to 20% decrease in hit ratio. Compared to evicting one object each time, group-based eviction evicts more objects than needed at each eviction, which may reduce the efficiency upper bound group-level learning can achieve. However, we show in §4.2 that evicting object groups can achieve hit ratios very close to Belady, indicating that group eviction will not be the bottleneck for cache efficiency.

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

Table 3: Three sets of 128 traces were used in the evaluation.

Dataset	# traces	# requests (millions)	Source
CloudPhysics [94]	103	2115	VM disk I/O
MSR [73]	14	410	Disk I/O
Wikimedia [87]	1	2804	CDN requests

3.7 A spectrum of GL-Cache

GL-Cache has three parameters in its design (Table 2): the size of each object group S_{group}, the number of object groups to merge at each eviction N_{merge}, and how many groups are evicted using one inference which is determined by $F_{eviction}$. Varying these parameters leads to a spectrum of algorithms for optimizing hit ratio and throughput. A larger S_{group} reduces learning granularity; a larger N_{merge} retains fewer objects; and a larger $F_{eviction}$ reduces the ranking frequency. Each of these changes reduces the computation overhead with a potential hit ratio drop. Therefore, GL-Cache allows the users to navigate the trade-off between cache efficiency and throughput. For scenarios that are more sensitive to overheads, such as local cache deployments, GL-Cache can provide higher throughput with a slightly lower hit ratio, and vice versa. In §4.6, we show that these parameters generalize well across workloads.

4 Evaluation

In this section, we evaluate GL-Cache to answer the following questions.

- Will group-based eviction limit the efficiency upper bound when compared to object-based eviction (§4.2)?
- Can GL-Cache improve hit ratio and efficiency over other learned caches (§4.3)?
- Can GL-Cache meet production-level throughput requirements and how much overhead does GL-Cache add (§4.4)?
- How does GL-Cache improve efficiency without compromising throughput (§4.5)?

4.1 Experiment methodology

Prototype system. GL-Cache groups objects using write time and can be efficiently implemented using a log-structured cache. Hence, we implement GL-Cache on top of Segcache [105], an open-source production in-memory cache that uses segment-structured (log-structured) storage. We map an object group in GL-Cache to a "segment" in Segcache and replace FIFO with the learned model. We use the XGBoost [1] library to implement our GBM models and use the default values for all parameters. GL-Cache has three parameters (Table 2). In our evaluation, GL-Cache uses 1 MB group size, merges five groups at each eviction, and evicts 5% of ranked groups after each inference. We compare GL-Cache with Segcache [105], a segment-structured cache used

by Twitter; Cachelib [9], Meta's production cache library, which uses slab storage and a throughput-optimized LRU for eviction; TinyLFU [28], implemented within Cachelib by Meta engineers. We have also implemented LHD [7] on top of Pelikan's slab storage [81].

Micro-implementation. In addition to the prototype system, we build a storage-oblivious implementation of GL-Cache in C on top of libCacheSim [101] to compare different eviction algorithms. Our implementation mimics Memcached's design but has neither a networking stack nor object value storage, and we call it micro-implementation. Compared to the prototype, the micro-implementation only performs eviction-related metadata operations and does not consider storage layout or system overheads such as fragmentation. We use two sets of parameters (Table 2) to demonstrate the spectrum of GL-Cache. The first demonstrates a better *efficiency* and uses $S_{group} = 60$ objects, $N_{merge} = 2$ groups, $F_{eviction} = 0.02$. We call this system GL-Cache-E. The second demonstrates a higher *throughput* using $S_{group} = 200$ objects, $N_{merge} = 5$ groups and $F_{eviction} = 0.1$, and we call it *GL-Cache-T*. We remark that the parameters are not tuned per workload. Thus GL-Cache may provide better performance (hit ratio or throughput) with workload-specific fine-tuning.

Besides GL-Cache, we implement Cacheus [82] in C following the authors' open-source Python implementation. For LHD [7] and LRB [87], our micro-implementation used code open-sourced by the authors. We use default parameters except for changing the LRB optimization target from byte miss ratio to object miss ratio (implemented by LRB's author). Besides state-of-the-art designs, we have also implemented FIFO, LRU, and size-aware Belady [11].

GL-Cache trains the first model after running one day of workload (using timestamps from the traces). Before a model is trained, it uses FIFO to perform evictions, GL-Cache then trains the model once a day from scratch, which has little overhead as discussed in §3.5.

Workloads. We use a wide variety of traces representing a diverse set of workloads from three dataset sources (Table 3). The CloudPhysics [94] dataset includes 103 block I/O traces with different CPU/DRAM configurations and access patterns. Each trace records the I/O requests from a VM for around one week. Because 86% of the VMs had DRAM sizes between 1 GB and 16 GB with a median of 3880 MB, we performed evaluations at 1 GB, 4GB, and 16 GB cache sizes. We present only 1 GB and 16 GB for space reasons. We have also evaluated GL-Cache using 14 block I/O traces (we ignore the traces which contain fewer than 5 million requests) from Microsoft Research Cambridge (MSR) [73]. Because the working set sizes of MSR traces exhibit a very wide range, we set cache sizes for each trace at 0.01%, 0.1%, and 1% of each trace's footprint (size of all objects). Besides block I/O request traces, we have also evaluated GL-Cache with the Wikimedia CDN trace used in previous works such as LRB [87] and LFO [10]. All the workload traces have at least three fields: the timestamp, id, and size of the requests.

