2023 Design, Automation & Test in Europe Conference (DATE 2023)

A Lightweight and Adaptive Cache Allocation Scheme for Content Delivery Networks

Ke Liu†, Hua Wang†*, Ke Zhou†,Cong Li‡
†Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
‡Tencent Technology (Shenzhen) Co., Ltd., Shenzhen, China
(liu_ke, hwang, zhke)@hust.edu.com, clusterli@tencent.com, *Corresponding author

***Abstract*—Content delivery networks (CDNs) caching systems usually use multi-tenant shared caching due to their operational simplicity. However, this approach often results in interference among applications. Dynamic cache allocation schemes based on miss ratio curve (MRC) could be a good choice except for its high computational overheads and performance fluctuations. In this paper, we propose a lightweight and adaptive cache allocation scheme for CDNs (LACA). Rather than searching near-optimal configurations for each tenant, LACA detects in real time whether any tenants are using cache space inefficiently (named abnormal tenants), and then adjusts space restricted within these abnormal tenants by constructing their local MRCs instead of the global ones. We have deployed LACA in Tencent's CDN system and LACA can reduce the miss ratio by 27.1% and reduce the average user access latency by 28.5 ms. Compared with the-state-of-the-art schemes, LACA also achieves a higher-accuracy local MRC with marginal overhead.**

***Index Terms*—Content Delivery Networks, Cache Allocation, Local Miss Ratio Curve**

I. Introduction

A content delivery network (CDN), is a large distributed system with hundreds of thousands of servers deployed around the world [5], [13]. The servers are grouped into clusters, where each cluster is deployed within a data center on the edge of the Internet. The goal is to provide high availability and performance by distributing the service spatially relative to end users. Unlike storage systems, CDN servers do not store the original content copies, and when a request miss occurs, the content is retrieved from another CDN cluster or the origin servers operated by the content provider [20].

In order to facilitate management and make full use of storage resources and CPU resources, CDNs currently adopt a multi-tenant hybrid storage model. All tenants use naturally competing cache spaces for uniform storage. In addition, CDN clusters use consistent hashing for storage [10], that is, it is uncertain which cache server in the cluster the data of each tenant is stored on. However, multi-tenant servers have the added challenge of ensuring that each tenant cache meets its performance goals; a range of production and research multi-tenant caches currently provide different sharing policies, such as enforcing a limit on the used storage capacity and bandwidth [1], guaranteeing a level of quality-of-service (QoS) [2], and allocating resources proportionately [14], [22].

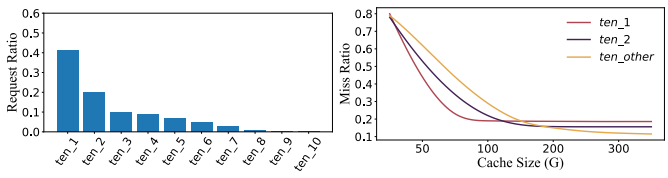
Unfortunately, existing multi-tenant cache sharing policies are not readily use to CDNs. On the one hand, static allocating resources is often suboptimal for the cloud environment and induces resource wastage, because the cloud I/O workloads are commonly highly-skewed [3], [7], [9]. On the other hand, consistent hashing storage and multi-tenant cache mode increase the operational complexity of a customized allocation scheme (the more machines in the cluster, the more complex the operation). What's more, the current cache space allocation scheme is a time-consuming operation.

In this paper, we aim to address the management of cache resources shared by multiple instances of a CDN cache system. We propose LACA: a lightweight dynamic cache allocation model with local miss ration curves (MRC). LACA does not require to construct all tenants MRCs and does not require to construct a complete MRC. LACA searches for a near-optimal configuration scheme at a very low complexity and thus improves the overall effectiveness of the cache server. Specifically, the core idea of LACA is three-fold. First, only tenants who are using storage resources inefficiently can be adjusted. Second, LACA uses a lightweight machine learning method to construct the local MRCs. Third, LACA searches for an optimal resize using dynamic programming methods from the local MRCs.

