3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches

Wenbin Zhou, *Beijing University of Technology;* Zhixiong Niu and Yongqiang Xiong, *Microsoft Research;* Juan Fang and Qian Wang, *Beijing University of Technology*

https://www.usenix.org/conference/fast25/presentation/zhou-wenbin

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings of the 23rd USENIX Conference on File and Storage Technologies is sponsored by

3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches

Wenbin Zhou[1], *Zhixiong Niu*[2], *Yongqiang Xiong*[2], *Juan Fang*[1], *Qian Wang*[1] *
[1]*Beijing University of Technology;* [2]*Microsoft Research*

Abstract

Caches can effectively reduce request latency and network traffic, with the eviction policy serving as a core component. The effectiveness of an eviction policy is measured by both the byte miss ratio and the object miss ratio. To reduce these miss ratios, various learning-based policies have been proposed. However, the substantial computation overhead introduced by learning limits their deployment in production systems.

This work presents **3L**-Cache, an object-level learning policy with **L**ow computation overhead, while achieving the **L**owest object miss ratio and the **L**owest byte miss ratio among learning-based policies. To reduce overhead, we introduce two key advancements. First, we propose an efficient training data collection scheme that filters out unnecessary historical cache requests and dynamically adjusts the training frequency without compromising accuracy. Second, we design a low-overhead eviction method that integrates a bidirectional sampling policy to prioritize unpopular objects and an efficient eviction strategy to effectively select evicted objects. Furthermore, we incorporate a parameter auto-tuning method to enhance adaptability across traces.

We evaluate 3L-Cache in a testbed using 4855 traces. The results show that 3L-Cache reduces the average CPU overhead by 60.9% compared to HALP and by 94.9% compared to LRB. Additionally, 3L-Cache incurs only 6.4× the average overhead of LRU for small cache sizes and 3.4× for large cache sizes, while achieving the best byte miss ratio or object miss ratio among twelve state-of-the-art policies.

1 Introduction

Caches play a pivotal role in modern production environments, significantly reducing network traffic [37, 64, 65, 81] and accelerating response times for client requests [22, 30, 60, 80]. For instance, in 2020, Netflix's demand accounted for 1% of the total bandwidth across transatlantic cables, with 95% of its traffic served directly from caches [5]. At the heart of any caching system lies its eviction policy, which determines which data remains stored and which is removed to free up space in the constrained cache. The performance of this policy is typically measured using two key metrics: (1) object miss ratio, which indicates the proportion of requests resulting in cache misses, and (2) byte miss ratio, reflecting the volume of data missed by the cache. A lower object miss ratio leads to faster processing of requests, while a reduced byte miss ratio translates to fewer data requests to the origin server, thereby decreasing overall network traffic.

Over the years, progressing from heuristic eviction policies [14, 15, 32, 45, 53, 56, 67, 84, 87] to modern learning-based policies [20, 37, 39, 50, 62, 64, 65, 73, 79, 80], cache miss ratios have been reduced. However, the average computation overhead of learning-based policies can be up to 172.5× higher than that of LRU (Table 1). In real-world cache applications, such as CDNs, clusters of numerous servers are deployed worldwide [1, 4, 82]. Replacing the deployed LRU with learning-based policies would result in a hundredfold increase in deployment costs, rendering such a transition economically prohibitive [35, 85]. Therefore, this work explores the possibility of reducing computation overhead to the same magnitude as heuristic policies, while retaining the advantages of learning policies in terms of miss ratios.

The learning-based policies are divided into three categories. *Policy-level learning* (e.g., LeCaR [73], CACHEUS [62]) uses reinforcement learning to directly make eviction decision. Its miss ratio is significantly impacted by the delayed adaptation to workload changes inherent to reinforcement learning. *Group-level learning* (e.g., GL-Cache [80]) clusters similar objects into groups and performs learning and eviction at the group level substantially reducing computation overhead. However, the category performs well on a limited number of traces due to the limitations of grouping methods and group utility definitions. *Object-level learning* (e.g., LRB [64], HALP [65], Raven [37], Parrot [50]) performs training and prediction on each object. By predicting reuse distance, this type of policy can mimic Belady's MIN algorithm [19], which makes "optimal decisions" by evicting the object re-

*Corresponding author: Qian Wang, wangqian2020@bjut.edu.cn

quested the furthest in the future among all objects in the cache. These categories are described in detail in §6.2.

Although object-level learning has the potential to achieve the lowest miss ratio among all learning policies, its fine-grained learning incurs the highest computation overhead. Our analysis reveals that the overhead from training and prediction account for 70% of the total overhead, with their proportions varying depending on cache size. We explore opportunities to reduce the overhead of these two components (§3), but three challenges have to be addressed:

(1) How to reduce training overhead waste without compromising model accuracy? Online training is necessary for a cache due to the constantly changing workloads. We observe there are tremendous wastes of computation resources in training. Reducing the training frequency to a certain extent only slightly affects miss ratios but significantly reduces the computation overhead. However, finding an appropriate frequency is challenging because object features vary at all times and they are different across traces.

(2) How to reduce prediction overhead without sacrificing miss ratios? We define "the eviction ratio" as the proportion of objects that can be evicted among the predicted eviction candidates. A higher eviction ratio indicates lower prediction overhead. Our analysis reveals considerable potential to increase this ratio. However, increasing the ratio without impacting miss ratios is challenging, as existing random sampling policies cannot ensure a sufficient number of unpopular objects among the eviction candidates.

(3) How to improve the generalizability to traces? The complexity and variability of cache access patterns in production traces pose significant challenges. The key lies in designing a solution that enhances generalizability while maintaining low computational overhead, ensuring consistently good performance across a wide range of traces.

We address these challenges with 3L-Cache, an object-level learning policy. **First**, we propose a training data collection scheme that filters out unnecessary historical cache requests, enabling dynamic adjustment of training frequency without compromising accuracy (§4.2). **Second**, we propose an object eviction method to significantly reduce the prediction overhead (§4.3). The method includes a bidirectional sampling policy to increases the ratio of unpopular objects among eviction candidates; an efficient object eviction policy determines the eviction ratio and efficiently identifies evicted objects from a large number of eviction candidates. **Third**, we design a parameter auto-tuning module finding appropriate values of parameters for each trace with low overhead (§4.4).

We implement 3L-Cache[1] and compare its performance with eleven state-of-the-art policies using 4855 traces from eight public production datasets. The policies include six heuristic and five learning-based policies, in which LRB and HALP are object-level learning policies. In evaluation, 3L-Cache reduces the average CPU overhead by 60.9% compared to HALP and by 94.9% compared to LRB. Meanwhile, 3L-Cache incurs only 6.4× the average overhead of LRU for small cache sizes and 3.4× for large cache sizes.

This paper makes the following contributions.

- For the first time (to the best of our knowledge), representative cache eviction policies (including heuristic and learning-based) are evaluated from three metrics: byte miss ratio, object miss ratio, and computation overhead.
- We find that object-level learning policies have considerable potential to reduce computational overhead without compromising the miss ratio.
- We design and implement 3L-Cache, which overcomes the challenges of reducing the training and prediction overhead, and even further lowers the byte and object miss ratio compared to state-of-the-art policies.
- We evaluate and analyze 3L-Cache on diverse production datasets of 4855 traces. Moreover, we verify that 3L-Cache can be applied to other object-level learning policies to enhance performance, such as reducing LRB's CPU overhead by 80% and its byte miss ratio by 0.6%.

2 Background and Motivation

While most caching systems[3, 9, 11, 12] deploy basic FIFO and LRU policies, we believe modern eviction policies should be designed using machine learning due to the reduction in miss ratio it offers. We first analyse the performance of three categories of learning-based policies (§2.1), and then illustrate their deployment bottlenecks (§2.2).

2.1 Performance Analysis

Currently, no learning-based policy is evaluated across all three dimensions: object miss ratio, byte miss ratio and CPU overhead. We evaluate five representative learning-based policies from three categories using 4855 production traces from eight open-source datasets (Table 3). Since cache sizes influence the performance of policies to a great extent, all metrics are evaluated under two representative sizes: 0.1% and 10% of the footprint (size of all objects in the trace) [84, 87]. Table 1 presents the evaluation results, where CPU overhead is defined as (CPUload × time).

We observe no policy performs well on all 3 metrics. For example, the object-level learning policy LRB has the lowest byte miss ratio and a relatively low object miss ratio, but its mean CPU overhead is up to 172× higher than LRU under small cache size. Additionally, the object miss ratio of the group-level learning policy GL-Cache are the lowest, but its byte miss ratio is the worst, even higher than LRU, because it focuses more on minimizing the object miss ratio at the expense of the byte miss ratio.