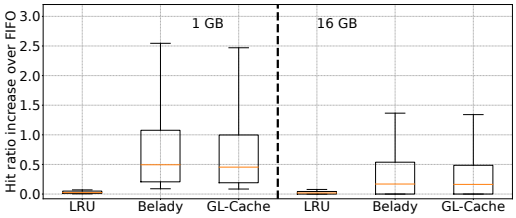


Fig. 5: With oracle assistance, group eviction can achieve a similar hit ratio improvement as object eviction.

We ran micro-implementation experiments on the Cloudlab [25] Utah site using m510 nodes with Intel Xeon D-1548 CPU, 64GB ECC DDR4 DRAM. And we ran prototype experiments on the Cloudlab Clemson site using c6420 nodes with Intel Xeon Gold 6142 CPU and 384 GB of DRAM.

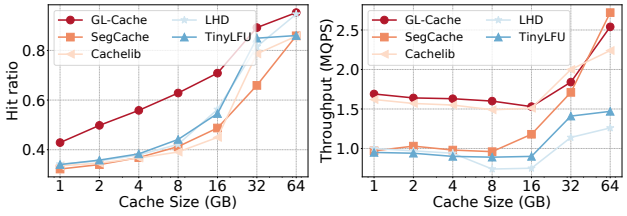
Metrics. We replayed traces by reading and writing to a local cache in a closed loop and measured hit ratio and throughput. Because all traces are week-long traces, we started measurements after finishing the first three days' requests to make sure the cache is properly warmed up under all the configurations considered. We present evaluations using a one-day warmup time in §4.6, which shows that the observations remain the same as with a three-day warmup.

We report aggregated results from 103 CloudPhysics traces and 14 MSR traces using box plots for the micro-implementation results. Due to the diversity of the workloads, both hit ratio and throughput have wide ranges. Hence, for ease of visual presentation, we report results compared to FIFO using the following two metrics: *hit ratio increase over FIFO* defined as $\frac{HR_{alg} - HR_{FIFO}}{HR_{FIFO}}$ where HR stands for hit ratio; throughput relative to FIFO defined as $\frac{R_{alg}}{R_{FIFO}}$ where R is the throughput. The box plots have the following format: the orange line inside the box is the median, the box shows 25 and 75 percentiles, and the whiskers show 10 and 90 percentiles. Because several other factors in the prototype systems (e.g., storage layout) affect efficiency and throughput, for ease of understanding, we focus our evaluation on the micro-implementation results. We present raw hit ratio and throughput numbers using the prototype systems for one representative trace in §4.3 and §4.4.

4.2 Group-based eviction

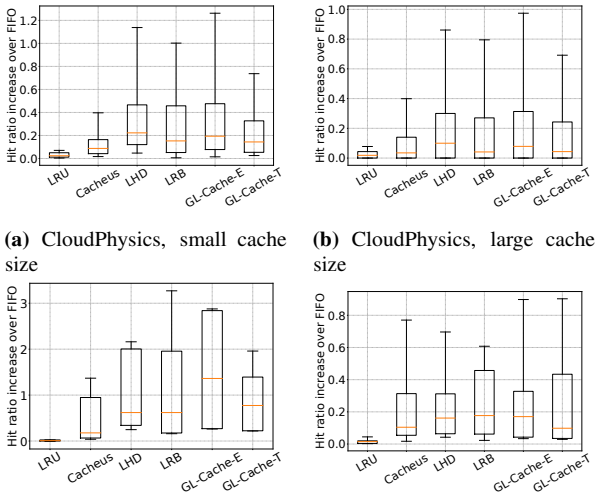
Group-level learning evicts most objects in the selected groups. The bulk eviction may limit the efficiency of group-level learning. To understand the limitation of group eviction, we compare oracle-assisted group eviction with oracle-assisted object eviction (size-aware Belady [11]). The oracle-assisted group eviction uses the same design as GL-Cache except using future request time to calculate group utility and retain objects. Size-aware Belady evicts the object that has the largest $(T_{next} - T_{now}) \times s_o$ where T_{next} is the time of the next request, and s_o is the object size.

We compare these two approaches using CloudPhysics



(a) Hit ratio **(b)** Throughput

Fig. 6: Prototype evaluation of a CloudPhysics trace.



(a) CloudPhysics, small cache size **(b)** CloudPhysics, large cache size

(c) MSR, small cache size **(d)** MSR, large cache size

Fig. 7: Hit ratio increase over FIFO. GL-Cache runs under two modes, GL-Cache-E is the efficient mode, GL-Cache-T is the throughput mode.

traces. Fig. 5 shows that group-based eviction can achieve a hit ratio similar to object-based eviction at both small and large cache sizes. The similar hit ratios suggest that *group eviction will not become the bottleneck for achieving high efficiency*. While the algorithms in this comparison use oracle information, in the following sections, we show how GL-Cache can use learning to replace the oracle and achieve high cache efficiency.

