2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00066

A Lightweight and Adaptive Cache Partitioning Scheme for Content Delivery Networks

Peng Wang*, Zhelong Zhao*, Yu Liu†✉, Ke Zhou*, Zhihai Huang‡, Yanxiong Chen‡
*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
†School of Computer of Science and Technology, Huazhong University of Science and Technology, Wuhan, China
‡Tencent Technology (Shenzhen) Co., Ltd., Shenzhen, China
{wp_hust, zhelongzhao, liu_yu, zhke}@hust.edu.cn, {tommyhuang, yxbillchen}@tencent.com

***Abstract*—Allocating exclusive resources for different applications in content delivery networks (CDNs) allows for a higher overall hit ratio. The cache partitioning schemes on Last-Level Cache (LLC) are promising solutions that dynamically split cache sizes into partitions corresponding to threads by the miss ratio curve (MRC). Nonetheless, due to the sheer number of applications and various item sizes in CDNs, partitioning via MRC will cause high computational overheads and performance fluctuations. As a result, in this paper, we propose a lightweight and adaptive cache partitioning scheme (LAP) for CDNs. LAP establishes a shadow cache for each partition, where the size of the partition and its shadow cache is equal to the size of the integral cache. The average number of hits on the granularity unit in the shadow caches, where the size of the granularity equals the size of the probable largest item, is used to sort N partitions in decreasing order. When resizing partitions, LAP transfers a capacity of the size of granularity from the $(N-k+1)$-th ($k \leq \frac{N}{2}$) partition into the k-th partition. Meanwhile, we provide a threshold that neglects partition resizing and improves partitioning efficiency. This lightweight scheme can enhance resource utilization by progressively adapting to workload variations. We have deployed LAP in PicCloud of Company-T and LAP can improve the OHR by 9.34% and reduce the average user access latency by 12.5ms. Then, we verify LAP in the public trace from Akamai and the real trace from PicCloud. Experimental results demonstrate that LAP outperforms other cache partitioning schemes and tackles the performance cliff problem with little overhead.**

***Index Terms*—content delivery network, cache partitioning, performance isolation**

I. Introduction

Content Delivery Networks (CDNs), which have carried 56% of web traffic in 2017 and are expected to reach 72% by 2022 [2], deliver those content to users through networks of caching servers. With the increasing traffic, traditional schemes (*e.g.*, replacement policies, prefetch policies, and admission policies) are difficult to obey the request rules of workloads consisting of applications sharing caches. Those colocated workloads often suffer from significant performance degradations due to resource contention [13], which makes cache partitioning an important solution.

This work was also achieved in Key Laboratory of Information Storage System and Ministry of Education of China. It was supported by the National Natural Science Foundation of China No.61902135 and the Joint Founds of ShanDong Natural Science Funds (Grant No.ZR2019LZH003).

To the best of our knowledge, the existing cache partitioning methods predominantly work on Last-Level Cache (LLC), including UCP [4], PriSM [12], Dynacache [15], KPart [3], and DCAPS [5]. The majority of them partition the cache for different threads and determine the sizes of partitions via MRC, ensuring that exclusive resources for each thread can deliver benefits, resulting in an increase in the overall cache hit ratio (OHR). Although these methods have achieved satisfactory performance on LLC, there are two challenges when we apply them by replacing the threads with applications in CDNs.

High Computational Overheads Caused by Mass Applications. Since the number of threads and the size of LLC in CPU are comparatively small, the overheads using MRC are acceptable. Nevertheless, in the CDNs, the number of applications often exceeds 10^4, and the cache size exceeds 1TB. Compared to the smallest computation complexity of partitioning in LLC, the partitioning overheads are at least 10^7 times larger in CDNs Moreover, the lookahead algorithm enables the partitioning scheme, *e.g.*, UCP (lookahead), to tackle the performance cliff problem [16], requiring higher computation complexity and overheads. 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 [3].

Performance Fluctuations Caused by Different Item Sizes. Another challenge comes from different item sizes. After partition resizing, the cache must pass through a time window where the items are needed to replenish the expanded partitions. Since MRC-based judgments are scarcely sensitive to both quantity and size, inefficient resource utilization with performance fluctuations may occur during this window. We express the fluctuations induced by the filling process in terms of resource use to quantify them.