As the key contribution, unlike the conventional construct-all-tenants MRCs, LACA simply detects in real time whether any tenants are using cache space inefficiently on the cache server and only calculates the elimination of space from the tenants that are using storage resources inefficiently. In addition, LACA uses a machine learning method to construct local MRCs, and compared with previous cache allocation methods, LACA is more lightweight and we shift the cost of processing I/O traces to that of using machine learning method to construct the local MRCs of the inefficient use of storage resource tenants. Furthermore, local MRC also reduces the search scope of the dynamic programming method. Experimental results show that LACA achieves better results in terms of hit rate and latency. In a real system, LACA can reduce the miss rate by 27.1% and reduce the average user access latency by 28.5ms. In a simulated environment, tested on different traces, LACA is more accurate in the majority of cases in the constructed local MRC.

II. Background and Motivation

A. CDN Architecture and Challenges



(a) Request Distribution (b) Miss Ratio Carves

Fig. 1. (a) The request frequencies of top 10 tenants in CDN. (b) Miss Ratio Carve (MRC) of ten_1, ten_2 and ten_other. The ten_other includes other 8 tenants and their requests are executed as if they were a tenant.

978-3-9819263-7-8/DATE23/© 2023 EDAA

TABLE I
EXISTING WELL-KNOWN CACHE ALLOCATION SCHEMES AND OUR SCHEME. N DENOTES THE NUMBER OF TENANTS. C REPRESENTS THE SIZE OF THE WHOLE CACHE. THE DATA OF OURS ARE FROM OUR PROPOSED SCHEME.

Method	Criteria	Memory Overhead	Runtime	Improvement
Original	sharing cache	-	-	-
ORPS [17]	sharing cache	N	0.1ms	-4%-2%
PriSM [11]	static allocation	$N \times C$	1ms	-2%-3%
Pelikan [2]	static allocation	$N \times C$	5ms	-7%-6%
SHARDS [19]	dynamic allocation	$N \times C$	90s	-1%-9%
Mini-Sim [18]	dynamic allocation	$N \times C$	300s	-1%-11%
OSCA [22]	dynamic allocation	$N \times C$	40s	1%-12%
APAC [8]	dynamic allocation	$N \times C$	37s	1%-13%
Ours	Combination of the three	$\leq(N \times C)$	73ms	6%-27%

Content Delivery Network (CDN) Architecture. A CDN is a large distributed system with hundreds of thousands of servers deployed around the world [5], [13]. The servers are grouped into clusters, where each cluster is deployed within a data center on the edge of the Internet. When a user requests an object, the global load balancer of the CDN routes the request to a cluster that is proximal to the user [4]. Next, the local load balancer within the cluster routes the request to one or more servers within the chosen cluster that can serve the requested object. A consistent hash algorithm is used to store data within the cluster [10] and for better system utilization and easy to operate, CDN providers are not differentiated by tenant, all tenants are stored according to a uniform hash algorithm. Each server performs cache replacement individually, and each server has relatively limited storage resources. When the cache space of a sever is full, the cache of this sever needs to be replaced. This type of storage allows each tenant to have different occupancy on different servers, and there is no way to predict in advance how many files a tenant will need to store on a particular server.

Why cache allocation in CDNs? A CDN aims to serve content faster than a tenant's origin by a specified speedup factor. The operating cost of CDN mainly includes bandwidth and hardware replacements and the hardware replacements cost mainly driven by SSDs replacement [20]. The CDN providers charge for access bandwidth, so some tenants use the CDN nodes as storage nodes to save costs, storing some files that do not have much traffic but need a lot of storage space (for example, some files have a certain access cycle rule). Furthermore, large traffic tenants compete for too much space, resulting in a portion of the cache space being used inefficiently, and small traffic tenants fail to compete for space, resulting in a high miss rate. It makes the overall hit rate and latency not optimal. We display the requests distribution and Miss Ratio Carve (MRC) from Tencent's CDN using the LRU algorithm in Fig 1. As shown in Fig 1(a), the requests of ten_1 and ten_2 far exceed the counterparts of other tenants. It means that most of the cache will be filled with the content of ten_1 and ten_2, and other tenants will compete for the little space available. As shown in the MRC of Fig 1(b), ten_1 can only fully utilize 80G of cache resources and there are more overall benefits at a bigger cache size when only other tenants share the cache. Therefore, it is necessary to include cache allocation strategies both from the user experience and cost-saving perspective.