4.3 Cache efficiency

We compare the efficiency of GL-Cache with state-of-the-art designs in both the prototype and the micro-implementation. Fig. 6a shows hit ratios for the prototype running one CloudPhysics trace at different sizes. Compared to other systems, GL-Cache consistently achieves the best efficiency, providing a significant hit ratio increase (up to 40%) over the best of all baselines. Compared to Segcache, which uses the same storage layout with FIFO-based group eviction, group-level learning increases the hit ratio by 60% at 8 GB. Cachelib uses a throughput-optimized LRU and has the lowest hit ratio among all the baselines. LHD and TinyLFU use two object features to make eviction decisions: LHD models hit density based on age and size; TinyLFU uses frequency to filter out unpopular objects and uses recency to evict objects. Leveraging more than one feature to choose eviction candidates allows LHD and TinyLFU to achieve higher hit ratios. However, not using more features puts an upper bound on their potential. In comparison, GL-Cache evicts groups based on seven features covering recency, frequency, cache, and workload states at group creation time (miss ratio, write rate, request rate). Considering multiple features in conjunction with learned importance allows GL-Cache to make better eviction decisions and achieves a higher hit ratio. Evaluations on the other traces show similar results.

To compare with more algorithms and on more traces, we show hit ratio results from the micro-implementation on CloudPhysics and MSR traces in Fig. 7. Because of the wide range of hit ratios across traces, we show the relative hit ratio increase compared to FIFO instead of the raw hit ratios. We observe that both LRU and Cacheus improve FIFO's hit ratio, but only by a single-digit percentage for the median workload on both datasets. Meanwhile, LRB, LHD, and GL-Cache increase FIFO's hit ratio more prominently.

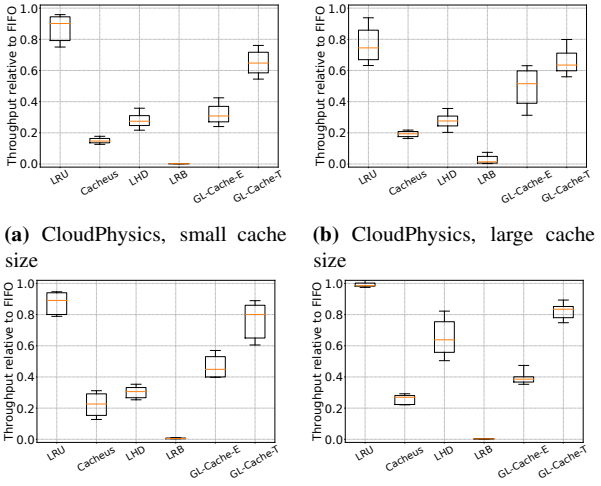
Among LRB, LHD, and GL-Cache-E, LRB has the smallest observed hit ratio improvement. We conjecture that learning at the object level receives limited information on each object since cache workloads often follow Zipf distributions, and thus is more challenging to learn compared to learning at the group level. Compared to LHD, we observe that GL-Cache-E shows similar efficiency on CloudPhysics traces. However, on MSR traces, GL-Cache-E is more efficient than LHD with a 60% hit ratio increase for a median workload at the small size. This observation suggests that leveraging more features to make eviction decisions can be very useful for some workloads at certain cache configurations.

Compared to GL-Cache-E, GL-Cache-T trades hit ratio for higher throughput (§4.4). However, we observe that GL-Cache-T's efficiency is still on-par with LRB. Overall, we observe that GL-Cache improves the hit ratio by up to 37.8% compared to LHD and 87% compared to LRB (not shown in the figure). While LRB uses more features/information than other eviction algorithms, it does not always provide the highest hit ratio. More information leads to higher efficiency only when the information is useful and well-utilized. We conjecture that perhaps not all the features in LRB are useful, and the model may not be making the best use of the features.

When comparing prototype and micro-implementation results, we observe that the hit ratio difference also depends on the storage design. GL-Cache uses log-structured storage, and the difference between prototype and micro-implementation is smaller (<10%); LHD uses slab storage, and sometimes the prototype can have a significantly lower hit ratio (>20%) compared to the micro-implementation. This large difference comes from fragmentation and slab calcification problems [39, 105]. However, we did not find a way to efficiently implement LHD on top of log-structured storage because it requires the storage to have the capability of evicting (removing) any cached object, while log-structured storage can only

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



(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.

efficiently support sequential write and removal.

Besides block I/O cache traces, we have also evaluated GL-Cache using the Wikimedia CDN trace from LRB evaluations. Table 4 shows that learning helps LRB to achieve miss ratios 35% to 70% lower than FIFO. Compared to LRB, GL-Cache-E further reduces the miss ratio by up to 16%. In summary, the evaluations on three datasets totaled 118 traces illustrating the high efficiency and generality of group-level learning.

4.4 Throughput and overheads

Not only does GL-Cache achieve a high hit ratio, but GL-Cache also achieves high throughput. Fig. 6b shows the throughput of GL-Cache in the prototype. We observe that compared to production systems (Cachelib, Segcache), GL-Cache achieves a similar throughput, indicating that GL-Cache meets the throughput requirement of a production system. Moreover, compared with eviction algorithms such as LHD and TinyLFU, GL-Cache is 2-3× faster.

