ε-LAP: A Lightweight and Adaptive Cache Partitioning Scheme With Prudent Resizing Decisions for Content Delivery Networks

Peng Wang, Yu Liu, *Member, IEEE*, Ziqi Liu, Zhelong Zhao, Ke Liu, Ke Zhou, *Member, IEEE*, and Zhihai Huang

***Abstract*—As dependence on Content Delivery Networks (CDNs) increases, there is a growing need for innovative solutions to optimize cache performance amid increasing traffic and complicated cache-sharing workloads. Allocating exclusive resources to applications in CDNs boosts the overall cache hit ratio (OHR), enhancing efficiency. However, the traditional method of creating the miss ratio curve (MRC) is unsuitable for CDNs due to the diverse sizes of items and the vast number of applications, leading to high computational overhead and performance inconsistency. To tackle this issue, we propose a lightweight and adaptive cache partitioning scheme called ε-LAP. This scheme uses a corresponding shadow cache for each partition and sorts them based on the average hit numbers on the granularity unit in the shadow caches. During partition resizing, ε-LAP transfers storage capacity, measured in units of granularity, from the $(N-k+1)$-th ($k \leq \frac{N}{2}$) partition to the k-th partition. A learning threshold parameter, i.e., ε, is also introduced to prudently determine when to resize partitions, improving caching efficiency. This can eliminate about 96.8% of unnecessary partition resizing without compromising performance. ε-LAP, when deployed in *PicCloud* at *Tencent*, improved OHR by 9.34% and reduced the average user access latency by 12.5 ms. Experimental results show that ε-LAP outperforms other cache partitioning schemes in terms of both OHR and access latency, and it effectively adapts to workload variations.**

***Index Terms*—Cache partition, content delivery networks.**

I. Introduction

CONTENT Delivery Networks (CDNs) are responsible for delivering content to users through networks of caching servers. In 2017, CDNs carried 56% of web traffic, and this number is expected to reach 72% by 2022 [1]. CDNs deploy numerous servers worldwide to handle user requests. A cache hit occurs when the requested content is available on a nearby server, leading to faster response times and decreased latency. However, if a cache miss occurs, the content needs to be retrieved from a remote server, leading to a significant increase in latency. As traffic increases, traditional methods like replacement, prefetch, and admission policies find it challenging to analyze request patterns from workloads involving cache-sharing applications. These colocated workloads often suffer significant performance degradation due to resource contention [2]. We display the requests and performance clips from *PicCloud* using the LRU algorithm in Fig. 1, where *PicCloud* is *Tencent*'s[1] latest generation of image storage and service platform, serving more than 15,000+ applications in CDNs. As shown in Fig. 1(a), the requests from *app*_1 far exceed the other applications' requests. It means that the cache will be filled with the content of *app*_1 and will not be available for other applications. Highly-skewed workloads like these are common in the cloud environments [3], [4], [5], [6]. Nevertheless, disregarding other applications does not guarantee an improvement in the overall object hit ratio (OHR), as resource limitations may restrict the potential increase in the OHR of *app*_1. As shown in the miss ratio curve (MRC) of Fig. 1(b), *app*_1 can only fully utilize 100 MB of cache resources and there are more overall benefits at a bigger cache size when other applications share the cache. When *app*_1 and other applications have exclusive cache resources, the OHR has a higher upper limitation. Therefore, cache partitioning has become a crucial solution for enhancing the service quality of CDNs. It involves dividing the cache into separate partitions, and allocating dedicated cache resources to each workload. This approach minimizes interference and contention among concurrent tasks, enabling efficient utilization of cache resources and ultimately boosting performance.

As far as we know, most existing cache partitioning methods primarily use MRC to determine partition sizes. This ensures that each workload benefits from its exclusive resource and acquires an increase of OHR. Table I lists twelve well-known methods, detailing their release year, partitioning criteria, additional memory overhead, and computational complexity. While these

Manuscript received 8 October 2023; revised 17 April 2024; accepted 26 June 2024. Date of publication 28 June 2024; date of current version 6 September 2024. This work was achieved in the Key Laboratory of Information Storage System and Ministry of Education of China. It was supported in part by National Key Research and Development Program under Grant 2023YFB4502701, in part by the National Natural Science Foundation of China under Grant 62232007 and under Grant 61821003, in part by the National Natural Science Foundation of China under Grant 61902135, in part by the Natural Science Foundation of Hubei Province under Grant 2022CFB060, and in part by Ministry of Education-China Mobile Research Funding under Grant MCM20180406. Recommended for acceptance by J. Aghaei. *(Corresponding author: Yu Liu.)*

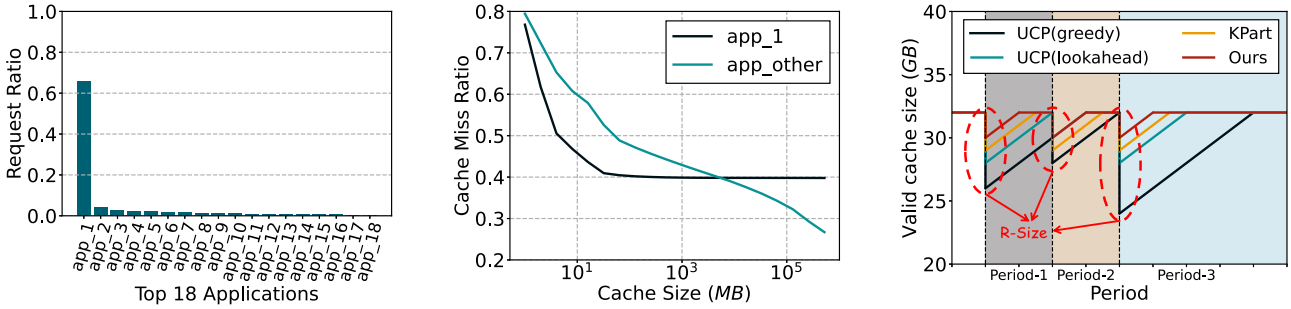
Peng Wang, Ke Liu, and Ke Zhou are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: wp_hust@hust.edu.cn; liu_ke@hust.edu.cn; zhke@hust.edu.cn).

Yu Liu, Ziqi Liu, and Zhelong Zhao are with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: liu_yu@hust.edu.cn; zhelongzhao@hust.edu.cn; liuziqi@hust.edu.cn).

Zhihai Huang is with the Tencent, Shenzhen 518000, China (e-mail: tommyhuang@tencent.com).

Digital Object Identifier 10.1109/TCC.2024.3420454

[1]https://www.tencent.com



(a) Request Distribution (b) MRC (c) Resource Utilization

Fig. 1. (a) The request frequencies of the top 18 applications in PicCloud. (b) MRC of *app*_1 and *app*_other. The *app*_other includes 17 other applications and their requests are executed as if they were an application. (c) Different methods generate the changes in cache resource utilization in three consecutive periods. In this toy example, each method determines the adjustment size individually for the same case, and the same number of requests triggers the end of each period.

TABLE I
EXISTING WELL-KNOWN CACHE PARTITIONING SCHEMES

Method	Year	Partitioning Criteria	Fixed Item Size	Memory Overhead	Computation Complexity
UCP-G [7]	2006	MRC	✘	$N \times C$	$O(P \times N)$
UCP-L [7]	2006	MRC	✘	$N \times C$	$O(P^2 \times N^2)$
PriSM [8]	2012	Eviction Probabilities	✔	$N \times C$	$O(N)$
SHARDS [9]	2015	MRC	✘	$N \times C \times r$	$O(N \times \log(P \times r))$
EMB [10]	2017	MRC	✘	$N \times C$	$O(N \times P^2)$
KPart [11]	2018	MRC	✘	$N \times C$	$O(P^2 \times N^2)$
DCAPS [12]	2018	MRC	✘	$N \times C$	$O(P \times N^2)$
OSCA [4]	2020	MRC	✔	$N \times C$	$O(N)$
LPCA [13]	2022	MRC	✘	$N \times C$	$O(P^2 + N \times logN)$
APAC [14]	2022	MRC	✔	$(N + 2P) \times C$	$O(r \times N + P \times C)$
ACTION [15]	2023	MRC	✔	$(N + 2P) \times C$	$O(N)$
LACA [16]	2023	MRC	✘	$(N + P) \times C$	$O(r \times N)$
Ours	-	**Hits in granularity in shadow cache**	✘	$(N - 1) \times C$	$O(N \times \log N)$

N denotes the number of threads or applications. C represents the cache size. $P = \frac{C}{G}$, where G denotes the minimum granularity of adjustment. r denotes the sampling rate of the corresponding algorithm, which depends on the dataset. The bold data are from our proposed scheme.

methods can generate satisfactory performance, two challenges arise when applying them to replace threads with applications in CDNs.