B. Cache Allocation Scheme

The current cache space allocation schemes mainly include global sharing policy, static allocation policy and dynamic allocation policy. The sharing policy is currently used by CDN because it is simple to operate and easy to deploy. However, some aggressive data streams may exhaust most of the cache space, making shared policies potentially disruptive to performance [12], [17]. The static allocation policy statically divides cache resource across streams to deal with performance interference problem. Static allocation policy statically allocates cache resources to different data streams according to certain rules (e.g. QoS, bandwidth) to handle performance interference. However, this statically partitioning policy may underutilize the valuable cache resources, because the access patterns of workloads keep changing during runtime [9], [21]. The mainstream dynamic cache allocation schemes are based on MRC to obtain the near-optimal cache space allocation. However, constructing the MRC and obtaining a near-optimal configuration is a very time-consuming operation, and it is impossible to adjust the cache space in real time. A more comprehensive comparison is shown in Table I, including the memory overhead, time cost and hit rate improvement on the Original basis(values are the average of the CDN-A1 traces (mentioned in Sec IV-A)). Original represents a natural competitive cache without any restrictions. Note that this is the time it takes to generate an MRC for a tenant, and the more tenants, the longer it takes.

In this paper, based on the above three schemes, we propose a rule-based dynamic space adjustment strategy for shared caches. This scheme can be quickly deployed on CDN cache servers and can dynamically adjust the cache space in real time. This is done by not changing the CDN's shared caching approach and determining whether a tenant is "overloading" the cache space based on the collected workload and the current access patterns of each tenant. If the tenant is "over" occupying cache space, we construct a local MRC for the tenant and then calculate how much cache space the tenant needs to replace over time. Through extensive experiments, we found that the frequent space adjustments are basically in the top 10 tenants, so the memory overhead and time cost are reduced significantly. In addition, our scheme searches for the required adjustment space only on the local MRCs, so the time cost is significantly reduced.

III. *LACA* DESIGN AND IMPLEMENTATION

A. Design Overview

As shown in Fig 2, LACA performs four steps: information collector (cluster information collector), anomaly detector, model generator, and spatial scheduler. The cluster information collector mainly collects the status information of each server in the cluster. Based on the status information collected, the anomaly detector determines whether each server has tenants that are occupying space abnormally(mentioned in Sec III-B) and gives an indication of which tenants need to be adjusted,

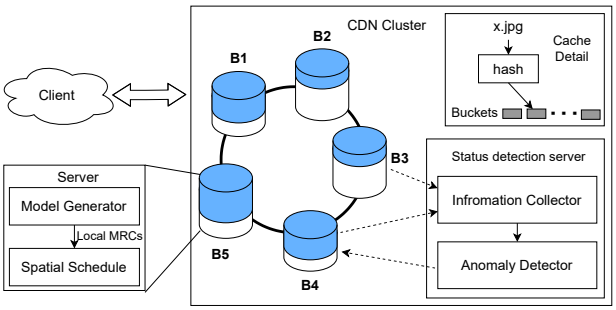


Fig. 2. **LACA Architecture.** An simple architecture of CDN system and cache allocation system. Information collector mainly collects the status information of each server in the cluster. Anomaly Detector detects if there are tenant status anomalies, then Model Generator generates local MRCs for anomalous tenants. Finally, Spatial Scheduler calculates the space to be adjusted from the exception tenant. The blue part on each bucket represents the distribution of a particular tenant in the cluster.