Rethinking the feasibility of utilizing MRC for partitioning under CDNs, we believe that workload variation is more complex in the case of inconsistent item sizes. Resizing partitions by MRC may be untrustworthy. The greater the *R-size*, the total extra size of the expanded partitions after partition resizing, the higher the risk of resource waste is. Through experimental observation, we find that *R-sizes* with small variance can yield relatively high hit ratios. Furthermore,

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00066

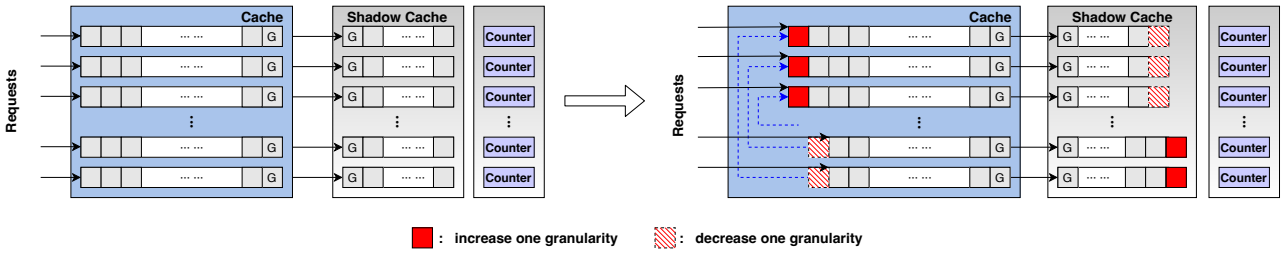


Fig. 1. CDN partition initial state and cache partition after a partition adjustment.

we find a range of *R-sizes* in the item size distribution corresponding to high hit ratios. As a result, we treat all partitions as multiple pairs. For each pair, we transfer one *G-size* from a partition to the other in a resizing process, where the *G-size* equals the size of granularity while the sum of *G-sizes*, *i.e.*, *R-size*, is in the range. Specifically, our scheme and other partitioning schemes resemble each other in shadow cache construction, but we use the numbers of hits on granularity in shadow caches to determine partition resizing instead of MRC.

We have deployed LAP in PicCloud of Company-T and LAP can improve OHR by 9.34% and reduce the average access latency by 12.5ms. Other evaluations on public and real-world traces indicate that our scheme can reduce the MR by 5%-10% in the simulator and outperform other state-of-the-art partitioning methods. In addition, experimental results verify that our scheme can tackle the performance cliff problem with little overhead.

II. Lightweight and Adaptive Cache Partition

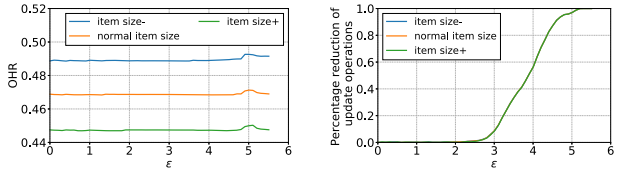
A. Overview

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:

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

As shown in Fig. 1, 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, used to record 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 resources for each application, satisfying Eq. (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 Eq. (1). We show the concrete process of once partition adjustment in Fig. 1.



(a) OHR (b) Reduction in update times

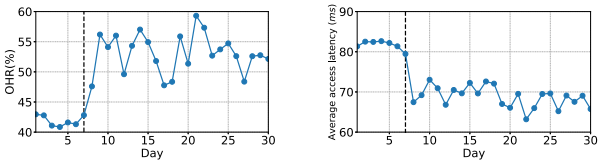
Fig. 2. The change of OHR and reduction in update times with changing ε under different item sizes on Trace-BJ at 40GB cache size. Note that when $\varepsilon = 0$, all partitions need to be updated. Three lines overlap in (b).

For the trigger of partition adjustment, we set τ as the update interval. When the total request misses for all applications reaches τ, partition adjustment will be executed. In the process of the adjustment, LAP first calculates $R_i = \frac{cnt_i}{S_i/G}$ $(i = 1, 2, \ldots, N)$ and arranges R_i in descending order. Then, 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 R_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 miss. 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. Reduce unnecessary partition resizing

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

$$G(\frac{cnt_i}{S_i} - \frac{cnt_j}{S_j}) > \varepsilon. \tag{2}$$



(a) OHR (b) Latency

Fig. 3. OHR and latency in the monitoring system for a continuous month. Note that the LAP algorithm was deployed online at 24:00 on day-7.