Besides the prototype evaluation, Fig. 8 compares the throughput of GL-Cache with several state-of-the-art algorithms evaluated on *all* CloudPhysics and MSR traces. While LRU achieves throughput close to FIFO, all advanced eviction algorithms exhibit a significant slowdown compared to FIFO. However, among all learned caches, GL-Cache is significantly faster than others. Compared to LRB, GL-Cache-E has a 228× higher throughput, and GL-Cache-T has a 586× higher throughput on average at the small cache size. Compared to the *fastest* of all learned caches, GL-Cache-E is on average 64% faster, and GL-Cache-T is on average 3× faster at the small cache size. Similarly, on the Wikimedia trace (Table 4), GL-Cache-E is tens to hundreds of times faster than LRB and achieves almost half of FIFO's throughput.

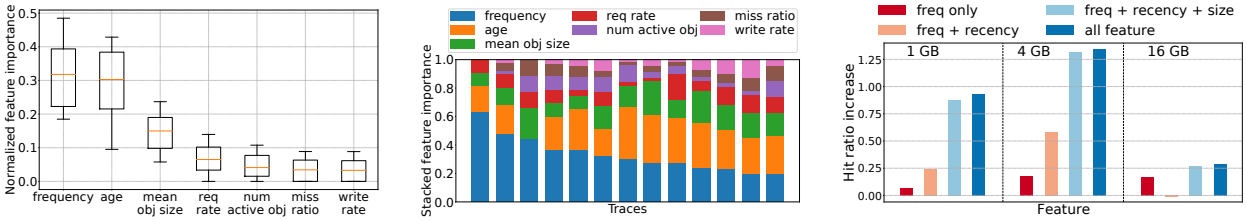
GL-Cache achieves high throughput because it needs very few metadata updates on cache hits and misses. On a cache hit, GL-Cache only needs to update the last access time and group utility if it is on a sampled group (§3.5). On a cache miss, GL-Cache does not need to update any metadata most of the time; occasionally, it performs a group eviction and evicts 100s to 1000s of objects. In contrast, other systems must update multiple metadata entries on both cache hits and cache misses. For example, TinyLFU needs to maintain the frequency counting sketch and the LRU chain; LHD needs to sample 32 objects, thus having 32 random DRAM accesses for each eviction. Segcache is simpler than GL-Cache in per-request operations. However, the lower hit ratio of Segcache leads to its reduced throughput because of more evictions.

The second reason for GL-Cache's high throughput is that the overheads of training and inference are amortized. Because GL-Cache uses fewer features to learn simpler high-level patterns instead of per-object access patterns, it uses a simple model and is only retrained once a day. In our measurement, each training consumes 10 - 50 ms of one CPU core (not amortized by the number of training samples). In addition, each inference consumes 0.4 - 3 ms of one CPU core and is triggered every time 5% of ranked groups are evicted. Because each inference evicts many groups and each eviction evicts many objects, the inference computation is amortized. The amortization is the key reason for GL-Cache's high throughput compared to other learned caches. Moreover, although training and inference are not on the critical path of request serving, our throughput evaluation measures run time including both training and inference.

While throughput evaluations show the low computation overhead of GL-Cache, machine learning in caching also introduces storage overhead. First, GL-Cache uses DRAM to store 8000 training samples. The training data storage is pre-allocated and small (256 KB) compared to the cache size (GBs). For deployments with very limited memory, the training data can also be stored on the storage device. Second, each object group in GL-Cache uses 28 bytes of features — each object thus adds less than one byte. Besides the group-level features, GL-Cache tracks each object's last access time using 4 bytes. In total, GL-Cache uses 5 bytes of object metadata for eviction. As a comparison, LRU requires two pointers with 16 bytes of metadata per object, and LRB uses 192 bytes of features per object.

4.5 Understanding GL-Cache's efficiency

So far we have demonstrated that GL-Cache has a higher miss ratio and throughput than existing systems. While amortized overhead explains the high throughput, this section ex-



(a) Feature importance across traces **(b)** Feature breakdown across example traces. **(c)** Feature utility case study.

Fig. 9: Feature case study.

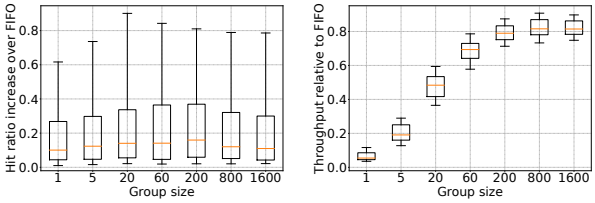
plores how learning helps GL-Cache achieve high efficiency.

Most eviction algorithms use one or two object features to decide which object to evict. For example, LRU evicts the object with the largest access age (recency), LeCaR and Hyperbolic [15] use recency and frequency to make eviction decisions, LHD relies on access age and object size to choose eviction candidates. In contrast, object-level learned cache such as LRB uses 44 features covering different measurements of recency and frequency, as well as object size, to compare objects. Similarly, GL-Cache uses seven features to compare object groups. To better understand GL-Cache's efficiency, we examine how GL-Cache uses these features.