[1]The source code is available at https://github.com/optiq-lab/3L-Cache

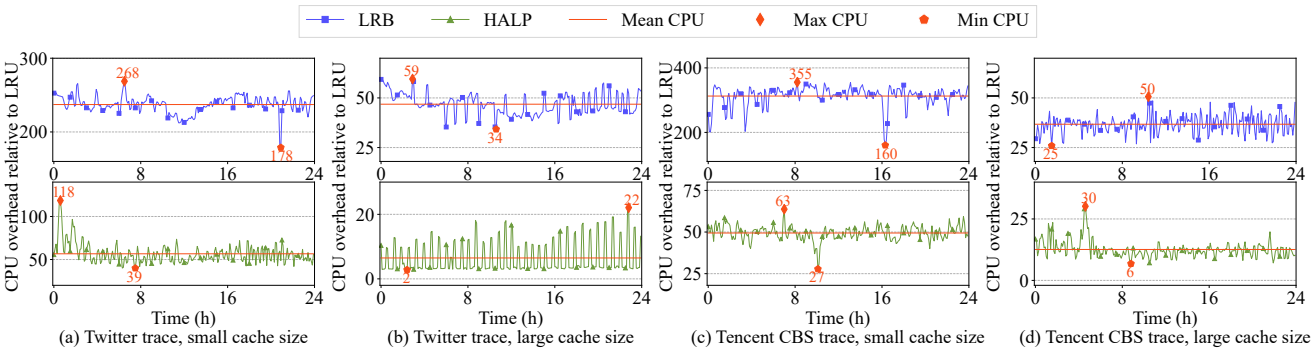


Figure 1: Real-time CPU overheads of two typical object-level learning policies under different cache sizes.

Table 1: Performance comparison. Miss ratio is the reduction from LRU, and CPU overhead is the times relative to LRU. S/L represents small and large cache size, respectively.

Category	Policy	Mean byte miss ratio (S/L)	Mean object miss ratio (S/L)	Mean CPU overhead (S/L)
Policy-level	LeCaR	0.029/0.024	**0.049/0.029**	1.190/1.209
	CACHEUS	0.051/0.024	0.081/0.031	2.289/2.122
Group-level	GL-Cache	**-0.088/-0.376**	**0.063/0.096**	1.991/1.288
Object-level	LRB	**0.068/0.071**	0.060/-0.018	**172.467/58.247**
	HALP	0.055/0.044	0.066/0.026	22.980/10.297

2.2 Deployment bottlenecks

The deployment of policy-level learning and group-level learning policies is hard due to their high miss ratios. For example, the byte miss ratio of GL-Cache is even higher than that of LRU, and LeCaR exists the highest object miss ratio among all learned policies, as shown in Table 1. However, for the existing object-level learning policies that perform well in both miss ratios, its primary deployment bottleneck is the CPU overhead, detailed as follows.

Table 1 shows mean CPU overheads of two representative object-level learning policies, which are far higher than LRU (∼172× and ∼58× on LRB, ∼23× and ∼10× on HALP, under small and large cache size, respectively). These increased overheads significantly raise deployment costs. Additionally, Figure 1 shows the real-time CPU overhead of the policies on a web trace from Twitter (cluster 17) with over 9 billion requests, and a block trace from Tencent CBS (ns 1351) with 195 million requests. We observe: (1) regardless of the request volume and cached data type, the peak CPU overhead of these two policies is over 10× higher than LRU, and the peak CPU overhead of LRB even reaches 300×; (2) the real-time CPU overhead fluctuates "frequently and greatly" for any traces and cache sizes, with the difference between the max and min values being about 1.5×. Therefore, if CPU resources are deployed based on average overheads, they cannot meet the constantly arriving peak demands.

In conclusion, it is hard to deploy learning-based policies efficiently and economically in production systems.

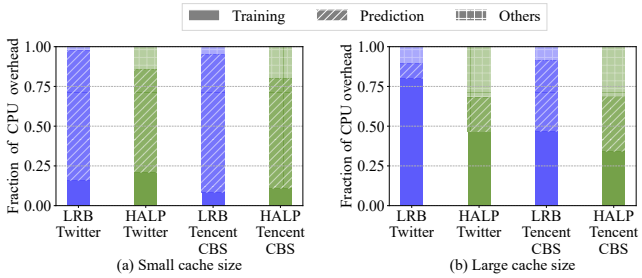


Figure 2: The proportion of overhead for each component.

3 Design Space and Challenges

Building on the above analysis, we explore opportunities to significantly reduce CPU overhead of object-level learning policies, which perform well in terms of both miss ratios. Figure 2 shows the proportion of CPU overhead for each component of LRB and HALP at different cache sizes. We observe: 1) the majority of computation resources for this category are consumed by training and prediction; 2) the proportions of training and prediction overhead vary with cache size. When the cache is small, the miss ratio is high, requiring frequent predictions to evict objects, leading to prediction becoming the main overhead. However, as the cache size increases, the miss ratio decreases, and the frequency of predictions decreases, while the training frequency remains largely unchanged. Consequently, training gradually becomes the main overhead.

There exists tremendous wastes of computation resources in model training. Figure 3 shows the changes in miss ratio [2] and CPU overhead as training frequency varies. We find that as the training frequency decreases, the CPU overhead prominently reduces while the miss ratio remains almost unchanged. However, once the frequency is reduced to a certain level, the reduction in CPU overhead slows, and the miss ratio begins to increase noticeable. For example, the training frequency of LRB is originally 2^3 times per million requests. When the frequency is reduced from 2^3 to 2^{-3}, the

[2]The miss ratio refers to byte miss ratio. Changes in the object miss ratio are similar but are omitted due to space limitations.

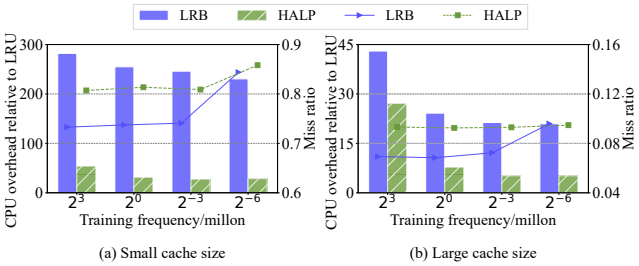


Figure 3: The miss ratio and CPU overhead at different training frequencies on the Tencent CBS trace (ns1351). The line represents miss ratio. The bar represents CPU overhead.

CPU overhead of LRB decreases by 48% with negligible changes in the miss ratio. In contrast, further reducing the training frequency to 2^{-6} results in a 38% increase in the miss ratio, as shown in Figure 3b.

There is considerable potential to reduce prediction overhead in eviction. The prediction overhead of object-level learning policies arises from selecting objects for eviction from the cache. We define **the eviction ratio** as the fraction of evicted objects to the predicted eviction candidates. For example, LRB randomly samples 64 cached objects for prediction and evicts only 1 object based on the prediction results during each eviction, while HALP samples 4 objects for inference and evicts 1 object, resulting in eviction ratios of 1.56% and 25%, respectively. A higher eviction ratio corresponds to lower prediction overhead. For LRB and HALP, their eviction ratio can reach up to 72%, as large amounts of production trace data indicate that 72% of cached objects receive only one request at a cache size of 10% of the trace footprint [84].

3.1 Challenges

There is potential to reduce the CPU overhead of object-level learning policies while maintaining their advantage in miss ratios. However, we must address the following challenges:

C-1. How to reduce overhead waste in training without reducing model accuracy? There exists an appropriate training frequency in online training. Above this frequency, the CPU overhead increases sharply, but the miss ratio only decreases slightly. Below this frequency, the miss ratio significantly increases. However, finding the frequency is challenging because object features constantly change and vary across traces.

C-2. How to reduce the prediction overhead without sacrificing miss ratios? Directly increasing the eviction ratio can reduce computation overhead, but it will inevitably sacrifice miss ratios, because existing random sampling policies cannot ensure that enough unpopular objects are among the eviction candidates. This presents two challenges:

C-2.1. How to sample unpopular objects? In the cache queue, newly arrived objects (whose popularity follows the Zipf distribution [26, 31, 34, 38, 66, 68]) and objects that are requested again are mixed at the tail. Additionally, the popularity of objects changes over time. This makes it difficult to know the position of unpopular objects in the queue, increasing the difficulty of sampling them as eviction candidates. How can we sample in a way that increases the ratio of unpopular objects among eviction candidates?

C-2.2. How to choose evicted objects from candidates? The time-varying nature of object popularity makes it impossible for any sampling policy to ensure that all eviction candidates are unpopular. Moreover, the ratio of unpopular objects varies for each set of sampled candidates. Therefore, determining an appropriate eviction ratio (which dictates the number of evicted objects) and selecting the specific objects to evict, in order to minimize the eviction of popular objects and reduce CPU overhead, remains a challenge.

C-3. How to improve generalizability across traces? Cache access patterns are complex and diverse in production traces. For example, block access in storage system workloads often mixes scan and LRU-friendly patterns, whereas in-memory caching workloads rarely exhibit scan requests [83]. Therefore, the challenge is how to design a low-overhead auto-tuning method, can find the appropriate parameter values for each trace, ensuring low miss ratios across most traces.