TABLE II
INFORMATION COLLECTED BY THE INFORMATION COLLECTOR. ALL INFORMATION IS OVER A PERIOD OF TIME.

Symbol	Definition
A_s	total accesses of the server
A_t	total accesses of the tenant
H_s	the hit rate of the server
H_t	the hit rate of the tenant
S_t	cache space occupied by the tenant
S_n	the size of objects written to the cache over a period of time
S_9	space occupied by 90% of the tenant's accesses
O_t	the number of objects of the tenant
T_t	the access traffic of the tenant
U_t	unique objects of the tenant

with the results returned to that server. After the server receives the exception results, the model generator generates the local MRCs of the space tenants to be adjusted according to the feedback information, and the spatial scheduler adjusts the corresponding space according to the local MRCs and then prioritizes the replacement of these tenants when there are new objects to be stored. Next, we introduce the four components in detail.

Information Collector. The information collector is collecting information from each tenant in the cluster on each cache server. Each server needs to report the information collected at intervals to provide the anomaly detector to analyze whether the space occupation is abnormal. The interval time can be freely set, the fastest setting can be 1 minute and we used it in our experiments to collect information every ten minutes. The information collected is presented in TABLE II.

Anomaly Detector. The anomaly detector is to detect if a tenant is overusing storage space, and we use a decision tree to intelligently detect if a tenant is overusing. Compared with static allocation policy, Anomaly Detector can determine whether a tenant is over-occupying cache space based on real-time load information. Moreover, Anomaly Detector integrates the state information used in the existing static allocation policy study, and the accuracy of the determination is higher.

Model Generator. The model generator mainly generates local MRCs for anomalous tenants. The space size setting of the local MRC is determined by the time between information collection. The space size of the local MRC is set to [c-x, c],

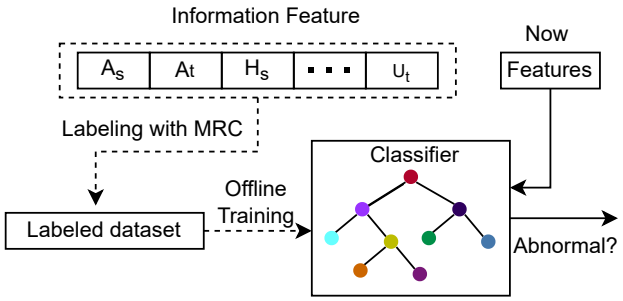


Fig. 3. Anomaly Detector Architecture.

c represents the cache space currently occupied by the tenant and x represents the amount of data written to the cache by the cache server over a period of time. Local MRC generation using Gradient Boosting Machines [6] (GBM) method.

Spatial Scheduler. The spatial scheduler adjusts the size of the space (x) occupied by the newly generated objects over a period of time according to the local MRC. We define the space size for each exception tenant adjustment as (C_1, C_2...C_i), i is the number of abnormal tenants. The adjusted hit rate for these tenants is reduced by (R_1, R_2...R_i) respectively. In order to achieve the optimal adjustment effect, a minimum hit rate reduction is required after the adjustment. Equation (1)) represents the target of adjustment. Equation (2)) represents the limit of the space adjusted by each abnormal tenant. Finally, we use dynamic programming to find the size of the space to be adjusted for each anomalous tenant from the local MRC. When there are new objects to be written to the cache, replacement operations from these tenants are given priority.

$$min(\sum_{j=0}^{i} R_j) \tag{1}$$

$$C_x = \sum_{j=0}^{i} C_j \tag{2}$$

B. Abnormal State Detection

The most critical step in the effectiveness of LACA is the ability to accurately find tenants who are over-occupying cache space. The traditional approach is to construct an MRC for each tenant and then see if the tenant is overusing cache space. Obviously this approach is costly and not suitable for real-time detection. There are also heuristics to detect tenants inefficient use of resources, but these methods are only very accurate for certain workload scenarios and are not applicable to dynamic and variable workloads. By analyzing the construct MRC methods and heuristic methods, we find that determining whether a tenant is over-occupying space is primarily determined by the information in TABLE II, and as the workload changes, it is only necessary to adjust the importance of these characteristics to accurately analyze whether the tenant is using storage space inefficiently. Recall[1] is an important metric in our design, i.e.,

[1]Recall refers to the proportion of correct predictions that are positive to all actuals that are positive.