We obtained the feature importance score directly from XGBoost. The importance score is calculated using the number of times a feature is used to split the data across all trees and may not represent the ground truth. Fig. 9a shows the normalized feature importance scores of different features across traces obtained from the models trained for each trace. We observe that across traces, frequency and age have relatively high scores with medians of around 0.3. This aligns well with existing literature on eviction algorithms, which mostly use recency and frequency to make eviction decisions. The next important feature is the mean object size, which is essential for algorithms that consider variable-size objects. Besides these features, the workload and cache states (request rate, miss ratio, write rate) at the group creation time have similar scores with a median of around 0.05. When summed up, they have a similar importance as the object size.

While we observe that the most commonly used features (recency, frequency, size) are critical, we also observe that no feature is dominant across all traces. Fig. 9b shows the feature importance score for 12 randomly selected traces. For some traces, frequency is more important, with an importance score of 0.6. For others, recency or size is more important. GL-Cache weighing features differently across traces suggests that GL-Cache can effectively adapt the feature importance to each workload. For comparison, the algorithms leveraging more than one feature often combine the features in a way that cannot adapt to workloads. For example, Hyperbolic scores an object using $\frac{frequency}{age}$, leaving the *relative* importance of frequency and age unchanged across workloads.

Fig. 9c uses one trace to illustrate the importance of GL-Cache *adaptively* using multiple features. It shows how gradually including more features improves the hit ratio. We



(a) Hit ratio **(b)** Throughput

Fig. 10: Impact of group size.

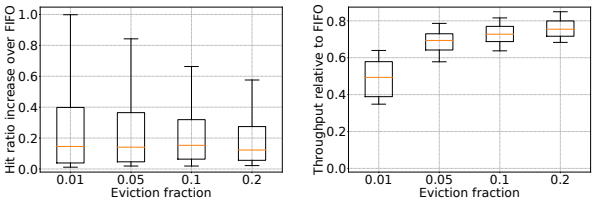
observe that the combination of frequency, recency, and size at small sizes (1 GB and 4 GB) leads to a large hit ratio increase (e.g., 80% at 1 GB). Meanwhile, frequency alone is insufficient and can only increase the hit ratio by 10% at 1 GB. Using all features increases the hit ratio modestly on this trace compared to only using frequency, age, and size. Moreover, Fig. 9c shows that feature importance could change with cache sizes. Object size is more important at 1 GB cache size, while frequency becomes more important than other features at 16 GB. This could be because small objects contribute more hits per consumed byte than large objects, so caching small objects is better when the cache size is small. Meanwhile, when most small objects are cached at a larger cache size, choosing between large objects depends on request frequency. This observation suggests that in GL-Cache, the choice and use of features adapt not only to the workloads but also to different configurations such as cache sizes.

In summary, learning at the group level can leverage multiple features to adapt to both workload and cache sizes, enabling higher cache efficiency.

4.6 Sensitivity analysis

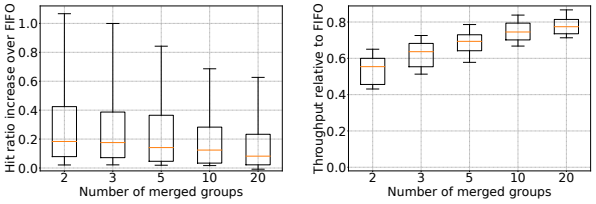
We have discussed the three parameters used by GL-Cache in §3.7, and we have shown the two modes of GL-Cache: one achieves higher efficiency (GL-Cache-E), and the other achieves higher throughput (GL-Cache-T). This section shows in detail how these parameters affect hit ratio and throughput. In addition, we show that the warmup time does not significantly change the hit ratios.

Group size. A smaller group indicates a finer granularity for learning and evictions. Varying group size affects both throughput and efficiency. First, reducing group size increases storage and computation overhead due to finer learning granularity. As a result, throughput increases with group size, as shown in Fig. 10. Second, the hit ratio increases when



(a) Hit ratio **(b)** Throughput

Fig. 11: Impact of eviction fraction $F_{eviction}$.



(a) Hit ratio **(b)** Throughput

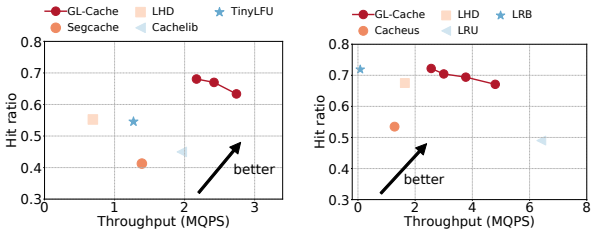
Fig. 12: Impact of the number of groups to merge at each eviction.

the group size increases from 1 (object-level learning) to 20, then decreases as the group size further increases from 60 to 1600. A smaller group indicates that each eviction evicts fewer objects, enabling a higher hit ratio. However, when the group size is too small, each group gets too few requests for group feature learning to be effective, thus decreasing the hit ratio. The non-monotonic hit ratio change (hit ratio first increases then decreases) also explains why object-level learning achieves a lower hit ratio than GL-Cache.