4 The 3L-Cache Design

4.1 Overview

To address the above challenges, we design 3L-Cache, which consists of four main components, as shown in Figure 4. (1) Training data collection includes a coarse-grained adjustment method to efficiently determine an appropriate sliding window size for storing historical cache requests, and labeling rules to exclude unnecessary requests from the training data (§4.2.1). (2) Bidirectional sampling policy targets unpopular objects as eviction candidates from the cache queue with complex popularity distributions (§4.3.1). (3) Efficient object eviction includes determining the number of evicted objects by setting the eviction ratio (referred to as the eviction count), and efficiently identifying the object with the furthest next request time from a large number of eviction candidates for eviction (§4.3.2). (4) Auto-tuning parameter efficiently determines appropriate parameter values for each trace (§4.4).

Figure 4 illustrates the workflow of 3L-Cache. Newly arrived cache requests are incorporated into the sliding window as historical information. The Gradient Boosting Machine (GBM) model [41] is updated after every M labeled training entries. When the cache lacks sufficient space for a newly arrived object, 3L-Cache first checks whether the eviction count is 0. If it is, the bidirectional sampling policy is activated, and the count is updated. If the count is not 0, the object with the furthest next request time is evicted, and the count is decremented. This process continues until there is enough space for the new object. Additionally, the auto-tuning module adjusts the parameter values of other modules when triggers occur.

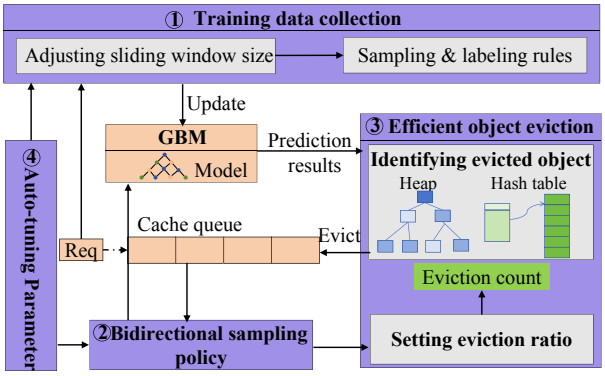


Figure 4: The overview of 3L-Cache.

4.2 Online Training

To address the challenge of setting the appropriate training frequency, we propose a training data collection scheme that filters out unnecessary historical cache requests, thereby reducing training frequency without compromising accuracy. Additionally, we provide a brief overview of the GBM model.

4.2.1 Training data collection

It faces two problems: (1) How to efficiently determine the appropriate size of sliding window for storing historical request data? (2) How to avoid sampling unnecessary requests into the training data?

Sliding window size. An appropriate size is critical to model accuracy, as a small window may provide insufficient historical information, while a large window risks incorporating outdated data. Additionally, efficiently determining the window size is essential to avoid excessive computational overhead. For example, LRB performs dozens or even hundreds of tests to identify an appropriate size, with each test using a portion of the trace's validation prefix (e.g., the initial 20% of the trace data).

Therefore, we propose a coarse-grained sliding window adjustment method. The window size is dynamically adjusted to ensure that the number of objects it contains equals h_{sw} multiplied by the number of objects in the cache queue, where h_{sw} is a positive integer variable. This method enables rapid determination of an appropriate sliding window size, as each time h_{sw} is incremented by 1, the window size increases by an amount equal to the current cache queue size.

Different traces and cache sizes require different window sizes due to varying demands for historical information. For instance, compared to a large cache, a small cache primarily stores popular objects, and to accurately evict the few unpopular objects, more information is needed for precise predictions. Thus, h_{sw} is automatically adjusted, as detailed in §4.4.

Sampling and labeling training data. Eviction decisions primarily target unpopular objects, thus our sampling is object-oriented to prevent training data from being biased towards popular objects, which are frequently requested within the sliding window. We sample one object for each request and record the sampling time. The sampling rules are as follows: (1) To ensure a rich source of training data, we randomly sample and record one object within the sliding window. (2) If a previously sampled object is selected again before it receives a new arrived request, its information will not be recorded again. This approach filters out unnecessary data while ensuring that the training data encompasses a wider variety of object access patterns.

We label the sampled data to generate training data. When a sampled object is requested again, the interval until its next arrival is determined and used as the label for training. When a sampled object exits the window due to a prolonged lack of requests, we label it with its waiting time plus the longest waiting time recorded among the training samples. This distinction ensures a clear separation between objects that are eventually accessed and those that are not.

After every M labeled entries, the model is retrained and updated [3]. The value of M affects the accuracy of the model. If it is too large, the model cannot be updated in time, while if it is too small, there will be insufficient training information, making it difficult to train an accurate model. We evaluated several values on 4855 traces, including 16K, 32K, 64K, 128K, 256K and 512K. The results indicate that the policy performs well across most traces when M is within the range of [32K, 128K]. In this paper, we set M=64K.

4.2.2 GBM Model

GBM is chosen to predict the log(next-arrival-time-interval) for an object due to its ability to efficiently handle missing historical features without requiring costly imputation, unlike SVMs [36, 72], Random Forests [16, 24], and MLPs [70]. Furthermore, GBM has been proven to perform well in prior works [37, 64, 75]. 3L-Cache uses mean square loss, and six input features: age, size, frequency and the three most recent inter-arrival times (filled with ∞ if the number of object requests is insufficient). "Age" is the time elapsed since the latest request. "Frequency" represents the number of times an object is requested within the sliding window. "Inter-arrival time" is the time interval between consecutive requests for an object. This feature captures the arrival process of objects, which is crucial for predicting their next arrival times. Experimental results verify that these six features are sufficient to ensure model accuracy.

In addition, since 3L-Cache uses only six features for eviction, it incurs an average storage overhead of 67 bytes per object. For example, in the Wikipedia CDN dataset, which contains 126 million objects and 8.4 billion requests, this overhead accounts for approximately 2.5% of the cache space for small cache sizes and around 0.4% for large cache sizes.

[3] Since model training requires a substantial amount of data, we initially employ LRU until the model is trained.

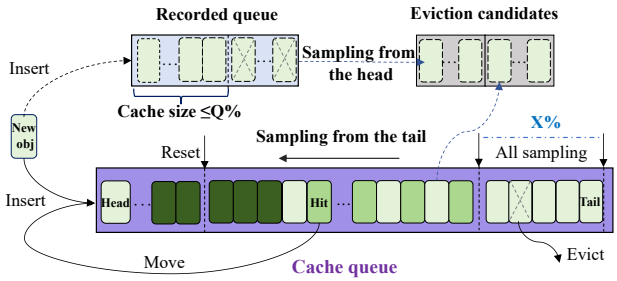


Figure 5: An illustration of bidirectional sampling policy.

4.3 Cached Object Eviction

Recall from §3.1, in order to achieve low prediction overhead in eviction process without affecting the miss ratio, we need to answer two questions: (1) How to increase the ratio of unpopular objects among the sampled eviction candidates? (§4.3.1) (2) How to evict objects from candidates,including how many to select and which ones to evict? (§4.3.2)

4.3.1 Bidirectional sampling policy

The proposed policy consists of the following steps: (1) Sampling from the tail, which targets unpopular objects that have remained in the queue for a period. (2) Sampling from the head, designed to capture unpopular objects among those that have newly arrived, as illustrated in Figure 5. When no eviction candidates are available, each of these two sampling methods is performed once.

Sampling from the tail. This is because newly arrived objects or objects that are requested again (referred to as hit objects) are inserted at the head, positioning unpopular objects at the tail of the queue (consistent with LRU behavior). Sampling proceeds with n objects at a time until the total number of objects scanned by the pointer equals the initial queue length. Afterwards, the sampling pointer is reset to the tail. This process, referred to as one round of sampling, is repeated continuously. The sampling rules are as follows:

In each round of sampling, starting from the tail of the queue, all objects in the first $x\%$ of the queue are sampled, while the remaining objects are sampled only if their access frequency is less than or equal to f. This rule is based on the distribution of popularity in the cache queue: unpopular objects are densely distributed at the tail, while the remaining part has a mix of popular and unpopular objects. If the sampling round ends but the number of sampled objects remains less than n, the pointer resets to the tail, and the parameters x, f, n are updated in preparation for the next round. The auto-tuning method for these three parameters is described in §4.4.

Sampling from the head. It focuses on sampling newly arrived objects, referred to as those that enter the cache by displacing previously cached objects. Since newly arrived objects and hit objects are intermixed at the head of the queue, we use an additional queue to record the IDs of newly arrived objects (referred to as the recorded queue). This facilitates the rapid identification of these objects within the cache queue.

Before each sampling, it is necessary to assess whether the cache space occupied by the objects in the recorded queue exceeds $Q\%$. Establishing this space allows newly arrived objects to accumulate feature information, enhancing the accuracy of predictions and leading to better eviction decisions.