Challenge ①. High memory and computational overhead caused by mass applications and varying cache item sizes: Cache partitioning has already been a mature hardware-based technology at *Intel*,[2] also known as Cache Allocation Technology (CAT). Since the number of threads and the size of Last-Level Cache (LLC) in CPU are comparatively small, e.g., AMD EPYC^{TM} 7551 Processor has 64 threads with 64 MB LLC,[3] the overhead using MRC in Table I can be acceptable, based on the 1 MB granularity corresponding to a way unit. Nevertheless, in CDNs, the number of applications often exceeds 10^4, and the cache size exceeds 1 TB. Compared to the smallest computational complexity of partitioning in LLC, the partitioning overhead is at least 10^7 times larger in CDNs, based on the 4 MB granularity. Moreover, the lookahead algorithm enables the partitioning scheme, e.g., UCP (lookahead), to tackle the performance cliff problem, requiring higher computational complexity and overhead. In addition, PriSM is lightweight due to the non-MRC partitioning criteria, but it cannot be applied to CDNs because it requires all items to be of equal length [11].

[2]https://www.intel.com
[3]https://www.amd.com/en/product/1956

Challenge ②. Performance fluctuations caused by different cache item sizes and workloads: Another challenge comes from different cache item sizes and various workloads of CDNs. After partition resizing, the cache must pass through a period during which the items are needed to replenish the expanded partitions. Since MRC-based judgments are minimally sensitive to both quantity and size, there may be inefficient resource utilization and performance fluctuations during this period. To quantify the variation caused by the replenishing process, we describe it in terms of resource utilization. As shown in Fig. 1(c), on *PicCloud*, all three classical algorithms experience a period of temporary cache invalidation, and *the summary of sizes of the invalidated space is referred to as R-Size*. We use this size to analyze the performance variations resulting from resizing. As the *R-Size* is adjustable, we believe there is room for improvement in the cache hit ratio during the underutilization of resources.

In addition, we believe that resizing partitions for CDNs by MRC may be untrustworthy since workload variation is more complex in the case of inconsistent item sizes. As shown in Fig. 1(c), the larger the *R-Size*, the greater the risk of resource wastage. This phenomenon arises since once the MRC-based method has determined the partitioning scheme, the data in the cache space that is about to be adjusted becomes immediately invalid. This results in a waste of cache resources until the completion of the entire partition adjustment. Intuitively, we

believe that reducing the variance of *R-Size* can save the cache resources. Based on this hypothesis, through the experiments, we find a range of *R-Size* in the item size distribution corresponding to high hit ratios. We treat all partitions as multiple pairs to resize partitions within the appropriate *R-Size*. For each pair, we transfer a granularity from one partition to the other in a resizing process. For the sake of description, *we define the granularity size as G-Size*, while the sum of *G-Size*s is *R-Size*. Specifically, our scheme and other partitioning schemes resemble each other in shadow cache construction [17]. Still, we use the numbers of hits on granularity in shadow caches to determine partition resizing instead of MRC. We sort all N partitions according to those numbers in descending order in a period. Then, we transfer a *G-Size* from the $(N-k+1)$-th ($k \leq \frac{N}{2}$) partition into the k-th partition. In addition, to balance hit ratio and efficiency, we learn a threshold parameter, ε. It determines whether to implement resizing for pair partitions. This policy can reduce approximately 96.8% of resizing/update operations, saving around 96.8% of time without compromising performance. As shown in Table I and Fig. 1(c), our scheme has lower computational complexity and keeps high resource utilization in workload variations. We have deployed ε-LAP in PicCloud of Company-T and ε-LAP can improve the OHR by 9.34% and reduce the average access latency by 12.5 ms. Extensive experiments on public and real-world traces show that ε-LAP can reduce the miss ratio (MR) by 5%–10% compared to other state-of-the-art partitioning methods in the simulator. Finally, we discuss the performance cliff issue and ε-LAP can alleviate this issue with little overhead.

In this paper, we propose a scheme that enables lightweight and adaptive cache partitioning with prudent resizing decisions in CDNs. The main contributions are outlined as follows:

① Instead of using MRC, we achieve partition resizing in CDNs with little overhead by using a heuristic pattern.

② By analyzing the relationship between *R-Size* and hit ratios, we propose a scheme to reduce fluctuations and improve the hit ratio by resizing in a range at a time. The ε obtained through learning determines whether resizing is performed, thereby enhancing the efficiency of partitioning for CDNs.

II. Background and Motivation

A. A Tencent CDN Case

PicCloud is a cloud service product offered by *Tencent*, which is a typical case of image caching in the context of CDN, as shown in Fig. 2. Various applications connect to PicCloud to share image access, ensuring a high-quality experience (QoE). The architecture of *PicCloud* consists of four key modules, including the Upload module, HTTP module, CDN Acceleration module, and Compress module. In this study, we only focus on the CDN Acceleration module. This module consists of the data center cache (DC) layer and the outside cache (OC) layer [18]. OC is closer to users in terms of access latency and helps enhance the quality of user experience. DC is located within the data center to alleviate the congestion burden on its storage system (COS).

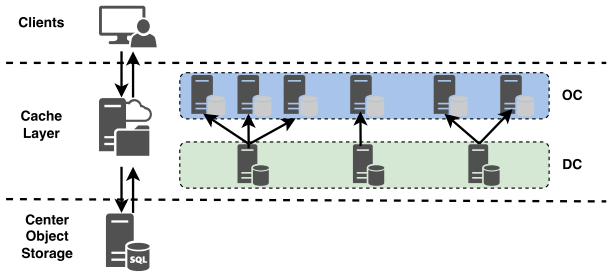


Fig. 2. The system architecture of PicCloud. The cache layer consists of the data center cache (DC) layer and the outside cache (OC) layer.

When users initiate download requests, the HTTP module first identifies whether the required images exist in the cache. If the images exist (i.e., hit the cache), they will be returned to the user from the cache. Otherwise, the images will be retrieved from COS (i.e., back to COS), and the images will be compressed and returned to the user. Meanwhile, the images will be written into the cache for the probable request. In this process, all applications have the equal priority of using cache resources, and preempting the resources based on the principle of "first come, first served".

B. Motivation

Table II illustrates the back-to-COS peak bandwidth in OC for various cities. The poor performance is attributed to a large proportion of the back-to-COS peak bandwidth compared to the total download peak bandwidth. To simplify the description, we use *Trace-BJ* to refer to the data collected from the OC of Beijing. We have attempted to upgrade the replacement algorithm (i.e., LRU) [1], [19] and deploy access admission algorithms [20], [21], but the hit ratio has only improved by 1%. We attribute it to the fact that the workload experiences intense resource competition, and the existing replacement and admission algorithms cannot adjust accordingly. This leads to heavy traffic applications occupying the majority of cache resources without contributing to any hit ratio, as shown in Fig. 1. Consequently, we have to explore alternative solutions.

The cache partitioning schemes in LLC have received considerable attention. These schemes allocate exclusive resources to each application, addressing the problem of uneven workload distribution among threads to ensure stable performance. The problem of uneven requests for applications in CDNs is similar to the thread-related issue. Therefore, adopting the cache partitioning scheme in LLC seems reasonable, specifically using MRC to dynamically resize partitions. However, the workload variation in CDNs is more complex due to the disparity of item sizes, while the instructions in threads are virtually equal in size in LLC. As a result, the MRC-based partitioning leads to performance fluctuations and low resource utilization in CDNs. To mitigate this issue, we propose using a heuristic pattern to achieve partition resizing in CDNs with minimal overhead.