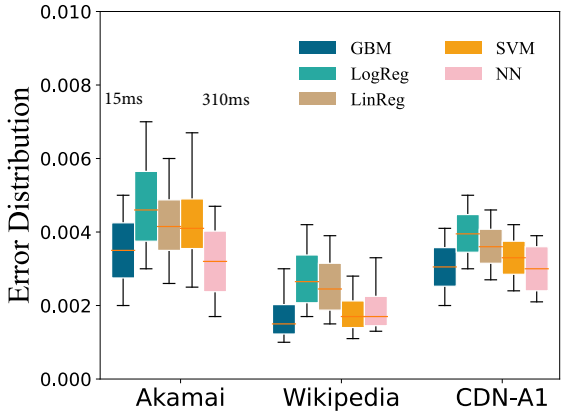


Fig. 4. The MAE distribution of different models. Model Generator chooses GBM as it robustly achieves low MAE on all traces. 15ms and 310ms represent the time spent using GBM and NN respectively.

it is desirable that all anomalous tenants are detected, even if some tenants that are not anomalous are detected by mistake, which will not have a significant impact on the final allocation scheme (it just increases the computational overhead). Through testing we found that the recall rates using decision trees on the Akamai, Wikipedia, Trace-A traces (mentioned in Sec IV-A) reached 98.6%, 99.1% and 100% respectively. Therefore, we use decision trees to analyze in real time whether a tenant is over-occupying storage space.

Fig 3 shows the architecture of the anomaly detector. The architecture includes offline training and online prediction. The most important aspect of offline training is labeling the dataset. We use the MRC to label the dataset. Whether a sample overuses the cache space or not is mainly determined by the threshold value of the MRC slope. The threshold value can be freely adjusted according to the actual situation. If the total cache space is large and the number of tenants is relatively small, the threshold value can be set higher accordingly. In our experiments, the threshold is set to $-\frac{0.02}{C}$, C denotes the current cache size. The threshold value indicates that reducing the miss rate by 1% requires $\frac{1}{2}$ of the current cache space.

C. Construct Local MRCs

To construct a local MRC, in addition to some basic information of tenants in TABLE II, some special features need to be used. By analyzing the features used in the previous construction of MRCs and the features used in the caching algorithm, we extracted 5 special features which are described as follows:

- **Frequency.** The number of total accesses to a same piece of object in the full trace.
- **Reuse time (RT).** The amount of requests between two consecutive references to the same object.
- **Once-Access Traffic Ratio.** The ratio of the size of objects that have been accessed only once to the total traffic.
- **Once-Access Object Ratio.** The ratio of the number of objects that have been accessed only once to the total number of objects.

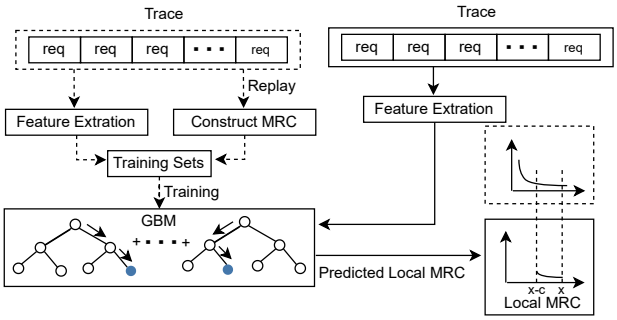


Fig. 5. Model Generator Architecture. The final generation is a local MRC (solid line box part). Note that the model training is to train a set of regression model parameters for each spatial point, while the prediction is only loaded with the parameters of the desired point location for prediction.