During sampling, priority is given to objects that have resided in the cache the longest. These objects are removed from the recorded queue until the total space occupied by the objects in the recorded queue falls below $Q\%$, or the number of sampled objects exceeds m. Sampling concludes once either condition is satisfied. The rationale is twofold: First, $Q\%$ regulates the proportion of newly arrived objects in the cache. Since only a small fraction of newly arrived objects are popular (following a Zipf distribution) [17, 25, 26], failing to control this proportion could result in the eviction of popular objects. Second, in real-world traces, evicting a single large object via "sampling from the tail" can trigger a substantial influx of newly arrived objects. This necessitates limiting the number of objects sampled in each "sampling from the head" round. Otherwise, the prediction accuracy for eviction candidates that are not promptly removed may degrade over time, negatively impacting the miss ratio.

Additionally, it is essential to ensure that the space occupied by new objects stably reduces to $Q\%$. This requires ensuring that m and n are proportional, with $m \geq n+1$. Thus, we set $m = 1.5n$, ensuring that the condition is met even when $n = 2$.

4.3.2 Object eviction

We first set the eviction ratio to determine the number of evicted objects after each sampling. Second, to improve eviction accuracy, we propose evicting the object with the furthest next request time from the accumulated prediction results of the objects that were not evicted. This approach leverages the fact that the prediction results are accurate prior to model updates, and the accumulated data provides a wealth of valid information. Additionally, to efficiently identify the object with the furthest next request time when the accumulated results reach the millions, we propose a management method.

Eviction ratio. In production traces, there are two primitive object access patterns: 1) frequently accessing only a subset of cached objects over a period of time, and 2) newly arrived objects not being accessed again after a certain period [62]. In these cases, the optimal strategy is to evict only the objects that have been waiting for a long time in the queue or the newly arrived objects. We designed two sampling methods to target these two types of objects separately. We also found that the number of eviction candidates sampled by these two methods is approximately 1:1 overall. Thus, to reduce the eviction of popular objects under any access pattern while minimizing CPU overhead, we set the ratio to 1/2. Moreover, we proved that further increasing the ratio would result in a

Table 2: Auto-tuning parameter rules.

Name	Initial value	Trigger event	Condition	Action	Description
h_{sw}	2	Model update during training	Satisfying Eq.1& Eq.2	$h_{sw}+=1$	Coarse-grained adjustment of the sliding window to quickly find the appropriate size.
f	1	End of a round of sampling from the tail	/	Update to the 99th percentile of the access times of the evicted objects in the round	Reduce the probability of sampling popular objects in "sampling from the tail".
x	1	End of a round of sampling from the tail	$p_1 > p_2$	$x+=1$, or $x-=1$	Determine the proportion of unpopular objects at the head of the cache queue.
Q	2	End of a round of sampling from the tail	$c_1 > c_2$	$Q+=1$, or $Q/=2$	Determine the proportion of newly arrived objects in the cache queue.
n	2	End of a round of sampling from the tail	/	min (1024, $1\% \cdot L+2$)	The proportion of newly arrived objects in the eviction process does not deviate significantly from $Q\%$.

sharp increase in the miss ratio by experiments.

Efficient management. We propose using a max heap structure combined with a hash table to store the prediction results. The management processes are as follows:

Query: The top of the max heap represents the eviction candidate with the furthest next access time.

Add: When an eviction candidate is sampled and predicted for the first time, the prediction result for that object is inserted into both the heap and the hash table.

Modify: When an eviction candidate is sampled and predicted again, the new prediction result is directly inserted into the heap, and the mapping value of the object in the hash table is updated to the new prediction result.

Delete: It includes three cases: (1) When the candidate with the farthest access is evicted, the data at the top of the heap is removed. It's necessary to check whether the data matches the corresponding data in the hash table. If they are consistent, the corresponding data in the hash table is removed; otherwise, no action is taken on the hash table. (2) When a candidate is accessed, it indicates that the object's arrival process has changed, making the previous prediction result unreliable. Thus, the result for the object in the hash table is removed. (3) When the prediction model is updated or a round of sampling is completed, implying that all data in the heap and hash table are no longer valid, these data are deleted.

The operations described above lead to asynchronous updates between the heap and the hash table. While the data in the heap might not always be valid, the records in the hash table are consistently valid. Thus, when retrieving the data from the top of the heap, it is crucial to check if the object exists in the hash table and matches its recorded value. If it does, the data is valid; otherwise, the data is deemed invalid.

4.4 Auto-tuning Parameter

The values of the 3L-Cache parameters h_{sw}, f, x, Q and n vary significantly across different traces and cache sizes. We propose a low-overhead auto-tuning method for these parameters, as shown in Table 2.

h_{sw}: The initial value ensures that the sliding window contains at least twice the information of the cache queue used for model training. Since the model's training samples are primarily derived from object hits, the hit ratio is used to represent the probability of providing valid sample information. Eq. 1 implies that uncached objects within the sliding window need to provide at least 1% of the valid training sample information. Eq. 2 limits the proportion of samples provided by uncached objects in the sliding window, preventing ineffective samples from being used in model training, which could reduce the model's accuracy.

$$\frac{p_s - p_q}{(h_{sw}-1)} > 0.01 \cdot p_q \tag{1}$$

$$(h_{sw}-1)\cdot(p_s - p_q) < 1 - p_q \tag{2}$$

where p_s is the hit ratio within the sliding window, p_q is the hit ratio of the cache queue. These variables are recorded and updated whenever the sliding window becomes full.

f: The initial value enables that only one-hit wonders objects are sampled as eviction candidates when there is no historical information.

x**,** Q: They are adjusted with a granularity of 1%. A finer granularity (e.g., 0.1%) results in negligible changes in the miss ratio, while a coarser granularity (e.g., 10%) leads to significant fluctuations. The initial values are intended to accumulate preliminary information, thereby improving the accuracy of subsequent sampling.

Each round of sampling is divided into two phases: sampling the first $x\%$ of objects and then sampling the remaining objects. The proportion of evicted objects to sampled objects is calculated separately for each phase, denoted as p_1 and p_2. If $p_1 > p_2$, it indicates that the objects sampled in first phase are more likely to be evicted, so we increase the sampling range by incrementing x by 1. Otherwise, x is decreased by 1.

By comparing the number of evicted objects from "sampling from the tail" (c_1) with the number of newly arrived objects in the recorded queue (c_2), if $c_1 > c_2$, it indicates that more objects from "sampling from the tail"should be evicted than newly arrived objects. Therefore, we increase the proportion of newly arrived objects by incrementing Q by 1 to further evict unpopular objects in the "sampling from the tail". Otherwise, Q is divided by 2.

n: L is the number of objects in the cache queue. The upper limit for n is set to 1024 based on the CPU's performance.

5 Evaluation

In this section, we evaluate 3L-Cache to address the following key questions:

- How does 3L-Cache compare to state-of-the-art eviction policies in terms of byte and object miss ratios? (§5.2)
- Does 3L-Cache exhibit CPU overhead comparable to heuristic policies? To what extent are the overheads of training and prediction reduced? (§5.3)
- Can 3L-Cache automatically tune the necessary parameters to achieve optimal performance? (§5.4)
- Is 3L-Cache 's acceleration method applicable to other approaches, allowing a reduction in the CPU overhead of learning-based policies? (§5.5)
- How closely does the performance of 3L-Cache 's prototype align with its simulation results? (§5.6)

5.1 Methodology

Traces. Our experiments utilize 8 open-source datasets collected between 2015 and 2023, with detailed information provided in Table 3. The datasets encompass 4,855 traces, approximately 14.72 billion objects, and 266.95 billion requests, all standardized in CSV format. They reflect a broad range of workload characteristics and are applied in diverse caching systems. This diversity ensures a comprehensive evaluation of performance.

Cache eviction policies. We compare 12 state-of-the-art cache eviction policies. It includes 6 heuristic policies (LHD [18], GDSF [29], ARC [53], SIEVE [87], S3-FIFO [84] and TinyLFU [32]), and 6 learning-based policies (LeCaR [73], CACHEUS [62], GL-Cache [80], LRB [64], HALP [65] and our proposed 3L-Cache).

Metrics. Due to the diversity of workloads across different traces, the miss ratios vary significantly. To enhance the visualization of our evaluation results, we present the miss ratio reduction relative to LRU (a baseline policy in this work): $\frac{mr_{LRU}-mr_{alg}}{mr_{LRU}}$, where mr is the miss ratio, representing byte miss ratio or object miss ratio. Additionally, the computation overhead of an eviction policy is presented by the CPU usage relative to LRU: $\frac{CPU_{alg}}{CPU_{LRU}}$. Since the cache size significantly impacts the performance of policies, all metrics are evaluated under two representative sizes: 0.1% and 10% of the footprint (size of all objects in the trace), referred to as the small cache size and large cache size, respectively.

Simulator. We implement the 3L-Cache simulator based on the libCacheSim library[7]. For all the state-of-the-art policies compared, we use the parameters described in their original papers. In the simulation experiment, we replay the trace under stress-testing conditions and record the CPU overhead and miss ratio of the simulation system.