III. Related Work

In this section, we will discuss research related to resource partitioning. Many researchers, inspired by resource isolation

TABLE II
OVERVIEW OF THE FOUR OCS OF PICCLOUD

OC	Number of registered Apps	Daily active Apps	Download peak bandwidth	Back to COS peak bandwidth
Guangzhou	30470	350	29.46 Gbps	2.71 Gbps
Shanghai	76295	653	103.31 Gbps	17.28 Gbps
Tianjin	1744	31	37.69 Gbps	3.51 Gbps
Beijing	15907	118	4.5 Gbps	6.01 Gbps

When a request fails to hit an item in the cache, the request will be sent back to the COS, forming the "back-to-COS" traffic.

techniques, use resource partitioning policies to enhance system performance.

MRC-Based Solutions: Recent studies have designed LLC and memory bandwidth partitioning policies based on hardware resource isolation techniques, e.g., AET [22], mPart [23], Dynacache [24], and Hopscotch [25]. Typically, these studies develop specific models based on extensive knowledge, such as MRC, to predict application performance under different partitioning schemes. Numerous policies for MRC generation have been proposed [22], [26], [27], [28], [29], [30], [31], but they mainly focus on eviction policy. Additionally, as previously mentioned, implementing MRC-based solutions in a CDN is not cost-effective. Several studies [26], [32], [33], [34] have attempted to reduce the overhead of building MRC. However, the heuristics derived from these models only perform well in predefined situations and cannot handle dynamically changing workloads in the CDN.

Multiple Resources Partitions Solutions: Some of the above research only focuses on a single cache resource, while others, such as CoPart [35], Quasar [36], and Heracles [37], focus on the overall allocation of multiple resource partitions. However, due to the absence of precise models that capture the relationship between resources and performance, these partitioning schemes have limited use. In recent years, the application of machine learning in resource management has led to attempts to combine resource allocation with intelligent algorithms like reinforcement learning. For instance, Chen et al. [2] propose a deep reinforcement learning framework to solve the problem of partitioning multiple resources, while Li et al. [38] leverage Q-Learning for adaptive online partitioning. Nevertheless, these methods are inefficient for CDNs because their data storage, model training, and updating require additional resource consumption.

Improvements: We select granularity hits in the shadow cache over MRC as the partitioning principle due to our focus on partitioning efficiency. Our proposed lightweight heuristic algorithm can dynamically adapt to complex workloads in CDNs. Unlike the multi-resource allocation scheme, our method doesn't have to deliberately select resource types, making it more interpretable and generalizable. Furthermore, we propose a learning threshold parameter to reduce invalid update operations and increase the algorithm's online operation efficiency. This is an improvement over the machine learning based allocation algorithm, which has high computation and storage overhead.

IV. RATIONALE OF *R-SIZE* AND *G-SIZE*

Thus, we need to clarify two issues: ① *What kind of R-Size can adapt to the different item sizes and yield a high hit ratio?* ② *How to determine the appropriate R-Size and G-Size?*

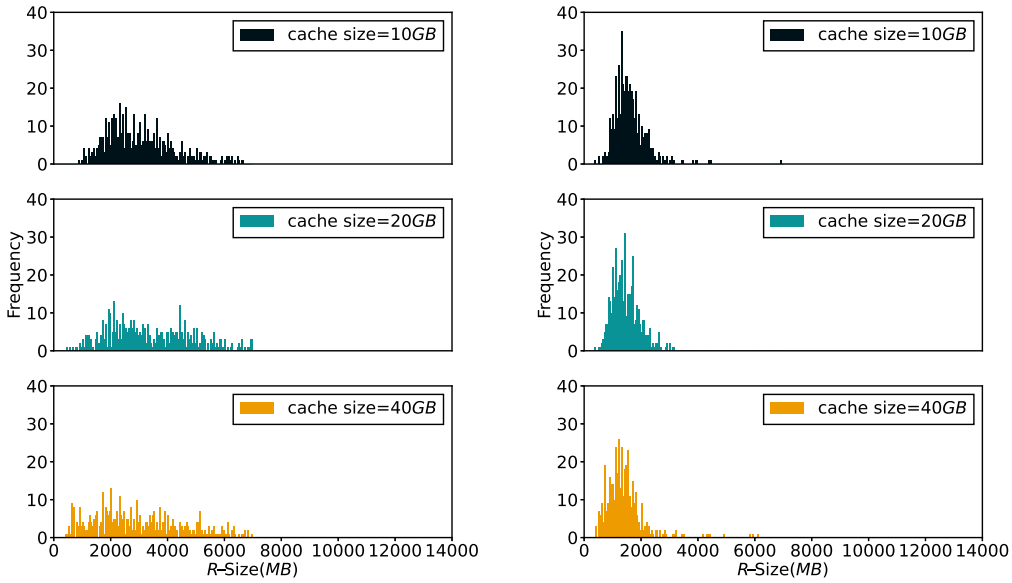
A. The Appropriate R-Size

From the experimental results, we observe that UCP-L consistently achieves a higher hit ratio than UCP-G for the same workload, although it does come with higher computational overhead. Therefore, we gathered their *R-Size* distributions under different cache sizes. As shown in Fig. 3, the distributions of *R-Size* using UCP-G are dispersed and even, whereas those using UCP-L are concentrated, with small values being used more frequently. This law corresponds to the initial adjustment position of UCP-L on the three periods in Fig. 1(c), resulting in relatively high resource utilization. Based on the above observations, we speculate whether setting R-Size with a small range or even a constant can obtain a high hit ratio. If the hypothesis holds, it is possible to design a scheme to apportion *R-Size* instead of a complex decision based on MRC, applying to CDNs.

A simple and feasible scheme is to apportion *R-Size* to half of all partitions in each resizing, with each partition increasing or decreasing by *G-Size*. This scheme can avoid the waste of resources due to the dramatic expansion of a partition. Based on this scheme, we tested the variation of OHR with *G-Size* under different cache sizes. As shown in Fig. 4(b), the OHR curve rises with increasing *G-Size* and falls rapidly after reaching the highest point. Furthermore, as shown in Fig. 4, this phenomenon is insensitive to both cache size and item size distribution, where small item size means narrowing all items' sizes by two times and big item size is just the opposite. In addition, the increase in cache sizes and the concentration of item size distribution will improve the OHR's peak, and the increase in cache sizes will make the peak correspond to a larger *G-Size*. At a relatively small cache size, the *G-Size* corresponding to the peaks are stable, with only a small variation when the cache size is 40 GB. As a result, the small range we anticipated is present.

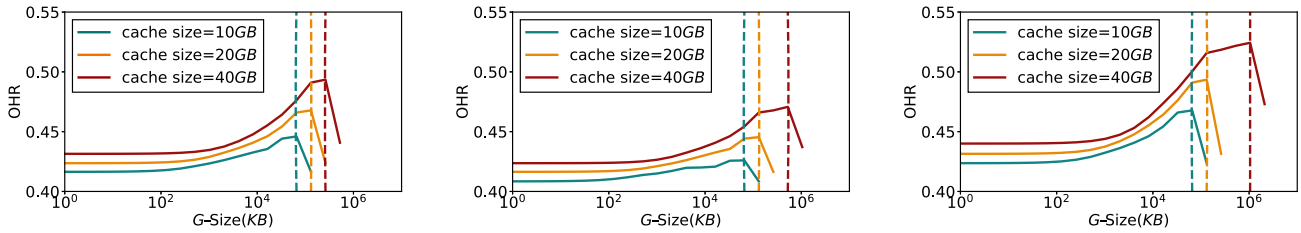
B. Determination of G-Size

Since *G-Size* and *R-Size* are linearly related, we can determine an *R-Size* by the *G-Size*. Intuitively, the *G-Size* should correlate with the item sizes in the CDN, e.g., to ensure that the expanded partition can accommodate more items, the *G-Size* should be larger than most items. To test this notion, we explored the location of appropriate *G-Size* in Fig. 4 corresponding to the position under the cumulative distribution of item sizes. As shown in Fig. 5, the larger the cache size is, the more stable the position of *G-Size* on the CDF curve. Especially at the cache size of 40 GB, the mean of *G-Size*'s positions is 0.96 with a variance of 0.000466 under different item size distributions. Since the cache size in the real environment is large, a reasonable *G-Size* should be stable and close to the size of an estimated maximum item in the CDN.