TABLE I
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

To determine an appropriate ε, we explore the relationship between ε and OHR. As shown in Fig. 2(a), when $\varepsilon = 4.9$, OHR in the blue curve (*i.e.*, item size-) reaches the peak value. For the orange curve (*i.e.*, normal item size) and the green curve (*i.e.*, item size+), the corresponding ε equals 5.0 and 5.1 respectively at the OHR peak. From this test, we find that the distribution of item sizes is insensitive to the determination of ε. We select $\varepsilon = 5.0$ based on the results with the normal item sizes, *i.e.*, we update sizes of partitions only when the difference in Eq. (2) 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 the time without a reduction in performance. We show this improvement in Fig. 2(b).

III. Implementation and Improvement on PicCloud

We have deployed LAP at the OC layer of PicCloud using the C++ library since most requests concentrate on the OC layer so that LAP can take advantage of the situation. 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 LAP and about three weeks after the completion of the deployment. Fig. 3 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 12.5ms, albeit these results may be biased due to some errors in the monitoring system. Sincerely, LAP improved the performance of PicCloud across the board.

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

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

IV. Evaluation

A. Settings and Simulator

Traces and *G-size*. We use two CDN traces, including a public trace, *i.e.*, Trace-Akamai [11] and a real-world trace, *i.e.*, Trace-BJ that is collected from PicCloud of Company-T. Their detailed information is shown in Table I, where the working set size means the size of all unique objects [14].

State-of-the-art cache partitioning schemes. We compare LAP with UCP [4], KPart [3], DCAPS [5], and No-partition (*i.e.*, LRU). UCP is the classical partitioning scheme. KPart is the representative partitioning scheme based on MRC. DCAPS is a typical partitioning scheme combined with intelligent algorithms such as machine learning.

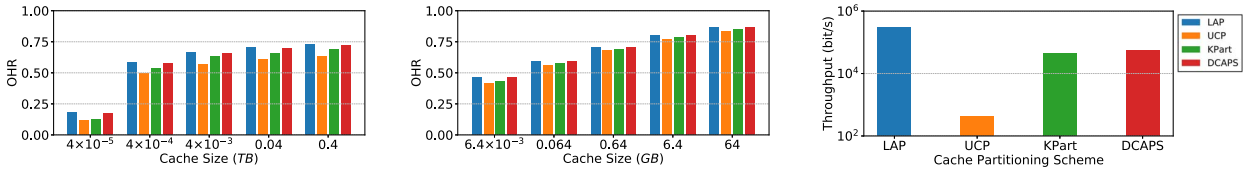
Simulator and testbed. We deploy the simulator based on *webcachesim* [1]. We implement LAP, UCP, KPart, DCAPS, and LRU on this simulator. To ensure fair and meaningful comparisons, we evaluate the above cache partition schemes on the same server, whose configurations are two 8-core AMD EPYC$^{\text{TM}}$ 7551 CPUs, 32GB RAM, and 1.5TB SSD.

B. Compare LAP with LRU on simulator

To verify the effect of partitioning in LAP, we compared No-Partition (*i.e.*, LRU) to LAP in terms of OHR on the simulator. 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, 0.04TB, and 0.4TB, respectively, on Trace-Akamai. Similarly, LAP can improve the OHRs by 8.03%, 6.11%, 3.87%, 4.56%, and 3.56%, respectively on Trace-BJ. We show the overheads produced by LRU, LAP, and UCP(lookahead), respectively, in Table II. The peak CPU utilization of LAP is slightly higher than that of LRU and is much lower than that of UCP(lookahead). For storage overheads, LAP only overtakes LRU by 0.05GB at 32GB RAM. In addition, the tiny gap in throughput between LRU and LAP is 1.08×10^{-5}Gbps.