Prototype. Our prototype is implemented using Python Web Framework [6] and real HTTP requests are sent using Python urllib. Upon receiving a request, the prototype applies various eviction policies to process the request and make caching decisions. During operation, we monitor real-time CPU usage to evaluate the overhead introduced by different eviction policies. All experiments are conducted on a Dell EMC Edge T640 server, equipped with dual Intel Xeon Gold 5128R 20-core processors and 192GB of RAM.

Table 3: Datasets used in this work (only including traces with over 10 million requests).

Datasets	Approx time	# Traces	Cache type	# Request (million)	# Object (million)
CloudPhysics [74]	2015	106	Block	2214	492
TencentPhoto [90, 91]	2018	2	Object	5650	1038
Wikipedia CDN [13, 64]	2016-19	3	Object	8400	126
Tencent CBS [88]	2020	4030	Block	33690	551
Twitter [82]	2020	54	KV	195441	10650
Alibaba [2, 47]	2020	652	Block	19676	1702
Meta KV [8]	2022	5	KV	1644	82
Meta CDN [8]	2023	3	Object	231	76

5.2 Miss Ratio

In this section, we compare the byte miss ratio and the object miss ratio of different cache eviction policies. For datasets with more than 50 traces (CloudPhysics, Tencent CBS, Twitter and Alibaba), we refer to them as **large datasets** and use box plots to report aggregated results. For datasets with fewer than 6 traces (Wikipedia, Tencent Photo, Meta KV, and Meta CDN), we refer to them as **small datasets**, and scatter plots are used to present the finding of every trace.

5.2.1 Byte miss ratio

Figure 6 and Figure 7 illustrate the reduction in byte miss ratio achieved by various eviction policies (compared to LRU). The results indicate that object-level learning policies (LRB, HALP, 3L-Cache) generally outperform other policies. Our proposed 3L-Cache demonstrates excellent performance across multiple datasets, excelling in scenarios involving large datasets and highlighting its superiority over other policies. Across all traces, 3L-Cache achieves the lowest byte miss ratio for 69.8% of the traces with small cache sizes and 41.2% of the traces with large cache sizes. In contrast, the next best policy (LRB) achieves only 10.2% and 18.5% respectively, which is significantly worse than 3L-Cache.

In addition, we observed that GL-Cache performs poorly in terms of byte miss ratio, mainly due to its preference for evicting objects with small sizes, which fails to effectively reduce the byte miss ratio. We also observed that, on some datasets, the learning-based caching policies LeCaR and CACHEUS do not perform as well as the heuristic policies SIEVE and S3-FIFO. This is because LeCaR and CACHEUS use machine learning to switch between two preset heuristic policies (such as LRU and LFU), limiting their performance to those heuristics. When SIEVE and S3-FIFO are better suited to

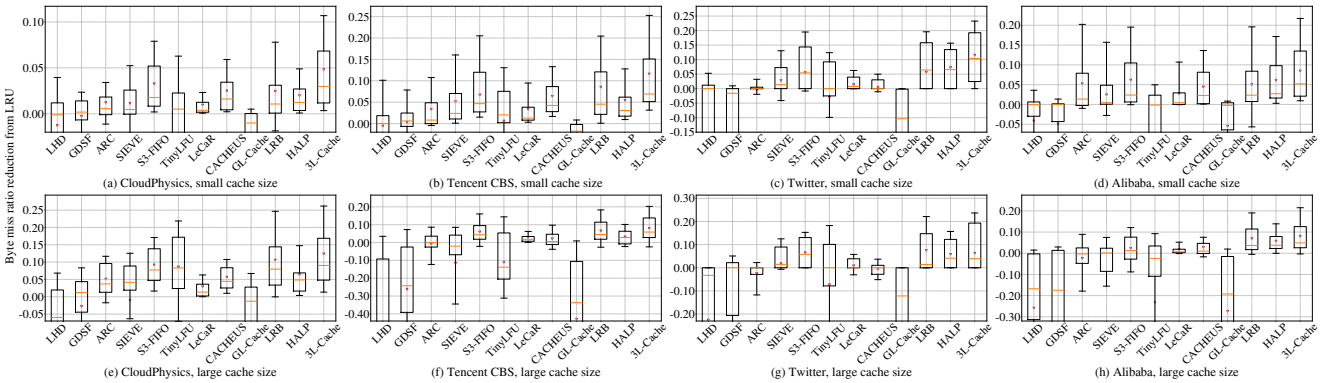


Figure 6: The figure shows the byte miss ratio reduction from LRU on large datasets. In the box plots, the box boundaries represent the 25th and 75th percentiles of the results, with the line inside the box indicating the median. The whiskers extend to the 10th and 90th percentiles, while the triangle denotes the average.

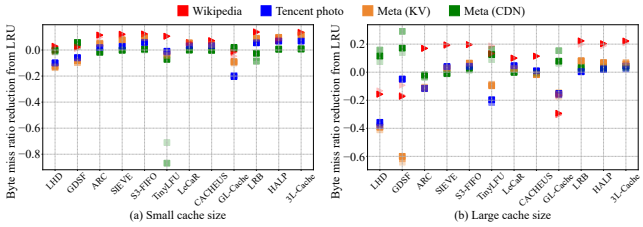


Figure 7: Byte miss ratio on small datasets. Different opacity of the same color indicates multiple traces from the dataset.

a trace's workload than the heuristics used by LeCaR and CACHEUS, they achieve a lower byte miss ratio.

Four large datasets. Figure 6 shows that 3L-Cache demonstrates the most significant reduction in byte miss ratio across nearly all percentiles. For small cache sizes, 3L-Cache achieves a higher reduction in byte miss ratio compared to other policies. For example, 3L-Cache decreases LRU's byte miss ratio by an average of 11.6% on Tencent CBS, while LRB decreases it by only 8.6% (Figure 6b). For large cache sizes, although the difference in byte miss ratio between policies diminishes, 3L-Cache remains the best. It reduces LRU's byte miss ratio by an average of 7.9%, while LRB reduces it by 6.7% (Figure 6f). These results are mainly due to 3L-Cache 's sampling strategy, which reduces the eviction of popular objects compared to the random sampling used by LRB. Additionally, the accumulated prediction results enable more accurate eviction decisions.

Notably, S3-FIFO also performs well, even better than HALP, achieving the 3rd-best performance. This is mainly because in the existing production environment, most workloads follow the Zipf distribution, and S3-FIFO can adapt to such workloads. However, S3-FIFO also has some limitations: as the power-law exponent decreases, the workload distribution becomes more uniform, which can lead to a decline in performance. In contrast, 3L-Cache, which evicts objects based on prediction results, is less affected by this issue.

Four small datasets. Figure 7 shows that 3L-Cache is not the best policy across four small datasets simultaneously, but it consistently outperforms LRU in terms of byte miss ratio. For small cache sizes, 3L-Cache performs slightly worse than LRB on Wikipedia and worse than GDSF and LHD on Meta CDN (Figure 7a). For large cache sizes, 3L-Cache is inferior to LeCaR on Tencent Photo, LRB on Meta KV, and LHD, GDSF, S3-FIFO, TinyLFU, and GL-Cache on Meta CDN (Figure 7b). However, some policies, such as LHD, ARC, TinyLFU, and GL-Cache, occasionally fail to reduce LRU's byte miss ratio, while 3L-Cache consistently avoids this issue.

5.2.2 Object miss ratio

When evaluating the object miss ratio, the eviction weight for 3L-Cache is the (reuse_distance $*$ object_size), rather than the next arrived time, while sampling all scanned objects without updating Q. The main reason is that when the request frequency of all objects is the same, prioritizing the eviction of larger objects increases the number of cached objects, thereby reducing the object miss ratio. Therefore, eviction policies that consider object size in their eviction decisions can achieve lower object miss ratios, as shown in Figure 8 and Figure 9. Policies such as LHD, GDSF, LRB, GL-Cache, and 3L-Cache, which incorporate object size into their eviction weights, generally achieve lower object miss ratios compared to other policies. According to statistics, 3L-Cache achieves the lowest object miss ratio in 66.3% of the traces with small cache sizes and 29.3% of the traces with large cache sizes.

Four large datasets. In Figure 8, we observe that 3L-Cache demonstrates the greatest reduction across nearly all percentiles for small cache sizes. For example, 3L-Cache reduces LRU's object miss ratio by more than 47.9% at the 90th percentile, with a mean reduction of 23.4% on Tencent CBS under small cache sizes (Figure 8b). For large cache sizes, 3L-Cache is inferior to GDSF on CloudPhysics (Figure 8e) and Tencent CBS (Figure 8f), and LHD on Alibaba (Figure 8h).