Eviction Fraction. GL-Cache evicts $F_{eviction}$ fraction of ranked groups between each inference to reduce computation overhead and better tolerate inaccurate predictions. The more groups (larger $F_{eviction}$) evicted per inference, the fewer inferences, thus higher throughput. However, a larger $F_{eviction}$ means more (useful) groups are evicted after each inference, resulting in a lower hit ratio. Fig. 11 shows that increasing $F_{eviction}$ reduces hit ratio and increases throughput.

Number of groups to merge. The last tunable parameter in GL-Cache is the number of groups to merge at each eviction. Because GL-Cache evicts the majority of the objects on the N_{merge} groups and retains one group worth of objects, merging more groups means that GL-Cache retains fewer objects from each group. Retaining fewer objects reduces the computation needed at each eviction, but it also reduces efficiency. Fig. 12 shows that increasing the number of merged groups increases throughput and reduces the hit ratio.

Besides the above three parameters, the learning component also introduces several parameters such as training data size and retraining frequency. GL-Cache retrains the model once a day because many events (such as cron jobs and diurnal patterns) happen on a daily basis. Wall clock time sometimes is more important than virtual time (reference count), and has also been recognized by researchers from Google when they use neural networks to predict the lifetime of a memory allocation [64]. The retraining interval affects both efficiency and performance. Note that more frequent retraining does



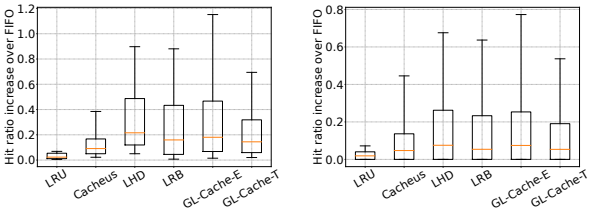
(a) Prototype evaluation **(b)** Micro-implementation

Fig. 13: A spectrum of GL-Caches allow users to tradeoff between hit ratio and throughput.

not always lead to a higher hit ratio because shorter retraining intervals reduce the accuracy of the group utilities used for training as they are accumulated over time. We observe that the best retraining interval depends on the workload — some workloads show higher hit ratios with half-day retraining, and some others benefit from two-day retraining. While fine-tuning retraining intervals can improve the hit ratio by up to 10%, one-day retraining achieves a good performance across workloads as shown. Besides training frequency, another parameter in training is the number of training samples. Because GL-Cache learns high-level access patterns, which we conjecture is easier to learn than per-object behavior, GL-Cache does not require a large amount of training data. While we cannot prove that 8000 training samples are sufficient for all workloads under all scenarios, we find that it is sufficient for the diverse traces in our evaluation.

The sensitivity analysis shows that GL-Cache is relatively robust to parameter changes. The parameters of GL-Cache-E and GL-Cache-T were chosen based on evaluations of 10 random traces. Our results show that these two sets of parameters work well across the diverse traces in the evaluation. However, like in any other system, a general set of parameters provides reasonable performance but does not guarantee the best performance. Per-workload fine-tuning can potentially provide larger benefits. GL-Cache provides the opportunity for users to explore the trade-off between efficiency and throughput. Fig. 13 shows the throughput and hit ratio of GL-Cache compared to baselines (we do not plot multiple close-by points of GL-Cache for clarity). In both prototype and micro-implementation evaluations, GL-Cache achieves higher throughput than systems with a similar hit ratio or a higher hit ratio than systems with a similar throughput. Deployments with less computation power can use GL-Cache in a high-throughput mode with a slightly lower hit ratio. And deployments that are less sensitive to computation may use GL-Cache to achieve a higher hit ratio.

Our evaluation so far used a warmup time of three days to make sure the cache is warmed up for any trace under any size. We have also evaluated with a one-day warmup time and presented the results in Fig. 14. We observe that although the absolute values exhibit some differences, the overall trends on hit ratio increase are similar when compared to using a three-day warmup time (Fig. 7). In addition to the hit ratio



(a) Small cache size **(b)** Large cache size

Fig. 14: Using one-day warmup, evaluated on CloudPhysics traces.

results, throughput results using a one-day warmup are also similar to that of a three-day warmup. Similarly, evaluations on the MSR and Wikimedia traces also exhibit little difference between using one-day and three-day warmup times.

5 Related work

The study of cache designs has a long history with the majority of works focusing on improving cache efficiency. With increasing complexity in cache management, many recent works have also improved the throughput and scalability.

Better eviction algorithms. Most works improving cache efficiency focus on cache eviction algorithms, especially how to define and use recency, frequency, and size to make better eviction decisions. For example, ARC [69] uses two LRU lists to balance between recency and frequency; CAR [6], LIRS [43, 44, 57, 108], Clock-pro [42], 2Q [85], SLRU [41], LRU-K [76] use a different metric to measure recency; variants of LFU [4, 48], LRFU [56], tinyLFU [26–28] and hyperbolic [15] use a combination of frequency and recency to make evictions; various greedy-dual algorithms [17,20,45,59] use two metrics (e.g., frequency and size) to choose eviction candidates. In addition, several learned caches have been designed in the past few years, as discussed in detail in §2.2. Compared to existing learned caches, GL-Cache employs group-level learning, which amortizes overheads and accumulates stronger learning signals to make better eviction decisions. Moreover, existing learning approaches to caching cannot be directly applied to group-level learning due to challenges such as comparing object groups' usefulness.