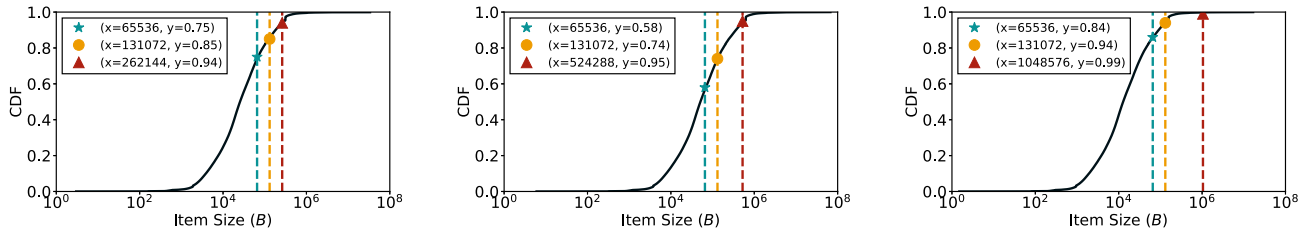
(a) UCP-G (b) UCP-L

Fig. 3. The distributions of *R-Size* using UCP-G and UCP-L under different cache sizes.



(a) Normal item size (b) Big item size (c) Small item size

Fig. 4. Object hit ratio (OHR) at different *G-Size* under different cache sizes and item size distributions.



(a) Normal item size (b) Big item size (c) Small item size

Fig. 5. The relationships between the cumulative distribution function (CDF) of item sizes and the positions of *G-Size* yielding the OHR peak in Fig. 4 under different item size distributions.

V. Lightweight and Adaptive Cache Partition With Decisions

A. Overview

Assume that there are N applications in the cache of C capacity. *The capacity of G-Size is G*. The partition size for the i-th application is C_i, and its corresponding shadow cache size is S_i. These parameters satisfy the following relationship:

$$\begin{cases} \Sigma_{i=1}^{N} C_i = C, \\ C_i + S_i = C, \\ \Sigma_{i=1}^{N} S_i = (N-1) \times C. \end{cases} \tag{1}$$

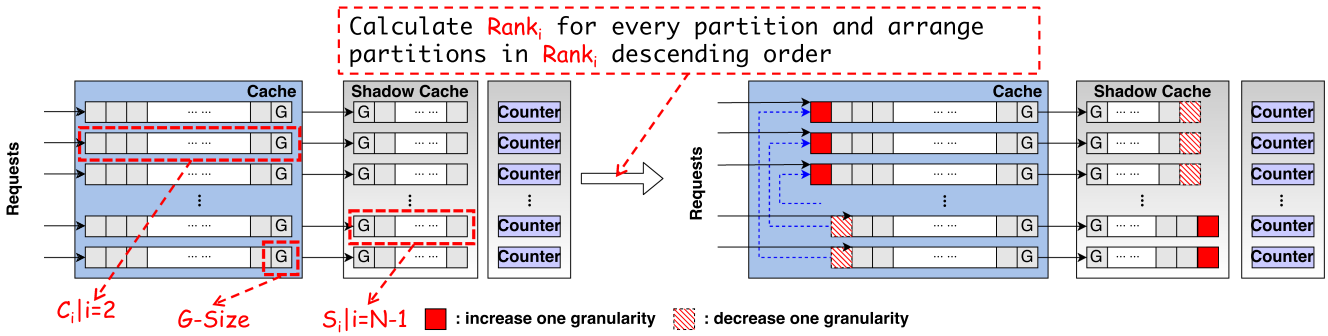


Fig. 6. CDN partition initial state and cache partition after a partition adjustment.

As shown in Fig. 6, each application occupies a cache space resource with a shadow cache. Since the in-memory shadow cache only stores the metadata of the application data, e.g., keys, indexes, etc., the shadow cache consumes a limited amount of memory resources. We set a *Counter*, denoted as cnt_i, which is used for recording the number of hits in the shadow cache. In practice, these *Counters* are also stored in memory, where each counter is a *long int* variable, accounting for 8 $bytes$.

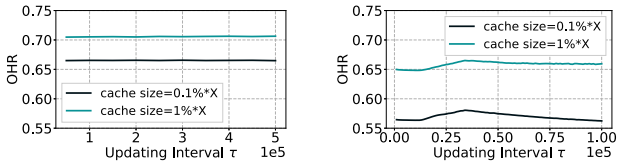
Initially, we allocate the cache space for each application, satisfying (1). For simplicity, we initialize the equal partitions, i.e., allocate $\frac{C}{N}$ cache space for each application. Similarly, the shadow cache allocates the corresponding resources to each application according to (1). We show the concrete process of once partition adjustment in Fig. 6.

$$Rank_i = \frac{cnt_i}{S_i/G}, (i = 1, 2, \ldots, N). \tag{2}$$

and arranges partitions in $Rank_i$ descending order. Then, ε-LAP transfers a capacity of *G-Size* from the $(N-k+1)$-th $(k \leq \frac{N}{2})$ partition into the k-th partition, where the order of partitions is based on the order of $Rank_i$. Note that the transferred *G-Size* is invalid immediately for the original application. For the expanded partition, the space of the *G-Size* will be gradually filled with items when the requests of the corresponding application are missed. In addition, we cut the *G-Size* from the tail of the cache queue to improve performance in the implementation, since the LRU algorithm pushes items that may not be reused to the end of the queue. Once a round of partition adjustment is complete, the values of *Counters* and the number of missed requests reset to 0, and the count function of each *Counter* restarts until the number of missed requests reaches τ.

B. Update Interval τ

The update interval τ directly affects the update frequency of partitioning. The frequent updates will bring a huge overhead. In contrast, the lagging updates cannot well adapt to the dynamic workload under CDNs. To find an appropriate update interval τ, we conducted experiments on the relationship between the update interval and the change of OHR on two traces. For the *Trace-Akamai* [39], as shown in Fig. 7(a), when the cache size equals 4×10^{-3} TB (the blue curve), the maximum hit ratio corresponds to $\tau = 3 \times 10^5$ and when the cache size reaches



(a) *Trace-Akamai* (b) *Trace-BJ*

Fig. 7. Comparison of ε-LAP performance at different update interval τ. Note that X equals 4 TB for *Trace-Akamai* and X is 640 GB for *Trace-BJ*.

0.04 TB (the orange curve), $\tau = 5 \times 10^5$. Similarly, for the *Trace-BJ*, as shown in Fig. 7(b), when the cache size equals 0.0640 GB (the blue curve), the maximum hit ratio corresponds to $\tau = 3.2 \times 10^4$ and when the cache size reaches 6.4 GB (the orange curve), $\tau = 3.3 \times 10^4$.

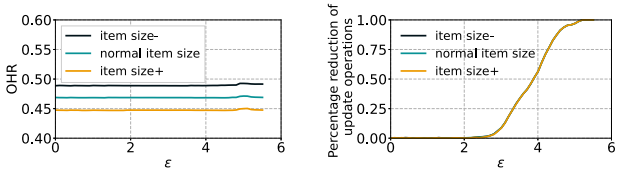
From Fig. 7, we get two conclusions. ① The update interval τ varies for different traces. As a result, setting the update interval needs prior knowledge, such as the average size of the item, the cache size, etc. ② As the cache size increases, the optimal update interval also increases.

C. Make Prudent Resizing Decisions

In the process of resizing partitions, we find that when the difference between $Rank_i$ of *app_i* and $Rank_j$ of *app_j* is small, resizing between these partitions cannot bring benefits but consume computational resources. We set a threshold ε to alleviate this issue. We stipulate that partition resizing only be performed if the difference between $Rank_i$ and $Rank_j$ $(1 \leq i \leq \frac{N}{2}, i = N - j + 1)$ exceeds ε, i.e., satisfying the following relationship:

$$G\left(\frac{cnt_i}{S_i} - \frac{cnt_j}{S_j}\right) > \varepsilon. \tag{3}$$

To determine an appropriate ε, we explore the relationship between ε and OHR. As shown in Fig. 8(a), when $\varepsilon = 4.9$, OHR in the black curve (i.e., small item size) reaches the peak value. For the cyan curve (i.e., normal item size) and the orange curve (i.e., big item size), the corresponding ε equals 5.0 and 5.1, respectively, at the peak. Based on these results, we find that the distribution of item sizes is insensitive to ε. We select $\varepsilon = 5.0$ based on the results with the normal item sizes, i.e.,



(a) OHR (b) Reduction in update times

Fig. 8. The change of OHR and reduction in update times with changing ε under different item sizes on *Trace-BJ* at 40 GB cache size. Note that when $\varepsilon = 0$, all partitions need to be updated. Three lines overlap in (b).