- **Local Re-Accesses.** The number of remaining identical objects between two consecutive references to the same object.

The accuracy of constructing a local MRC is the guarantee that determines the validity of the final adjustment space. Model Generator uses GBM to construct the local MRCs, which outperform all other models we explored and are highly efficient on CPUs. We also explored linear regression, logistic regression, support-vector machines, and a shallow neural network with 2 layers and 125 hidden nodes. Fig 4 shows the mean absolute error (MAE) between the approximate and exact MRCs of the different models on three traces across several different cache sizes. Results on the other traces are similar and not shown. GBM robustly achieve low MAE. Additionally, GBM do not require feature normalization and can handle missing values efficiently. In addition, GBM are highly efficient to train and use for prediction. On typical CDN server hardware, we can train our model in 10s. And, we can run prediction on 10 cache value points in 3ms.

As shown in Fig 5, the Model Generator uses GBM to generate local MRCs. To reduce the computational overhead and narrow down the exploration, we end up generating a local MRC (solid line box part). The miss rates under different cache spaces are predicted by a series of feature information, and then the local MRC is fitted by these miss rates. Therefore, the most critical aspect of constructing a local MRC is not only the choice of the expected prediction model but also the selection of the cache space values.

The MRC is constructed by fitting the miss rate to different cache space points. Therefore, it is important for the selection of cache space values, especially for the construction of local MRCs (the small range of cache space values can easily lead to fewer or even no selected points). Therefore, to ensure that there are 10 optional values in the interval time period (mentioned in Sec III-A), we finally use $\frac{C_t}{10}$ as the interval value for cache space selection in the training model. C_t represents the amount of cache writes during the selected interval.

D. Spatial Scheduler

Finding a near-optimal configuration from the MRCs in cache space allocation is also a more time-consuming operation, especially when there are many tenants. LACA reduces

TABLE III
SUMMARY OF THE THREE TRACES THAT ARE USED THROUGHOUT OUR EVALUATION

	Trace-Akamai	Wikipedia	Trace-A
Duration (Day)	3	15	7
Total Requests (Million)	81.04	2800	1200
Bytes Requested (TB)	71.6	31	14
Unique Objects Requested (Million)	17.18	37.53	170
Unique Bytes Requested (TB)	5	9	1.5
Mean Requested Object Size(KB)	6	54	13
Max Requested Object Size(MB)	79	127	29
Number of Tenants	96	178	112

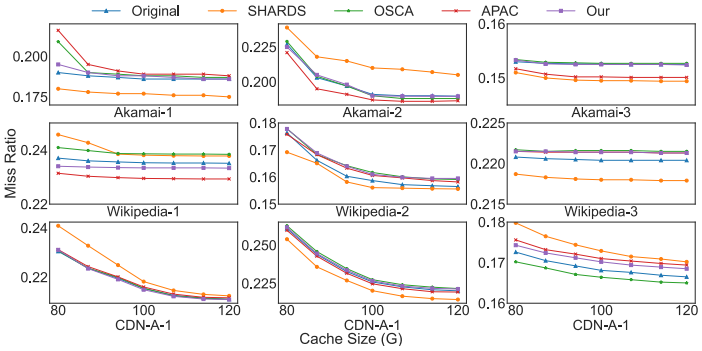


Fig. 6. Local MRCs. The cache space requirements vary among tenants and the curves of LACA are closer to the curves of the exact simulation than other methods in most cases.

the search space in two ways, by reducing the number of tenants searched and by reducing the scope of the search. Spatial Scheduler only adjusts space for tenants that use cache space inefficiently, and searches for configurations on local MRCs, so it can find the space to adjust at 1ms. After the Spatial Scheduler calculates the space that needs to be adjusted from tenants that are using cache space inefficiently, the objects that need to be written to the cache in the next period will replace those tenants' objects.