Improve cache throughput. Most algorithms that improve efficiency trade throughput for higher efficiency. With increasing complexity in cache systems, throughput and scalability become critical. MICA [62] uses a holistic design with a lossy hash table and partitioned log-structured DRAM storage to achieve high throughput and scalability; Segcache [105] uses an approximate-TTL-indexed segment-chain with batched eviction to achieve high throughput and scalability; MemC3 [30] uses a cuckoo hash table and Clock eviction to improve scalability; Cachelib [9] reduces LRU promotion frequency to improve scalability. These systems often use weaker eviction algorithms such as FIFO, Clock, or weak LRU. Compared to these works, GL-Cache improves efficiency without sacrificing throughput. Specifically, GL-Cache and Segcache share some design aspects such as object grouping. However, Segcache primarily innovates on the design of storage layout for key-value caches, and it uses FIFO for eviction. Instead, GL-Cache focuses on using learning for evictions, which is the key to GL-Cache's efficiency gains.

Use of machine learning to improve system efficiency. Machine learning has seen increasing use to improve system efficiency. For example, Google uses machine learning to improve the efficiency of data center operations [33]. Microsoft uses machine learning to improve database query optimizer [46]. Prior works have designed learned components to replace various parts of a system, such as index [23, 50, 54, 55, 74, 97] and query optimizer [66, 67] in databases, straggler mitigation in inference systems [52, 53], and FTL for SSD [89]. Moreover, many other works look into automatic database tuning using machine learning [60, 91]. In caching, in addition to the three categories of learned cache evictions that we have discussed in §2, recent works have also looked into using sub-sampling to reduce learned cache's time horizon [95], using machine learning to predict memory access [37], designing cache admission [35, 51], designing cache prefetching [61, 61, 88, 102] predicting hot records in LSM-Tree storage [106], using deep recurrent neural network models for content caching [72], using Markov cache model for size-aware cache admission policy [13]. Compared to these works, GL-Cache is the first system to perform learning on a group of entities and navigates efficiency-throughput trade-off using coarse-grained learning granularity.

6 Conclusion

We propose a new approach for using machine learning to improve cache efficiency: group-level learning. Group-level learning predicts and evicts the least useful object groups. Group-level learning leverages multiple object-group features to adapt to workload and cache size, accumulates stronger signals for learning, and amortizes learning overheads over objects. As a result, it makes better eviction decisions with a tiny overhead. We build GL-Cache in a production cache to demonstrate group-level learning and evaluate it on 118 production block I/O and CDN traces. GL-Cache achieves a significantly higher throughput as compared to all other learned caches while retaining a higher hit ratio. Thus, GL-Cache paves the way for the adoption of learned caches in production systems.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. In addition, our shepherd Carl Waldspurger has given very valuable feedback that improved the writing of this paper. This work was supported in part by a Facebook Ph.D. fellowship, and in part by NSF grants CNS 1901410 and CNS 1956271. The processing and evaluations were performed on Cloudlab [25] and ChameleonCloud [49].

Availability

We open-source our implementation at `https://github.com/Thesys-lab/fast23-glcache`.

References

[1] Xgboost. https://github.com/dmlc/xgboost. Accessed: 2022-09-06.

[2] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Acme: Adaptive caching using multiple experts. In *WDAS*, volume 2, pages 143–158, 2002.

[3] Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

[4] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 193–206. Springer, 1998.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[6] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd : Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[8] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[9] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[10] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140, 2018.

[11] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2:1–38, 2018.

[12] Daniel S. Berger, Sebastian Henningsen, Florin Ciucu, and Jens B. Schmitt. Maximizing cache hit ratios by variance reduction. *SIGMETRICS Perform. Eval. Rev.*, 43:57–59, sep 2015.

[13] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017.

[14] Adit Bhardwaj and Vaishnav Janardhan. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, volume 32, 2018.

[15] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[16] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.

[17] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, page 18, USA, 1997. USENIX Association.

[18] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[19] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. Learn to cache: Machine learning for network edge caching in the big data era. *IEEE Wireless Communications*, 25(3):28–35, 2018.

[20] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[21] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[22] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[23] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[24] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[26] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[27] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, page 94–106. ACM, Nov 2018.

[28] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[29] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[30] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[31] Qilin Fan, Jian Li, Xiuhua Li, Qiang He, Shu Fu, and Sen Wang. Pa-cache: Learning-based popularity-aware content caching in edge networks. *arXiv preprint arXiv:2002.08805*, 2020.

[32] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. Feedforward neural networks for caching: N enough or too much? *SIGMETRICS Perform. Eval. Rev.*, 46:139–142, jan 2019.

[33] Jim Gao. Machine learning applications for data center optimization. `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf`, 2014. Accessed: 2022-04-06.

[34] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, page 1489–1496, Cambridge, MA, USA, 2002. MIT Press.

[35] Yu Guan, Xinggong Zhang, and Zongming Guo. Caca: Learning-based content-aware cache admission for video content in edge caching. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 456–464, 2019.

[36] Beining Han, Zhizhou Ren, Zuofan Wu, Yuan Zhou, and Jian Peng. Off-policy reinforcement learning with delayed rewards, 2021.