TABLE III
LEARNING ε FROM THE PROPORTION

	Trace	Proportion	ε
Training sample	trace#1	[34.22, 8.60, 8.53, 5.60, 4.48, ...]	8.6
	trace#2	[43.53, 7.38, 7.32, 4.81, 3.85, ...]	1.5
	trace#3	[35.68, 24.08, 6.00, 3.94, 3.15, ...]	7.1
	trace#4	[35.66, 24.07, 6.05, 3.94, 3.15, ...]	7.1
	trace#5	[34.94, 23.58, 5.92, 5.88, 3.09, ...]	6.7
	trace#6	[34.68, 23.40, 5.88, 5.83, 3.83, ...]	6.6
	trace#7	[34.44, 23.24, 5.84, 5.79, 3.80, ...]	6.7
	trace#8	[34.42, 23.23, 5.83, 5.79, 3.8, ...]	6.9
	trace#9	[34.38, 23.20, 5.83, 5.78, 3.8, ...]	6.8
Testing sample	trace#10	[33.64, 22.71, 5.70, 5.66, 3.72, ...]	6.6

we resize partitions only when the difference in (3) between the pair applications is greater than 5.0. Under this constraint, we can reduce about 96.8% of updates and save 96.8% of time without a reduction in performance. We show this improvement in Fig. 8(b).

D. Set an Appropriate ε

In the previous section, it was established that ε is effective in reducing unnecessary partition operations. However, determining the appropriate ε for a new workload to achieve the high cache hit ratio can be a challenge. In this section, we explore a method for determining the optimal ε. As shown in (3), ε should be related to the number of hits per unit on the partition, which is closely tied to the distribution of requests. As a result, we investigate the relationship between ε and these applications' request distribution to directly obtain an appropriate ε through the proportion of these applications' requests, thereby making our model more universally applicable and practical.

We use *Trace-BJ* to generate new traces for experiments. Specifically, we remove all the requests belonging to the i-th application from the trace, and the remaining requests from the *trace#i* ($i = 1, 2, \ldots, 10$). Then, we run ε-LAP on these new traces with different ε, and the results are presented in Fig. 9. The trend of OHR in each figure is similar to that in Fig. 8(a). In addition, we record the applications' request proportion and the optimal ε for each trace. The results are shown in Table III. Intuitively, we find that *trace#3* and *trace#4* have almost the same proportion and optimal ε. It makes us wonder if there is a correlation between the proportion of requests for different applications and the optimal ε.

TABLE IV
PREDICTING ε FROM THE PROPORTION OF TRACE#10

Model	Weights	ε
Lasso [40]	[-0.37, 0.30, 2.14, 0., 0., ...]	**6.6099**
Ridge [41]	[-0.10, 0.09, 0.32, -0.01, -0., ...]	7.6263
Poisson [42]	[-0.76, -0.03, 0.31, -0.49, -0.33, ...]	7.5578

The bolding and underlining is to highlight the best value.

As a result, we use the statistics in Table III as samples for regression training. Several regression methods (i.e., Lasso, Ridge, and Poisson regression) were tested. We select the *trace#10* as the test sample, where the proportion is [33.64, 22.71, 5.70, 5.66, 3.72, 2.97, 2.30, 2.25, 2.12, 2.04, 1.48, 1.06, 0.99, 0.83, 0.57, 0.53, 0.53]. The prediction results using different regression methods are shown in Table IV. As shown in Table III, the prediction result of the Lasso model is the closest to the actual optimal result according to the actual effect running on *trace#10*. In addition, based on the weights of the Lasso regression output, it appears that the primary determinants of ε are the first three applications, which have the greatest share. The share of subsequent applications does not seem to heavily influence the value of ε. By implementing the ε mapping solely on the proportion of requests of the top three applications, we can improve computational efficiency while reducing computational overhead.

VI. IMPLEMENTATION AND IMPROVEMENT ON *PICCLOUD*

We have deployed ε-LAP at the OC layer of *PicCloud* using the C++ library since most requests concentrate on the OC layer, which allows ε-LAP to operate more efficiently. In addition, we place the indexes in the memory. Although setting indexes on SSDs can save memory space, frequent indexing operations may reduce the life of SSDs.

Through the monitoring system, we gathered the metrics (OHR and latency) for one month, where the one-month time span includes the week before the deployment of ε-LAP and about three weeks after the completion of the deployment. Fig. 10 shows the change in OHR and average access latency for a month, where the daily average OHR and access latency are calculated at a granularity of one day from the monitoring system. We can see that the OHR of PicCloud increased by 9.34% on average, and the average access latency dropped by 12.5 ms, albeit these results may be biased due to some errors in the monitoring system. Sincerely, ε-LAP improved the performance of PicCloud across the board.

In addition, deploying ε-LAP reduces inevitable overhead since the appropriate cache resources can be automatically allocated to applications corresponding to the hotspots when the hot data bursts. With the original method, Company-T would respond to a large amount of hot data by expanding the cache capacity. When the hotspots calm down, extra devices need to be manually removed to decrease hardware costs. However, this increases the costs related to O&M employees. ε-LAP's dynamic and self-adaptive adjustment mechanism is necessary for CDNs with large cache sizes. On the one hand, it reduces labor costs. On another hand, the total cache space of PicCloud

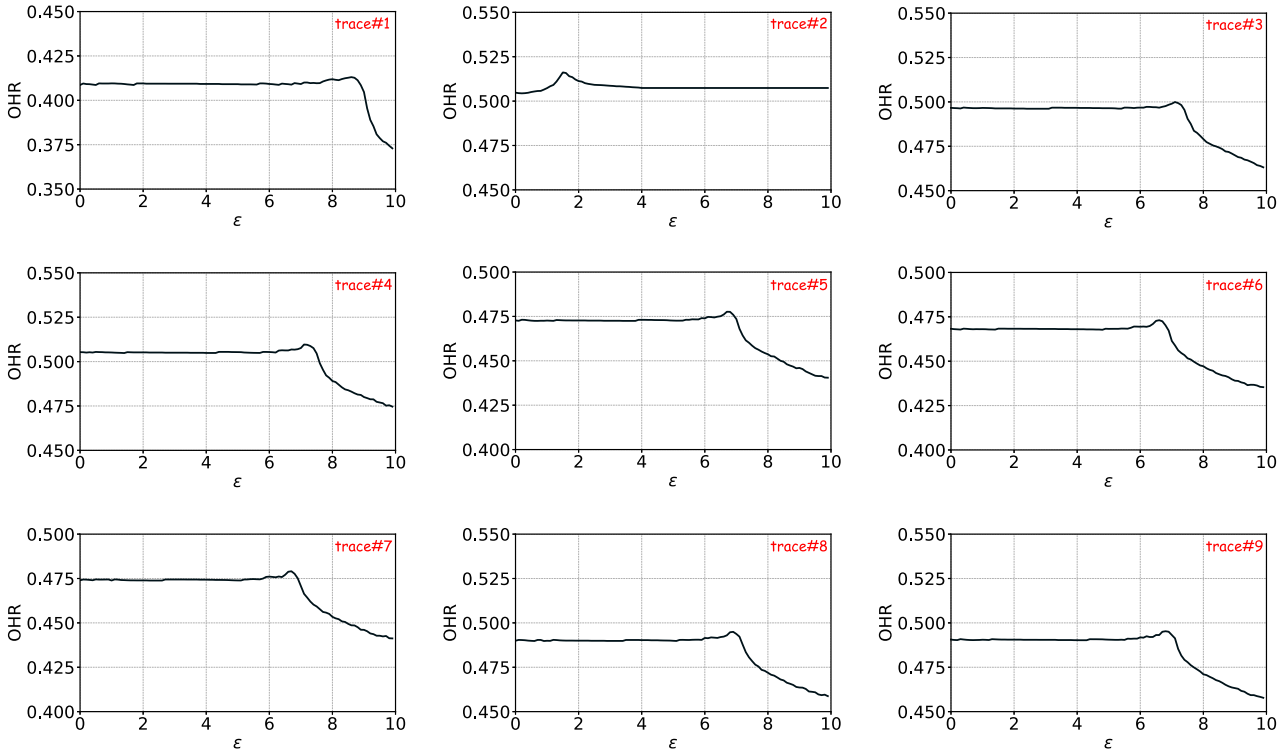
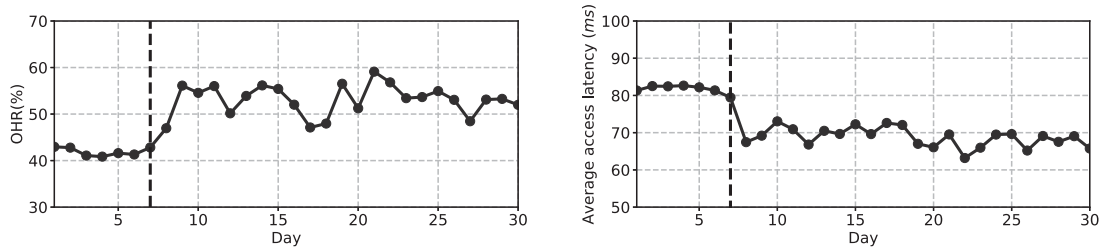