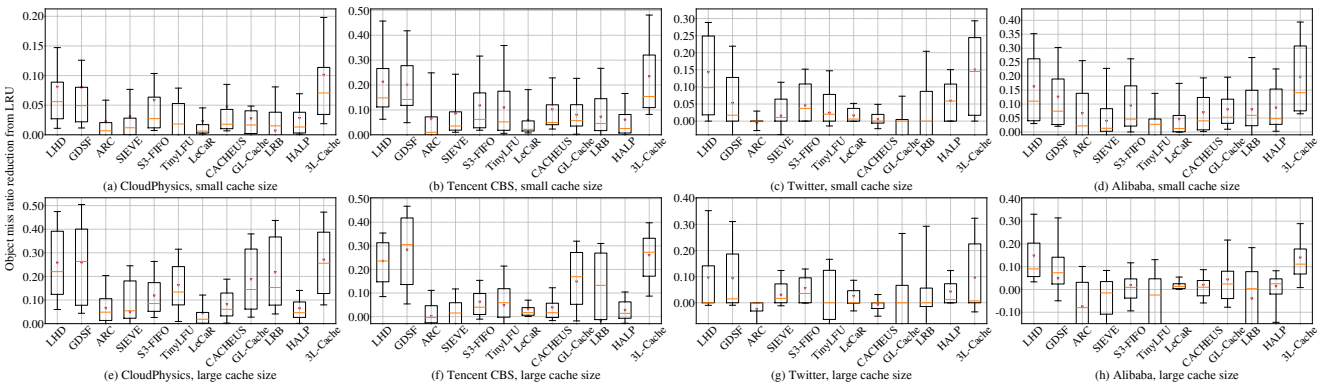


Figure 8: Object miss ratio reduction from LRU on large datasets.

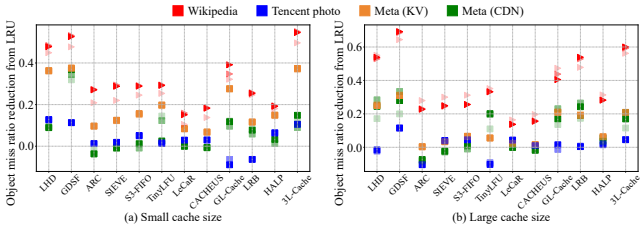


Figure 9: Object miss ratio on small datasets.

In addition to 3L-Cache, LHD and GDSF can also achieve low object miss ratios, with their reductions even surpassing that of LRB. For example, in Figure 8b, the mean reduction for LHD is 14.1% higher than that of LRB. LHD can achieve a low object miss ratio mainly because its hit density considers object size. However, when all objects are of the same size, LHD evicts them based solely on age. Many production environments today use slab storage [10] for caching, which narrows the size gap between different objects and may lead to a significant increase in LHD's object miss ratio [87].

Four small datasets. Figure 9 demonstrates that LHD, GDSF, GL-Cache, LeCaR, LRB, and 3L-Cache perform best on specific traces and cache sizes. For example, on Tencent Photo, LHD is the best policy for small cache sizes, but its performance is worse than LRU for large cache sizes. The main reason is that the one-hit-wonder ratio of Tencent photo is higher than 50%. For small cache sizes, there are a few objects that are accessed frequently, making them easier to cache. However, for large cache sizes, GDSF achieved the best performance across all four datasets. This is primarily because, with large cache sizes, there are more cached objects, and both LHD and 3L-Cache rely on sampling-based eviction, which limits their performance. In contrast, GDSF evaluates the eviction weights of all cached objects, leading to better performance for large cache sizes.

5.2.3 Trade-offs Between Byte and Object Miss Ratios

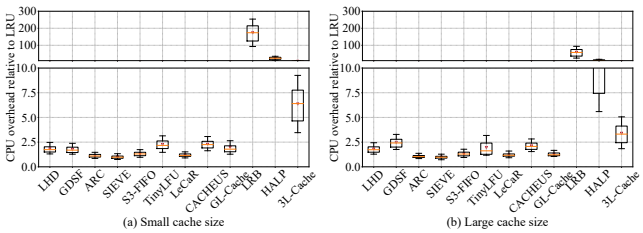
As discussed in §5.2.1 and §5.2.2, the eviction weights differ depending on whether the goal is to minimize the byte miss ratio or the object miss ratio. Operators can select the appropriate policy based on their primary objective. We evaluate all twelve policies across all 4855 traces, measuring the byte miss ratio under the optimal object miss ratio policy, as well as the object miss ratio under the optimal byte miss ratio policy. Due to space limitation, we only describe the average miss ratio ranking of 3L-Cache. Notably, 3L-Cache maintains reasonably good performance even under non-optimized policies, despite some performance degradation.

Object Miss Ratio under the Best Byte Miss Ratio Policy. 3L-Cache overall achieves a lower object miss ratio than ARC, SIEVE, S3-FIFO, TinyLFU, LeCaR, CACHEUS, LRB and HALP, but worse than LHD and GDSF. Although 3L-Cache performs worse than GL-Cache for large cache sizes, it outperforms GL-Cache for small cache sizes overall. Additionally, unlike LHD and GDSF, which exhibit some regressions compared to LRU, 3L-Cache outperforms LRU overall.

Byte miss ratio under best object miss ratio policy. 3L-Cache overall performs better than LHD, GDSF, GL-Cache and LRB, but worse than ARC, SIEVE, S3-FIFO, LeCaR, CACHEUS, and HALP. However, we found that byte miss ratio of 3L-Cache did not significantly decrease on small cache sizes, for example, on Tencent CBS, the byte miss ratio of 3L-Cache remained optimal.

5.3 CPU Overhead

As an object-level learning policy, 3L-Cache not only achieves low miss ratios but also reduces CPU overhead. Figure 10 compares the CPU overhead of all twelve policies evaluated on all 4855 traces when measuring object miss ratio. We observe a significant decrease in the CPU overhead of 3L-Cache compared to other two object-level learning policies. For example, 3L-Cache reduces the average CPU overhead by 60.9% compared to HALP and by 94.9% compared to LRB (Figure 10a). The mean CPU overhead of 3L-Cache is only 2.6× higher than that of LHD and 6.4× higher than that of LRU on average. Furthermore, as shown in Figure 10b, the mean CPU overhead of 3L-Cache is 3.4× that of LRU.



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

Figure 10: CPU overhead of each policy at various percentiles across 4855 traces. Lower values indicate better performance.

Table 4: Percentage of CPU overhead for training, prediction, and other steps in 3L-Cache.

	Training	Prediction	Others
Small cache size	12.5%	48.5%	39%
Large cache size	19.0%	18.3%	62.7%

Comparing Figure 10a and 10b, we find that as the cache size increases, the CPU overhead of 3L-Cache, HALP and GL-Cache decreases significantly. This is because the larger cache size reduces the object miss ratio, thereby lowering the prediction overhead and resulting in a significant reduction in overhead. In contrast, LeCaR and CACHEUS are not based on predicting cached object features but use machine learning to adapt to workload changes. Since workloads do not change with cache size, their overhead remains relatively stable.

3L-Cache achieves low CPU overhead mainly by reducing prediction and training overhead, and the CPU overhead for others mainly includes operations such as queue maintenance and hash queries. As shown in Table 4, under large cache sizes, the training and prediction overhead in 3L-Cache is only 37.3%, which is significantly lower than HALP's 64.5% and LRB's 90.9%. To mitigate prediction overhead, 3L-Cache attains a high eviction ratio (1/2) and the lowest byte miss ratio and object miss ratio. Regarding training overhead, 3L-Cache reduces it by decreasing the frequency of model updates. As shown in Table 5, 3L-Cache can reduce training overhead by up to 92.5% and prediction overhead by up to 98.5% compared to LRB. Additionally, 3L-Cache can reduce training overhead by up to 46.6% and prediction overhead by up to 85.6% compared to HALP.

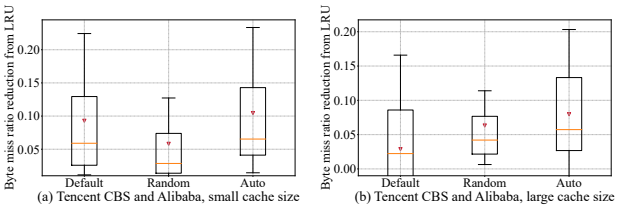
5.4 Parameter Auto-tuning

We compare our parameter auto-tuning strategy with random adjustment strategy and default value without parameter adjustment. The tuning parameters involve h_{sw}, x, f, n, and Q. The random adjustment strategy is to randomly select a value for each parameter within a reasonable range of adjustment when updating the conditional method. The default value strategy is to set h_{sw}, x, f, n and Q as fixed initial values, and not update these parameters again.

We conduct experiments on the Tencent CBS and Alibaba datasets including 4682 traces, as shown in Figure 11. Our auto-tuning method is significantly better than the other two methods. In summary, our auto-tuning methodology can achieve the best performance on 81.6% traces under small cache sizes and 53.6% traces under large cache sizes.