C. LAP vs. State-of-the-art Cache Partition Schemes

We compare LAP to four state-of-the-art cache partitioning schemes on two traces with a wide range of cache sizes. As shown in Fig. 4(a) and Fig. 4(b), LAP outperforms other cache partitioning schemes in terms of OHR. For Trace-Akamai, LAP can outperform other cache partitioning schemes by 5.01%, 5.61%, 4.87%, 5.85%, and 6.17% at different cache sizes of 4×10^{-5}TB, 4×10^{-4}TB, 4×10^{-3}TB, 0.04TB, and 0.4TB, respectively. Similarly, for Trace-BJ, the average improvements are 3.21%, 2.06%, 1.37%, 1.97%, and 1.58%, respectively. In addition, we compare different schemes in terms of throughput. Since there are only two applications



(a) Trace-Akamai (b) Trace-BJ (c) Throughput

Fig. 4. Performance comparison of several algorithms with different cache sizes on two traces.

in Trace-Akamai, we only achieve this experiment on Trace-BJ. As shown in Fig. 4(c), the throughput of LAP on the simulator is approximately 700 times that of UCP, 6.62 times that of KPart, and 5.34 times that of DCAPS.

V. Related Work

In this section, we will introduce the research related to resource partition. Many researchers, inspired by resource isolation techniques, resort to resource partitioning policies for system performance improvement. Recent studies have designed LLC and memory bandwidth partitioning policies based on hardware resource isolation techniques, *e.g.*, KPart [3], DCAPS [5], and EMBA [6]. They usually build specific models relying on extensive knowledge (such as MRC) to estimate application performance under various partitioning schemes. Based on those models, those handcrafted heuristics can only work well on pre-defined scenes but cannot cope with dynamically changing workloads in the CDN. CoPart [8], Quasar [9], and Heracles [7] can handle multiple resources partitions. Unfortunately, due to the lack of precise models to capture the relationship between resources and performance, these partitioning schemes are used in limited scenarios. In recent years, with the application of machine learning in resource management, many scholars have also tried to combine resource allocation with intelligent algorithms such as reinforcement learning. Chen *et al*. [13] propose a deep reinforcement learning framework for solving the problem of partitioning multiple resources and Li *et al*. [10] leverage Q-Learning to implement adaptive online partitioning. However, these methods cannot work efficiently under CDNs since their data storage, model training, and update require additional resource consumption.

VI. 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 used on LLC. 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 (LAP) to address the above issue. LAP uses the number of hits on the shadow cache, *i.e.*, R_i, rather than the MRC curve to trigger of 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 R_i, reducing unnecessary resizing operations. The experimental results validate the efficiency and accuracy of LAP, and it outperforms state-of-the-art methods in terms of OHR, throughput, and update time. As the first partitioning algorithm on CDN, LAP considers the impact of different item sizes in the selection of *G-size*. We hope that this exploration will spark improvements to CDN partitioning algorithms and contribute to the community.

References

[1] D. S. Berger, R. K. Sitarama, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in NSDI, USENIX, pp. 483-498, 2017.
[2] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in NSDI, USENIX, pp. 529-544, 2020.
[3] N. El-Sayed, A. Mukkara, P. Tsai, H., X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in HPCA, IEEE, pp. 104-117, 2018.
[4] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in MICRO, IEEE, pp. 423-432, 2006.
[5] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic cache allocation with partial sharing," in EuroSys, pp. 1-15, 2018.
[6] Y. Xiang, C. Ye, X. Wang, Y. Luo, and Z. Wang, "EMBA: Efficient memory bandwidth allocation to improve performance on intel commodity processor," in ICPP, pp. 1-12, 2019.
[7] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in ISCA, pp. 450-462, 2015
[8] 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 EuroSys, pp. 1-16, 2019.
[9] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in ACM SIGPLAN Notices, ACM, vol. 49, no. 4, pp. 127-144, 2014.
[10] Y. Li, X. Wang, H. Liu, L. Pu, S. Tang, G. Wang, and X. Liu, "Reinforcement Learning based Resource Partitioning for Improving Responsiveness in Cloud Gaming," in TC, IEEE, 2021.
[11] A. Sabnis and R. K. Sitaraman, "TRAGEN: a synthetic trace generator for realistic cache simulations," in IMC, pp. 366-379, 2021.
[12] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in ISCA, IEEE, pp. 428-439, 2012.
[13] 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 HPDC, pp. 175-188, 2021.
[14] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in OSDI, USENIX, pp. 191–208, 2020.
[15] C. Asaf, E. Assaf, A, Mohammad, and K, Sachin, "Dynacache: Dynamic cloud caching," in HotCloud, USENIX, 2015.
[16] C. Asaf, E. Assaf, A, Mohammad, and K, Sachin, "Cliffhanger: Scaling performance cliffs in web memory caches," in NSDI, USENIX, pp. 379-392, 2016.