Fig. 9. The hit ratio varies with ε for different traces (different application request proportions).



(a) OHR (b) Latency

Fig. 10. OHR and latency in the monitoring system for a continuous month. Note that the ε-LAP algorithm was deployed online at 24:00 on day-7.

has reached 5000 TB. Increasing the cache size by 100 TB can only bring a 1% improvement for OHR. The hardware cost is higher than the computational cost of ε-LAP for the same performance improvement. As a result, the deployment of ε-LAP without additional equipment is a better solution.

VII. Evaluation

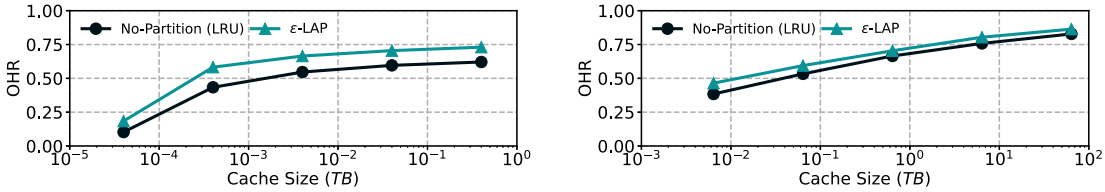
In this section, we show the performance of ε-LAP and analyze its overhead. We compare ε-LAP to the state-of-the-art cache partitioning schemes over two traces. Note that all codes and traces in Table V will be open source.

A. Settings and Simulator

Traces and G-Size: We use two CDN traces, including a public trace, i.e., *Trace-Akamai* [39] and a real-world trace, i.e., *Trace-BJ* [43] that is collected from *PicCloud* of *Tencent*. Their detailed information is shown in Table V, where the working set size means the size of all unique objects [44]. Note that the cache size should align with the working set size of the trace since the cache size is a parameter based on resource availability and system requirements. Based on the working set size shown in Table V and the 8:2 rule, we set cache sizes used to test on the

TABLE V
Summary of the Two Traces That are Used Throughout Our Evaluation

	Trace-Akamai	Trace-BJ
Total Requests (Millions)	81.04	110.42
Unique Items (Millions)	17.18	44.98
Total Bytes Requested (TB)	71.6	8.0
Unique Bytes Requested (TB)	26.0	3.3
Warmup Requests (Millions)	60	80
Working Set Size (TB)	16	2.5



(a) *Trace-Akamai* (b) *Trace-BJ*

Fig. 11. OHR on different traces with different cache sizes.

TABLE VI
RESOURCE USAGE FOR NO-PARTITION(LRU) AND OURS ON THE SIMULATOR OF PICCLOUD

Metric	No-Partition(LRU)	Ours	UCP-L
Peak CPU(%)	5.9	6.2	12.9
Peak Memory(GB)	0.94	0.99	1.25
Update Time (ms)	-	0.38	160.4

simulator as 4 TB and 640 GB for *Trace-Akamai* and *Trace-BJ*, respectively. Based on the size distribution of the sampled data from each trace, we assign the size of the data greater than 99% of data to *G-Size*. As a result, we set $G = 32$ MB for *Trace-Akamai* and $G = 20$ MB for *Trace-BJ*.

Sampling Method: Since the two traces are too large to be loaded in the simulator, we use the reservoir sampling method [45] to select a subset to achieve evaluation. Specifically, we extract all unique photo IDs and sample out 1 of the IDs to form an ID set. Based on this ID set, we obtain our training data by sequential extraction.

State-of-the-Art Cache Partitioning Schemes: As shown in Table I, PriSM [8], OSCA [4], APAC [14], and ACTION [15] require that item sizes be fixed and uniform and the performance of SHARDS [9] and EMB [10] is directly affected by partition granularity P. Thus, We only compare ε-LAP with UCP [7], KPart [11], LPCA [13], LACA [16], and No-partition (i.e., LRU). UCP is the classical partitioning scheme. KPart is the representative partitioning scheme based on MRC. LPCA is a typical partitioning scheme combined with machine learning. LACA is a recent partitioning scheme based on the heuristic method.

Warmup Trace: Every experiment goes through a warmup period, where the warmup period is determined by the period in which every algorithm has achieved a stable performance. The results in the warmup period are not recorded in Table VI. The warmup period is longer than the validation period.

Simulator and Testbed: We deploy the simulator based on *webcachesim* [20]. We implement ε-LAP, UCP, KPart, LACA, LPCA, and LRU on this simulator. To ensure fair comparisons, we evaluate the above cache partition schemes on the same server, whose configurations are two 8-core AMD EPYC$^{\text{TM}}$ 7551 CPUs, 32 GB RAM, and 1.5 TB SSD.

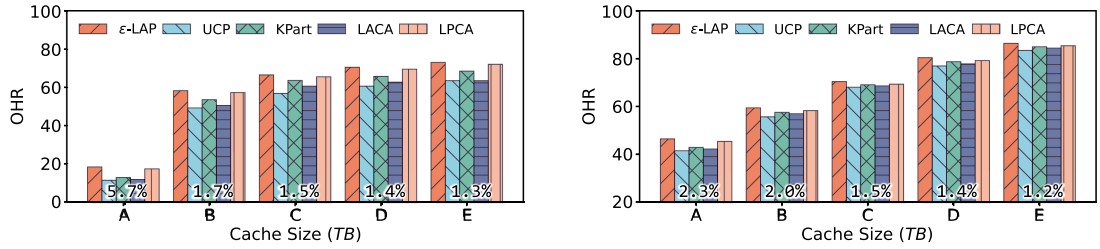
B. Compare ε-LAP With LRU on Simulator

To verify the effect of partitioning in ε-LAP, we compared No-Partition (i.e., LRU) to ε-LAP in terms of OHR on the simulator. As shown in Fig. 11(a), ε-LAP can improve the OHRs by 8.12%, 14.89%, 11.84%, 10.90%, and 10.95% on 4×10^{-5} TB, 4×10^{-4} TB, 4×10^{-3} TB, 0.04 TB, and 0.4 TB, respectively, on Trace-Akamai. Similarly, as shown in Fig. 11(b), ε-LAP can improve the OHRs by 8.03%, 6.11%, 3.87%, 4.56%, and 3.56%, respectively on Trace-BJ. We show the overhead produced by LRU, ε-LAP, and UCP-L, respectively, in Table VI. The peak CPU utilization of ε-LAP is slightly higher than that of LRU and is much lower than that of UCP-L. For the storage overhead, ε-LAP only overtakes LRU by 0.05 GB at 32 GB RAM. In addition, the tiny gap in throughput between LRU and ε-LAP is 1.08×10^{-5} Gbps.