IV. EVALUATION

In this section, we provide comprehensive experiments to evaluate the effectiveness of the LACA. First, we describe the experimental setup used in this paper. Next, we evaluated the accuracy of the learned Local MRCs. Finally, we compare the overall efficacy of LACA in terms of hit ratio and miss reduction.

A. Experimental Settings

Traces. We use three CDN traces, including traces, i.e., Trace-Akamai [15], Wikipedia [16] and a real-world trace Trace-A that is collected from a commercial CDN system of Company-T. Their detailed information is shown in Table III. Note that the number of tenants represents the number of applications in Trace-Akamai and the number of services in Wikipedia. In our experiments, we treat each dataset as an access to one server in the cluster.

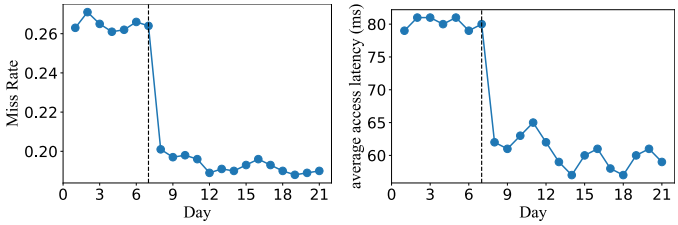
Schemes for comparison. We compare LACA with other three methods, including classical method SHARDS, OSCA, APAC on MRC approximation. In addition, we also compare with existing natural competition cache method (Default) in terms of hit rate and latency.

TABLE IV
THE MAES OF MRC APPROXIMATION. THE BEST ONES ARE IN BOLD. RUNTIME IS THE TIME TAKEN FOR OPTIMAL MEA.

trace	SHARDS	OSCA	APAC	LACA	Runtime
Akamai-1	0.0097	0.0041	0.0049	**0.0037**	69ms
Akamai-2	0.0088	0.0039	0.0042	**0.0036**	75ms
Akamai-3	**0.0031**	0.0035	0.0038	0.0034	81s
Wikipedia-1	0.0071	0.0023	0.0021	**0.0019**	62ms
Wikipedia-2	0.0064	**0.0018**	0.002	**0.0018**	32s / 61ms
Wikipedia-3	0.0073	0.0021	**0.002**	**0.002**	39s / 75ms
CDN-A-1	0.0089	**0.0027**	0.0031	0.0029	91s
CDN-A-2	0.0075	0.0024	**0.0022**	**0.0022**	37s / 73ms
CDN-A-3	0.0087	**0.0028**	0.003	**0.0028**	42s / 74ms

Simulator design. We have deployed LACA at the company-T's CDN system using the C++ library. In addition, for more comparative experiments, we designed a trace-based simulator in C++ to perform a quick verification. The cache replacement algorithm we keep the same as in the real system, using CLOCK. We set the cache space to 1T (cache size of a cache server). We are gradually making the simulation code and traces open source.

B. Results of MRC Approximation



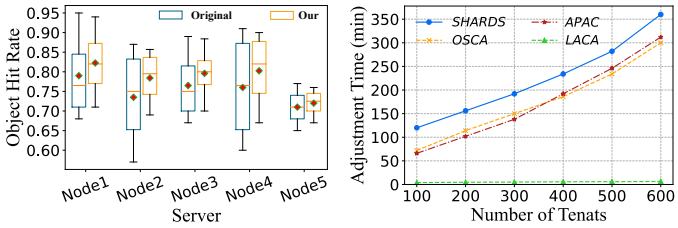
(a) Miss Rate (b) Latency

Fig. 7. Miss rate and latency in the monitoring system for 21 consecutive days. Note that LACA was deployed online at 24:00 on day-7.