Table 5: The CPU overhead reduction of 3L-Cache compared to LRB and HALP in training and prediction, respectively.

Compared policy	Training		Prediction	
	LRB	HALP	LRB	HALP
Small cache size	-90.9%	-34.7%	-96.5%	-76.1%
Large cache size	-92.5%	-46.6%	-98.5%	-85.6%



(a) Tencent CBS and Alibaba, small cache size (b) Tencent CBS and Alibaba, large cache size

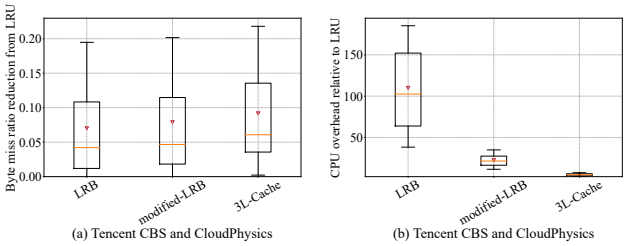
Figure 11: The impact of different parameter adjustment strategies on byte miss ratios of 3L-Cache.

5.5 Accelerating LRB with 3L-Cache

We further leverage the methods from 3L-Cache to accelerate LRB by replacing its random sampling policy and eviction strategy, which form the core design of 3L-Cache for reducing CPU overhead. Our goal is to verify whether this design can also enhance other learning-based policies in terms of overhead reduction. As depicted in Figure 12, the accelerated LRB achieves a CPU overhead reduction of 80% compared to the unmodified LRB, but it is still higher than 3L-Cache. This is primarily because the accelerated LRB only reduces prediction overhead, while training overhead remains high. Moreover, the figure also shows that the accelerated LRB decreases the byte miss ratio of the unmodified LRB by an average of 0.6%, but still worse than 3L-Cache. This is primarily due to the randomness of LRB's random sampling, which may sample and evict popular objects, whereas 3L-Cache's sampling strategy can reduce the probability of popular objects being sampled.

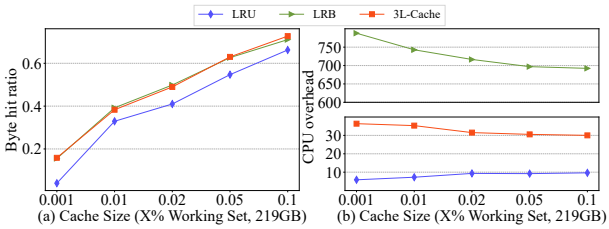
5.6 Performance of 3L-Cache Prototype

We compare 3L-Cache prototype with LRU, and LRB using the Tencent CBS trace (ns3246). Figure 13 shows the byte hit ratio and CPU overhead. For the byte hit ratio, we found the difference between prototype and simulation is only within 2%. For example, when cache size is 10% working set, the byte hit ratio of 3L-Cache is 72.6% in prototype and 72.7% in simulation. For CPU overhead, 3L-Cache is $3\times \sim 6\times$ that of LRU, while in simulation experiments it was about $5\times$ that of LRU. Therefore, 3L-Cache can achieve similar results in both simulation and prototype.



(a) Tencent CBS and CloudPhysics (b) Tencent CBS and CloudPhysics

Figure 12: The application of our design on LRB, comparing the impact on byte miss ratio and CPU overhead across traces.



(a) Cache Size (X% Working Set, 219GB) (b) Cache Size (X% Working Set, 219GB)

Figure 13: Prototype evaluation on a Tencent CBS trace.

6 Related Work

6.1 Heuristic Eviction Policy

Most heuristic policies optimize eviction decisions based on features [14, 15, 18, 21, 23, 29, 32, 38, 44, 44, 45, 46, 53, 56, 58, 59, 61, 67, 77, 83]. For example, LRU, LIRS [67], LRU-K [56], ARC [53], CFLRU [59] use recency as the eviction weight; LFU, LRFU [44, 45] based on the request frequency; LHD [18], TTL [21], TinyLFU [32], DLIRS [46] mix multiple features to evict. Additionally, a small number of studies focus on increasing the efficiency of heuristic policies. For example, S3-FIFO [84] provides a guaranteed demotion speed and high demotion precision; SIEVE [87] eliminates the need for locking during cache hits. However, heuristic policies share a common issue: instability. They exhibit significant differences in miss ratios under different workloads.

6.2 Learning-based Eviction Policy

We divide the existing learning-based eviction policies into three classes by comprehensively considering miss ratios and computation overhead. In the following, we analysis pros and cons of each class.

Policy-level learning. The work uses reinforcement learning to make eviction decisions. It either learns to choose one expert from multiple experts (LRU and LFU) such as LeCaR [73], CACHEUS [62] or directly learn the optimal eviction policy such as Rl-Cache [42], Trend_Caching [49], WDDQN [48]. These two types have a common problem inherent in reinforcement learning: delayed perception of changes in the workload. This results in sub-optimal caching decisions being made before the model adjusts to optimal caching decisions. Therefore, the performance improvement of this class has the bottleneck, especially when the workload changes frequently.

Group-level learning. Grouping is first used in GL-Cache [80]. However, GL-Cache has the following limitations: (1) It only performs well when arrived objects have strong temporal correlation due to its arrive-time-based grouping. (2) The utility of a group is hard to reflect its actual state. Group utility in Gl-cache is the sum of object utilities. An object utility is defined as $\frac{1}{t \times s}$, where t is next request time, and s is object size. Therefore, when a group contains only a very small number of popular small objects, it is difficult for the group to be evicted even if the other objects are unpopular, since the size difference among objects spans eight orders of magnitude in some traces (Wikipedia CDN). Thus, group-level learning performs well on a limited number of datasets.

Object-level learning. It performs training and prediction on each object. Multiple works have studied on making eviction decision by predicting objects' popularity [27, 28, 33, 43, 49, 54, 55, 63, 69, 71, 86, 89], hit probability (e.g., HRO [57] and LHR [79]), or reuse distance (e.g., Raven [37], LRB [64], LFO [20], Hawkeye [39], Demand-MIN [40], Parrot [50], HALP [65], PredictiveMarKer [52], RLB [78], SLAP [51], LRU-BaSE [76]). This type of policy has the potential to achieve the lowest miss ratio among all learning-based policies with large enough data and a complex model [80]. However, it inevitably incurs the highest computation overhead due to the finest-grained prediction and training.

7 Conclusion

We propose 3L-Cache, the first learning-based cache eviction with low CPU overhead, low object miss ratio and low byte miss ratio. 3L-Cache addresses the challenges of reducing training and prediction overhead without sacrificing the miss ratios. We finally verify that 3L-Cache can be applied to other object-level learning policies to enhance their performance. If an atypical web caching workload exhibits fewer Zipf access patterns, the objects sampled by 3L-Cache may not effectively represent low popularity. Instead, their popularity might align closely with the overall cache queue, causing the sampling effectiveness to degrade to that of random sampling. In future work, we plan to analyze various access patterns in production environments and design an efficient eviction policy that can adapt to diverse workloads.

Acknowledgments

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Juncheng Yang for his thorough guidance in our camera-ready preparation. This research was supported by National Natural Science Foundation of China (NSFC) 62101012, and Beijing Postdoctoral Research Foundation 2021-ZZ-079.

References

[1] Akamai platform. https://aws.amazon.com/about-aws/global-infrastructure/. Accessed: 2024-09-17.

[2] Alibaba block traces. https://github.com/alibaba/block-traces. Accessed: 2023-09-12.

[3] Apache traffic server. https://trafficserver.apache.org/. Accessed: 2023-09-20.

[4] Aws global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/. Accessed: 2024-09-17.

[5] Benefits of caching. https://www.analysysmason.com/contentassets/cb6f16932339489fa50ca31f6e89244d/analysys-mason_benefits-of-caching_may-2020.pdf. Accessed: 2024-04-27.

[6] Flask. https://github.com/pallets/flask. Accessed: 2024-05-03.

[7] libcachesim. https://github.com/1a1a11a/libCacheSim/. Accessed: 2023-09-12.

[8] Meta. https://cachelib.org/docs/Cache_Library_User_Guides/. Accessed: 2024-05-03.

[9] Nginx. https://nginx.org/. Accessed: 2024-05-03.

[10] Pelikan. https://github.com/twitter/pelikan. Accessed: 2023-09-12.

[11] Redis. http://redis.io/. Accessed: 2024-05-03.

[12] Varnish cache. https://varnish-cache.org/. Accessed: 2024-05-03.

[13] Wikipedia trace. https://ftp.pdl.cmu.edu/pub/datasets/. Accessed: 2023-09-12.

[14] Bahman Abolhassani, John Tadrous, and Atilla Eryilmaz. Single vs distributed edge caching for dynamic content. *IEEE/ACM Transactions on Networking*, 30(2):669–682, 2021.

[15] Bahman Abolhassani, John Tadrous, Atilla Eryilmaz, and Edmund Yeh. Fresh caching for dynamic content. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[16] Susan Athey, Julie Tibshirani, and Stefan Wager. Generalized random forests. 2019.