C. ε-LAP versus State-of-the-Art Cache Partition Schemes

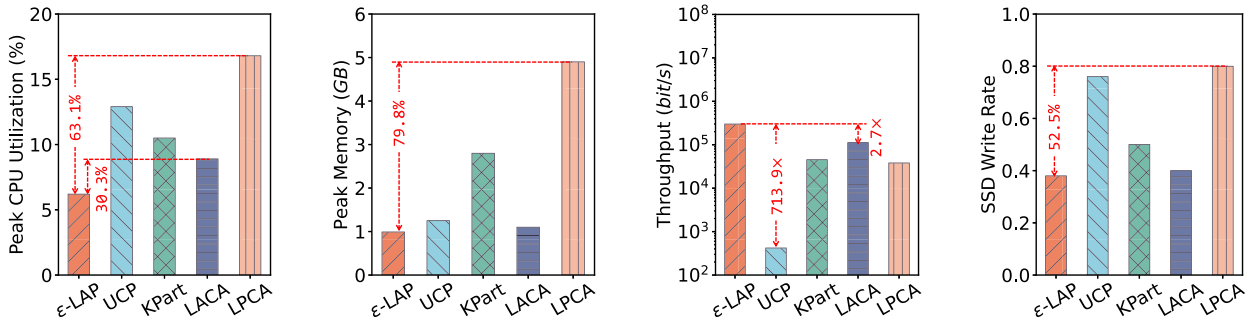
We compare ε-LAP to four state-of-the-art cache partitioning schemes on two traces with a wide range of cache sizes. As shown in Fig. 12(a) and (b), ε-LAP outperforms other cache partitioning schemes in terms of OHR. On *Trace-Akamai*, ε-LAP can outperform LPCA, the best scheme among the four state-of-the-art cache partitioning schemes, by 5.7%, 1.7%, 1.5%, 1.4%, and 1.3% at different cache sizes of 4×10^{-5} TB, 4×10^{-4} TB, 4×10^{-3} TB, 0.04 TB, and 0.4 TB, respectively. Similarly, on *Trace-BJ*, the average improvements are 2.3%, 2.0%, 1.5%, 1.4%, and 1.2%, respectively.

In addition, we compare different schemes in terms of peak CPU utilization, peak memory utilization, throughput, and SSD write rates. As *Trace-Akamai* only includes two applications, we will conduct this experiment solely on *Trace-BJ*. As shown in Fig. 13(a) and (b), in terms of CPU and memory utilization, ε-LAP outperforms the other four partitioning algorithms. LPCA has a high resource consumption rate due to its complex machine-learning MRC model. Compared with the LPCA scheme, ε-LAP saves 63.1% of CPU resources, and compared with the lowest CPU utilization of the four schemes (i.e., LACA), ε-LAP also saves 30.3% of CPU resources. KPart uses extra memory compared to UCP and the memory consumption of ε-LAP is similar to that of LACA. As shown in Fig. 13(c), the throughput of ε-LAP on the simulator is approximately 713.9$\times$ greater than that of UCP and 2.7$\times$ greater than that of LPCA. We attribute the ε-LAP's high throughput to its lightweight design and ε's success in reducing ineffective update operations. The SSD write rate directly affects the lifetime of SSDs. To reduce SSD wear, you need to minimize the SSD write rate. As shown in Fig. 13(d), compared with the other four algorithms, ε-LAP



(a) *Trace-Akamai* (b) *Trace-BJ*

Fig. 12. Performance comparison of several algorithms with different cache sizes on two traces. The numbers represent the improvement on OHR by ε-LAP relative to LPCA. For *Trace-Akamai*, A = 4×10^{-5}, B = 4×10^{-4}, C = 4×10^{-3}, D = 0.04, E = 0.4. For *Trace-BJ*, A = 6.4×10^{-3}, B = 0.064, C = 0.64, D = 6.4, E = 64.



(a) Peak CPU Utilization (b) Peak Memory Utilization (c) Throughput (d) SSD Write Rate

Fig. 13. Comparisons of ε-LAP, UCP, KPart, LACA, and LPCA in peak CPU utilization, peak memory, throughput, and SSD write rate.

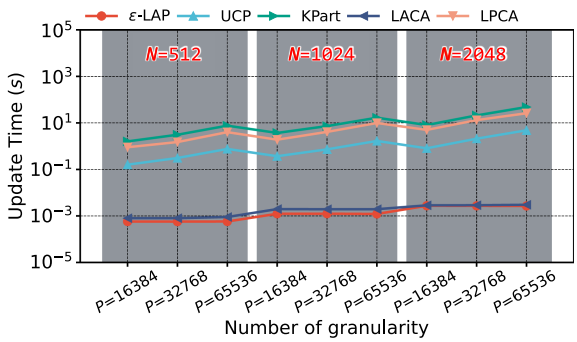


Fig. 14. Under different application numbers (N) and different partition granularity numbers (P), the updating time of several algorithms is compared.

reduces the partition adjustment due to the introduction of ε, thus reducing the SSD write rate.

In CDNs, the partitioning scheme should be able to resize partitions in real time to respond to changes in workload. Since the latency is unavoidable but affects the experience, we will compare the average time various algorithms take to perform an update operation on *Trace-BJ*. As shown in Fig. 14, compared to ε-LAP, the update time of the MRC-based approaches (UCP, KPart, LACA, LPCA) is greater by 2-3 orders of magnitude. As shown in Fig. 14, with N =512 and P =16384, the update time of ε-LAP is 0.576 ms, but the counterparts of UCP, KPart, and LPCA are 0.16 s, 1.7 s, and 0.92 s, respectively. Due to its lightweight design, LACA has an update time of 0.8 ms, which is slightly higher than ε-LAP.

D. Parameters Sensitive Study

As mentioned previously, the performance of ε-LAP can be influenced by various parameters such as *G-Size*, *R-Size*, ε, and the updating interval τ. Each parameter's sensitivity to ε-LAP performance was outlined separately in the preceding section. For a unified display, we'll summarize the performance parameter sensitivity of ε-LAP concerning these four parameters (i.e., *G-Size*, *R-Size*, ε, and τ).

First, as mentioned in Section I, there is a linear relationship between *R-Size* and *G-Size*. Therefore, we discuss only the performance sensitivity of ε-LAP to *G-Size*. As shown in Fig. 4(a), OHR displays a trend of initially increasing, then decreasing with the rise of *G-Size*. If *G-Size* is too small, according to ε-LAP, multiple adjustments are required, significantly reducing its performance. Conversely, if *G-Size* is too large, the partitioning algorithm becomes ineffective, and performance declines. Second, as mentioned in Section V-B and shown in Fig. 7, the update interval τ varies for different traces and the optimal update interval gradually increases as cache size increases. Lastly, we examine the performance sensitivity of ε-LAP to ε in Section V-D. As shown in Fig. 9, with an increase

in ε, OHR initially rises and then falls. As the best ε varies across different traces, it underscores the value of the learning scheme proposed in Section V-D.

Furthermore, when $P = \frac{C}{G}$ increases, the update time of the ε-LAP algorithm does not greatly increase, which is friendly to the large cache and small partitioning granularity. In the same situation, the update time of other algorithms is sensitive to P with dramatic increases. This phenomenon corresponds to Fig. 14 and Table I. In addition, the increase in ε-LAP update time is less significant compared to other MRC-based methods as N increases.

VIII. Conclusion

Since traditional cache algorithms, e.g., replacement algorithms and admission policies, are hard to improve the OHR in CDNs, we resort to cache partitioning schemes. Since the MRC-based schemes are hard to adapt to different item sizes and big cache sizes in CDNs, we have to improve existing methods in terms of the determination of partition resizing for accuracy and efficiency. As a result, we propose a lightweight and adaptive cache partitioning scheme with prudent decisions (ε-LAP) to address the above issue. ε-LAP uses the number of hits on the shadow cache, i.e., $Rank_i$, rather than the MRC curve to trigger resizing, improving execution efficiency and alleviating the performance cliff problem. Furthermore, we set a threshold, i.e., ε, to monitor the difference between a pair of $Rank_i$, reducing unnecessary resizing/update operations. The experimental results validate the efficiency and accuracy of ε-LAP, and it outperforms state-of-the-art methods in terms of OHR, throughput, and update time. As the first partitioning algorithm on CDN, ε-LAP considers the impact of different item sizes in the selection of *G-Size*.