[37] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pages 1919–1928. PMLR, 2018.

[38] Jeff Heaton. An empirical analysis of feature engineering for predictive modeling. In *SoutheastCon 2016*, pages 1–6. IEEE, 2016.

[39] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[40] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 22), pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[41] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.

[42] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[43] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[44] Song Jiang and Xiaodong Zhang. Making lru friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance. *IEEE Transactions on Computers*, 54(8):939–952, 2005.

[45] Shudong Jin and A. Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261, 2000.

[46] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. Microlearner: A fine-grained learning optimizer for big data workloads at microsoft. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2423–2434. IEEE, 2021.

[47] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[48] George Karakostas and D Serpanos. Practical lfu implementation for web caching. *Technical Report TR-622–00*, 2000.

[49] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[50] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014*, 2019.

[51] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. Rl-cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.

[52] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. Learning-based coded computation. *IEEE Journal on Selected Areas in Information Theory (JSAIT special issue on deep learning: mathematical foundations and applications to information science)*, 1(1):227–236, 2020.

[54] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[55] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[56] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[57] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 59–64, 2018.

[58] Cong Li. CLOCK-pro+: improving CLOCK-pro cache replacement with utility-driven adaptation. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 1–7, Haifa Israel, May 2019. ACM.

[59] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[60] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[61] Zhenmin Li, Zhifeng Chen, Sudarshan M Srinivasan, Yuanyuan Zhou, et al. C-miner: Mining block correlations in storage systems. In *FAST*, volume 4, pages 173–186, 2004.

[62] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica : A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[63] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning*, page 6237–6247. PMLR, Nov 2020.

[64] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[65] Ayush Mangal, Jitesh Jain, Keerat Kaur Guliani, and Omkar Bhalerao. Deap cache: Deep eviction admission and prefetching for cache. *arXiv preprint arXiv:2009.09206*, 2020.

[66] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814*, 2020.

[67] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[68] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 243–262, 2021.

[69] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, volume 3, pages 115–130, 2003.

[70] Sailesh Mukil, Prudhviraj Karumanchi, Tharanga Gamaethige, and Shashi Madappa. Cache warming: Leveraging ebs for moving petabytes of data. `https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3`. Accessed: 2022-09-16.

[71] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, NetAI'18, page 48–53, New York, NY, USA, 2018. Association for Computing Machinery.

[72] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 48–53, 2018.

[73] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage*, 4(3), November 2008.

[74] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 985–1000, 2020.

[75] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[76] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[77] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[78] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[79] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. predis: Penalty and locality aware memory allocation in redis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 193–205, 2019.

[80] Sejin Park and Chanik Park. Frd: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[81] Pelikan. `https://github.com/twitter/pelikan`. Accessed: 2022-09-06.

[82] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with cacheus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[83] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[84] Amazon SageMaker. Xgboost algorithm. `https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html`. Accessed: 2022-09-06.

[85] D Shasha and T Johnson. 2q: A low overhead high performance buffer management replacement algoritm. In *Proc. 20th Int. Conf. Very Large Databases*, pages 439–450, 1994.

[86] Wanxin Shi, Qing Li, Chao Wang, Gengbiao Shen, Weichao Li, Yu Wu, and Yong Jiang. Leap: learning-based smart edge with caching and prefetching for adaptive video streaming. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019.

[87] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[88] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[89] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. *arXiv preprint arXiv:2301.00072*, 2022.

[90] Richard S Sutton. Introduction: The challenge of reinforcement learning. In *Reinforcement Learning*, pages 1–3. Springer, 1992.

[91] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[92] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[93] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[94] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[95] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning caching policies with subsampling. In *NeurIPS Machine Learning for Systems Workshop*, 2019.

[96] Joseph Wang, Anne Holler, Mingshi Wang, and Michael Mui. Productionizing distributed xgboost to train deep tree models with large data sets at uber. `https://eng.uber.com/productionizing-distributed-xgboost/`. Accessed: 2022-04-06.

[97] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[98] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323. USENIX Association, February 2021.

[99] Nan Wu and Pengcheng Li. Phoebe: Reuse-aware online caching with reinforcement learning for emerging storage models. *arXiv preprint arXiv:2011.07160*, 2020.

[100] Gang Yan and Jian Li. Rl-bélády: A unified learning framework for content caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1009–1017, 2020.

[101] Juncheng Yang. libcachesim. `https://github.com/1a1a11a/libCacheSim`. Accessed: 2022-12-16.

[102] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.

[103] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1159–1177, Renton, WA, April 2022. USENIX Association.

[104] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[105] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[106] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: a learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13(12):1976–1989, 2020.

[107] Yuchao Zhang, Pengmiao Li, Zhili Zhang, Bo Bai, Gong Zhang, Wendong Wang, Bo Lian, and Ke Xu. Autosight: Distributed edge caching in short video network. *IEEE Network*, 34:194–199, May 2020.

[108] Chen Zhong, Xingsheng Zhao, and Song Jiang. Lirs2: an improved lirs replacement algorithm. In *SYSTOR'21: Proceedings of the 14th ACM International Conference on Systems and Storage*, 2021.

[109] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.

[110] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.