[17] 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.

[18] 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.

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

[20] 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.

[21] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):57–59, 2015.

[22] 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.

[23] 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.

[24] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[25] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of zipf's law for web caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998.

[26] 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.

[27] 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.

[28] Livia Elena Chatzieleftheriou, Merkouris Karaliopoulos, and Iordanis Koutsopoulos. Caching-aware recommendations: Nudging user preferences towards better caching performance. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[29] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *International Conference on High-Performance Computing and Networking*, pages 114–123. Springer, 2001.

[30] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[31] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

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

[33] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. Pa-cache: Evolving learning-based popularity-aware content caching in edge networks. *IEEE Transactions on Network and Service Management*, 18(2):1746–1757, 2021.

[34] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28, 2007.

[35] Eric Gourdin, Patrick Maillé, Gwendal Simon, and Bruno Tuffin. The economics of cdns and their impact on service fairness. *IEEE Transactions on Network and Service Management*, 14(1):22–33, 2017.

[36] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[37] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pages 72–90, 2022.

[38] 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.

[39] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News*, 44(3):78–89, 2016.

[40] Akanksha Jain and Calvin Lin. Rethinking belady's algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 110–123. IEEE, 2018.

[41] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[42] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. Rl-cache: Learning-based cache admission for content delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 57–63, 2019.

[43] Elias Koutsoupias and Christos H Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30(1):300–317, 2000.

[44] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 134–143, 1999.

[45] 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.

[46] 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.

[47] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020.

[48] Ruibin Li, Yiwei Zhao, Chenyang Wang, Xiaofei Wang, Victor CM Leung, Xiuhua Li, and Tarik Taleb. Edge caching replacement optimization for d2d wireless networks via weighted distributed dqn. In *2020 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2020.

[49] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. Trend-aware video caching through online learning. *IEEE Transactions on Multimedia*, 18(12):2503–2516, 2016.

[50] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

[51] Ke Liu, Kan Wu, Hua Wang, Ke Zhou, Peng Wang, Ji Zhang, and Cong Li. Slap: Segmented reuse-time-label based admission policy for content delivery network caching. *ACM Transactions on Architecture and Code Optimization*, 2024.

[52] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the ACM (JACM)*, 68(4):1–25, 2021.

[53] 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)*, 2003.

[54] 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.

[55] The-Vi Nguyen, Nhu-Ngoc Dao, Wonjong Noh, Sungrae Cho, et al. User-aware and flexible proactive caching using lstm and ensemble learning in iot-mec networks. *IEEE Internet of Things Journal*, 9(5):3251–3269, 2021.

[56] 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.

[57] Nitish K Panigrahy, Philippe Nain, Giovanni Neglia, and Don Towsley. A new upper bound on cache hit probability for non-anticipative caching policies. *ACM SIGMETRICS Performance Evaluation Review*, 48(3):138–143, 2021.

[58] 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.

[59] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[60] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. {EC-Cache}:{Load-Balanced},{Low-Latency} cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, 2016.

[61] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on networking*, 8(2):158–170, 2000.

[62] 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.

[63] Alireza Sadeghi, Fatemeh Sheikholeslami, and Georgios B Giannakis. Optimal and scalable caching for 5g using reinforcement learning of space-time popularities. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):180–190, 2017.

[64] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[65] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. {HALP}: Heuristic aided learned preference eviction policy for {YouTube} content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1149–1163, 2023.

[66] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 41–54, 2004.

[67] Bijeta Subedi. *An Evaluation of Page Replacement Algorithm Based on Low Inter-Reference Recency Set (LIRS) Scheme on Weak Locality Workloads*. PhD thesis, Department of Computer Science and Information Technology, 2012.

[68] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67, 2017.

[69] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. Popularity prediction of facebook videos for higher quality streaming. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 111–123, 2017.

[70] Hind Taud and Jean-Franccois Mas. Multilayer perceptron (mlp). *Geomatic approaches for modeling land change scenarios*, pages 451–455, 2018.

[71] Stefano Traverso, Mohamed Ahmed, Michele Garetto, Paolo Giaccone, Emilio Leonardi, and Saverio Niccolini. Temporal locality in today's content caching: Why it matters and how to model it. *ACM SIGCOMM Computer Communication Review*, 43(5):5–12, 2013.

[72] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[73] 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.

[74] 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, 2015.

[75] Joseph Wang, Anne Holler, Mingshi Wang, and Michael Mui. Productionizing distributed xgboost to train deep tree models with large data sets at uber, 2019.

[76] Peng Wang and Yu Liu. Optimizing replacement policies for content delivery network caching: Beyond belady to attain a seemingly unattainable byte miss ratio. *arXiv preprint arXiv:2212.13671*, 2022.

[77] Xianglin Wei, Jianwei Liu, Yangang Wang, Chaogang Tang, and Yongyang Hu. Wireless edge caching based on content similarity in dynamic environments. *Journal of Systems Architecture*, 115:102000, 2021.

[78] 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.

[79] Gang Yan, Jian Li, and Don Towsley. Learning from optimal caching for content delivery. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 344–358, 2021.

[80] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. {GL-Cache}: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, 2023.

[81] Juncheng Yang, Anirudh Sabnis, Daniel S Berger, KV 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, 2022.

[82] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[83] 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)*, pages 503–518, 2021.

[84] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[85] Hao Yin, Xu Zhang, Shuoyao Zhao, Yan Luo, Chen Tian, and Vyas Sekar. Tradeoffs between cost and performance for cdn provisioning based on coordinate transformation. *IEEE Transactions on Multimedia*, 19(11):2583–2596, 2017.

[86] Xu Zhang, Zhengnan Qi, Geyong Min, Wang Miao, Qilin Fan, and Zhan Ma. Cooperative edge caching based on temporal convolutional networks. *IEEE Transactions on Parallel and Distributed Systems*, 33(9):2093–2105, 2021.

[87] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. Sieve is simpler than lru: an efficient turn-key eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association*, 2024.

[88] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (snia iotta trace set 27917). *SNIA IOTTA Trace Repository. Storage Networking Industry Association*, 2018.

[89] 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(3):194–199, 2020.

[90] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (snia iotta trace set 27476). *SNIA IOTTA Trace Repository. Storage Networking Industry Association*, 2016.

[91] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 284–294, 2018.

A Artifact Appendix

Abstract

This artifact implements 3L-Cache with approximately 870 lines of C++ code. It is designed to significantly reduce the CPU overhead of learning-based eviction policies without compromising the miss ratio.

Scope

This artifact can verify that in object-level learning policies, the CPU overhead of 3L-Cache is significantly reduced, and 3L-Cache also achieve the optimal byte miss ratio and object miss ratio. The artifact can serve as a reference implementation for low-overhead, learning-based eviction policies, providing guidance for developing efficient caching eviction policies.

Contents

The artifact is implemented within the libcachesim library. It contains:

(1) **Source code**: Located in the 3LCache folder, it includes:

- TLCache.h: A header file that declares the cache structure and interfaces.
- TLCache.cpp: We mainly implement the following interfaces: lookup() for checking whether a requested object is a cache hit, admit() for inserting new objects that miss the cache into the cache queue, and evict() for evicting cached objects.
- TLCache_interface.cpp: A file that provides an interface for integrating the eviction policy with the libcachesim library.

(2) **Documentation**: Includes two README files:

- Main README: Located in the root directory, this file explains:
  - How to configure the environment and compile the code.
  - The file format of the traces used.
  - Commands for evaluating the performance of different eviction policies on traces.
  - Steps for reproducing the experimental results.
- Detailed README in the 3LCache folder: This file focuses on the implementation details of 3L-Cache, including:
  - The structure and functionality of the implementation files.
  - Explain the contents of the files stored in each folder.
  - Command on how to use the provided scripts.

These two README files provide all the necessary information for setting up, understanding, and reproducing the results of the artifact.

(3) **Scripts**: Benchmarking scripts in the 3LCache/scripts folder are provided to evaluate the performance of different eviction policies on given traces.

(4) **Test Traces**: We provide some trace samples in the data folder from different datasets to measure the differences in CPU overhead and miss ratio between different policies, and can quickly generate corresponding experimental result figures. If you need to obtain complete experimental results, you can download the corresponding traces, process them into the appropriate format, and store them in the data folder, then run scripts.

These components together enable users to configure the environment, compile the code, evaluate the eviction policies, and reproduce experimental results.

Hosting

The artifact is publicly available on GitHub at the following link: https://github.com/optiq-lab/3L-Cache. The latest version of the source code is hosted on the main branch. We recommend cloning or downloading the main branch to obtain the most up-to-date implementation.

Requirements

The artifact has the following software requirements:

- Operating System: Ubuntu 18.04.
- CMake version: 3.28.6.

The artifact was developed and tested on this platform to ensure compatibility and functionality. No additional hardware requirements are needed beyond a standard development environment capable of running Ubuntu.