References

[1] Z. Song et al., "Learning relaxed belady for content distribution network caching," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 529–544.

[2] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "DRLPart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *Proc. 30th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2021, pp. 175–188.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM Sigmetrics/Perform. Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.

[4] Y. Zhang et al., "OSCA: An online-model based cache allocation scheme in cloud block storage systems," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 785–798.

[5] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *Proc. 13th ACM Workshop Hot Top. Netw.*, 2014, pp. 1–7.

[6] Z. Liu et al., "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 143–157.

[7] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 423–432.

[8] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. 39th Annu. Int. Symp. Comput. Architecture*, 2012, pp. 428–439.

[9] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 95–110.

[10] C. Ye, J. Brock, C. Ding, and H. Jin, "Rochester elastic cache utility (RECU): Unequal cache sharing is good economics," *Int. J. Parallel Program.*, vol. 45, no. 1, pp. 30–44, 2017.

[11] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 104–117.

[12] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic cache allocation with partial sharing," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.

[13] Y. Gu et al., "LPCA: Learned MRC profiling based cache allocation for file storage systems," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, 2022, pp. 511–516.

[14] R. Li et al., "Accurate probabilistic miss ratio curve approximation for adaptive cache allocation in block storage systems," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2022, pp. 1197–1202.

[15] C. S. Mummidi and S. Kundu, "ACTION: Adaptive cache block migration in distributed cache architectures," *ACM Trans. Architecture Code Optim.*, vol. 20, no. 2, pp. 1–19, 2023.

[16] K. Liu, H. Wang, K. Zhou, and C. Li, "A lightweight and adaptive cache allocation scheme for content delivery networks," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2023, pp. 1–6.

[17] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.

[18] P. Wang, Y. Liu, Z. Zhao, K. Zhou, Z. Huang, and Y. Chen, "Smart cache insertion and promotion policy for content delivery networks," in *Proc. 52nd Int. Conf. Parallel Process.*, 2023, pp. 183–192.

[19] L. V. Rodriguez et al., "Learning cache replacement with CACHEUS," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 341–354.

[20] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "AdaptSize: Orchestrating the hot object memory cache in a content delivery network," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 483–498.

[21] H. Wang, J. Zhang, P. Huang, X. Yi, B. Cheng, and K. Zhou, "Cache what you need to cache: Reducing write traffic in cloud cache via "one-time-access-exclusion" policy," *ACM Trans. Storage*, vol. 16, no. 3, pp. 1–24, 2020.

[22] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 351–364.

[23] D. Byrne, N. Onder, and Z. Wang, "mPart: Miss-ratio curve guided partitioning in key-value stores," in *Proc. ACM SIGPLAN Int. Symp. Memory Manage.*, 2018, pp. 84–95.

[24] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *Proc. 7th USENIX Workshop Hot Top. Cloud Comput.*, 2015, Art. no. 19.

[25] Z. Jiang, K. Yang, N. Fisher, N. Guan, N. C. Audsley, and Z. Dong, "Hopscotch: A hardware-software co-design for efficient cache resizing on multi-core SoCs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 1, pp. 89–104, Jan. 2024.

[26] Z. Liu, H. W. Lee, Y. Xiang, D. Grunwald, and S. Ha, "eMRC: Efficient miss ratio approximation for multi-tier caching," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 293–306.

[27] X. Hu et al., "Fast miss ratio curve modeling for storage cache," *ACM Trans. Storage*, vol. 14, no. 2, pp. 1–34, 2018.

[28] K. Shakiba, S. Sultan, and M. Stumm, "Kosmo: Efficient online miss ratio curve generation for eviction policy evaluation," in *Proc. 22nd USENIX Conf. File Storage Technol.*, 2024, pp. 89–105.

[29] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2016, pp. 225–236.

[30] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 487–498.

[31] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield, "Characterizing storage workloads with counter stacks," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 335–349.

[32] R. Gu et al., "Adaptive online cache capacity optimization via lightweight working set size estimation at scale," in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 467–484.

[33] D. Carra and G. Neglia, "Efficient miss ratio curve computation for heterogeneous content popularity," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 741–751.

[34] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations," *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 121–132, 2009.

[35] J. Park, S. Park, and W. Baek, "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.
[36] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.
[37] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Architecture*, 2015, pp. 450–462.
[38] Y. Li et al., "Reinforcement learning-based resource partitioning for improving responsiveness in cloud gaming," *IEEE Trans. Comput.*, vol. 71, no. 5, pp. 1049–1062, May 2022.
[39] A. Sabnis and R. K. Sitaraman, "TRAGEN: A synthetic trace generator for realistic cache simulations," in *Proc. 21st ACM Internet Meas. Conf.*, 2021, pp. 366–379.
[40] R. Tibshirani, "Regression shrinkage and selection via the lasso," *J. Roy. Statist. Soc. Ser. B Methodol.*, vol. 58, no. 1, pp. 267–288, 1996.
[41] D. E. Hilt and D. W. Seegrist, "Ridge: A computer program for calculating ridge regression estimates," Research Note NE-236, U.S. Dept. Agriculture, Forest Service, Northeastern Forest Experiment Station, Upper Darby, PA, USA, p. 7, 1977.
[42] R. Paternoster and R. Brame, "Multiple routes to delinquency? A test of developmental and general theories of crime," *Criminology*, vol. 35, no. 1, pp. 49–84, 1997.
[43] P. Wang, Z. Zhao, Y. Liu, K. Zhou, Z. Huang, and Y. Chen, "A lightweight and adaptive cache partitioning scheme for content delivery networks," in *Proc. IEEE 40th Int. Conf. Comput. Des.*, 2022, pp. 407–410.
[44] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 191–208.
[45] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

Peng Wang is currently working toward the PhD degree with the Wuhan National Laboratory for Optoelectronics and School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. He has published papers in international journals and conferences, including ICCD, ICPP, IJCAI, the *ACM Transactions on Architecture and Code Optimization*, etc.

Yu Liu (Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST) in 2017. Then, he was a postdoctoral researcher with HUST from 2018 to 2021. Currently, he is an associate researcher with the School of Computer Science and Technology, HUST. His current research focuses on building cognitive storage systems with machine learning technologies and large-scale multimedia search technologies. He has published papers in international journals and conferences, including SIGMOD, DAC, IJCAI, MM, CIKM, ICME, ICPP, ICCD, the *IEEE Transactions on Image Processing*, *IEEE Transactions on Cybernetics*, *IEEE Transactions on Multimedia*, *ACM Transactions on Multimedia Computing, Communications, and Applications*, etc.

Ziqi Liu is currently working toward the bachelor's degree in computer science and technology with the Huazhong University of Science and Technology, Wuhan, China, where he will continue his studies to pursue a master's degree.

Zhelong Zhao received the BE degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2022, where he is currently working toward the master degree. He has published papers in ICCD, ICPP, etc.

Ke Liu is currently working toward the PhD degree with the Wuhan National Laboratory for Optoelectronics and School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. He has published papers in IPDPS, DATE, the *ACM Transactions on Architecture and Code Optimization*, etc.

Ke Zhou (Member, IEEE) is a professor with the School of Computer Science and Technology, HUST. His research interests include computer architecture, cloud storage, parallel I/O, and storage security. He has more than 50 publications in journals and international conferences, including the *IEEE Transactions on Parallel and Distributed Systems*, the A Page Endurance Variance Aware, SIGMOD, FAST, USENIX ATC, MSST, ACM MM, INFOCOM, SYSTOR, MASCOTS, ICC, etc. He is a member of the USENIX.

Zhihai Huang received the postgraduate degree from the Nanjing University of Posts and Telecommunications, China. He is currently an expert engineer with Tencent Corporation. Since joining Tencent, he has specialized in optimizing the storage, delivery, and processing of massive social media data. His main research interests include media storage, image processing, video transcoding, and real-time communications.