We select three tenants on each of the three datasets to test the MRC. Our goal is to accurately construct the local MRCs of the over-occupied cache space tenants, that is, the cache space is generally larger. So, we set the range of cache size of each tenants to be between 80G and 120G and the comparison results are shown in Fig 6. We can see that although the MRCs exhibit drastically different forms, LACA can always almost accurately predict them, and in many comparisons its constructed MRC is closer to the real MRC than OSCA and APAC. In addition, in TABLE IV, we present the mean values of the Mean Absolute Error (MAE) distribution for each tenant. The main reason for this result is that our solution contains all the useful features used in the previous studies (e.g., reuse time for OSCA, local re-access for APAC) and the computation is done automatically using machine learning method. The runtime shows that our solution can make predictions at the millisecond level and is more lightweight.

C. Performance Comparison

Through the monitoring system, we gathered the metrics (miss rate and latency) for 21 days, where the 21-day time span included one week before the deployment of LACA and two weeks after the completion of the deployment. Fig 7 shows



(a) Hit Rate (b) Adjustment Time

Fig. 8. (a) Object hit rate for cache servers. The upper and lower lines represent the maximum and minimum values of the object hit rate of the cache server. The middle lines in boxes indicate the middle values. The bottom and top side of the box represent the quartiles. The red dot represents the overall hit rate of the cache server. (b) Comparison of the time used to adjust the space once as the number of tenants increases.

the change in miss rate and average access latency for 21 days, where the daily average miss rate and access latency are calculated at a granularity of one day from the monitoring system. We can see that the miss rate of the server reduce by 27.1% on average, and the average access latency dropped by 28.5ms, albeit these results may be biased due to some errors in the monitoring system. Sincerely, LACA improved the performance of the CDN cache system across the board. In order to show the effect of LACA, we collected traces from 5 cache servers in the cluster, and then compared the hit rate of each tenant after adding LACA. As shown in Fig 8(a), LACA only slightly reduces the hit rate of the highest hitting tenant, but greatly improves the hit rate of the lowest hitting tenant, and makes a significant improvement in the overall hit rate of the cache server. What's more, Fig 8(b) shows the comparison of the time required to adjust the space once, and it can be seen that LACA takes significantly less time than the other solutions due to its more lightweight design.

V. Conclusion

LACA solves the problem of uneven allocation of cache resources in CDNs, and is a lightweight dynamic space allocation strategy. By adjusting the tenants that inefficient use of cache resources step by step, cache resources can be more fully allocated to the tenants that need them more. In addition, LACA reduces the load overhead not only by reducing the number of MRCs built, but also by reducing the search range of the cache space. Therefore, LACA can make a spatially adjusted strategy within 15ms. Experimental results demonstrate that LACA can reduce the miss ratio by 27.1% and reduce the average user access latency by 28.5ms.

Acknowledgment

We thank the anonymous reviewers for all their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No.62172180, No.62232007, No.61821003).

References

[1] Google memcache resource limit. https://cloud.google.com/appengine/docs/standard/python/memcache.

[2] Pelikan cache - taming tail latency and achieving predictability. https://twitter.github.io/pelikan/2020/benchmark-adq.html.

[3] 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.

[4] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review*, 45(4):167–181, 2015.

[5] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[6] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[7] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[8] Rongshang Li, Yingtian Tang, Qiquan Shi, Hui Mao, Lei Chen, Jikun Jin, Peng Lu, and Zhuo Cheng. Accurate probabilistic miss ratio curve approximation for adaptive cache allocation in block storage systems. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1197–1202. IEEE, 2022.

[9] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[10] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[11] Raman Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. Probabilistic shared cache management (prism). In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 428–439. IEEE, 2012.

[12] Sparsh Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys (CSUR)*, 50(2):1–39, 2017.

[13] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[14] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride:near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

[15] Anirudh Sabnis and Ramesh K Sitaraman. Tragen: a synthetic trace generator for realistic cache simulations. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 366–379, 2021.

[16] 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.

[17] Jian Tan, Guocong Quan, Kaiyi Ji, and Ness Shroff. On resource pooling and separation for lru caching. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–31, 2018.

[18] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.

[19] 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.

[20] 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.

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

[22] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Osca: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798, 2020.