Implementation of Multi-tenancy Adaptive Cache Admission Model

Huajun Ma\textsuperscript{1,2}, Jiyang Zhang\textsuperscript{2} and Zixian Wang\textsuperscript{1}

\textsuperscript{1}Foshan Saisichan Technology Co., Ltd., Foshan, China
\textsuperscript{2}Shenzhen Graduate School Peking University. Shenzhen, China
Email: mahj917@163.com

Abstract. Cyber performance is an evolving issue for enterprises and individuals. However, Internet applications with many concurrent accessing are obsessed with the tradeoff between the scale of users and the efficiency for each user. Compared with multi-tenancy catch technology, it is almost impossible to meet the requirements of enormous accessing when improving the hardware performance. Different from the classical multi-tenancy cache system focused on the size of cache partition, we propose a multi-tenancy cache model with adaptive admission strategy. The model evaluates the hit efficiency of cache objects in cache space, and expels cache objects with low hit rate through an effective algorithm. The experimental results show that the proposed method improve the response efficiency, especially in the high concurrent accessing scene for Internet applications.

Keywords. Multi-tenancy cache model; multi-tenancy adaptive cache; admission strategy.

1. Introduction

By 2021, the number of Chinese webpages had reached 315.5 billion, an increase of 5.9\% compared with the end of 2019 [1]. In addition, Chinese Internet access consumption had reached 165.6 billion GB, up 135.74\% from the end of 2019.

With a large amount of webpage data and network traffic, many solutions have been proposed in order to provide high interactive performance [2]. The most widely used is utilizing Web caches to temporarily store data that may be frequently requested, so as to reduce the overhead of server recomputation and response time of the website.

Traditional caching solutions are often used as a system component, not for a specific application. Due to the characteristic of low-coupling, the traditional caching scheme cannot meet the caching efficiency requirement of the application system when dealing with complex logic and personalized contents. As an emerging cache solution, application-level cache has attracted wide attention in the industry. Compared with traditional caching, application-level cache allows the system to cache data in an appropriate way by coupling the caching logic to the application logic. It improves the performance and scalability of the application system, as well as reduces the work responsibility and communication delay.

Application-level caches are usually developed by professionals who accumulate experience of system maintenance. They are trying to develop specific functions and business scenarios in a performance-optimized method. This targeted and optimized development may completely change the original cache design scheme [3]. Developers frequently adjust caching strategy in order to optimize system performance constantly [4], thus giving birth to adaptive application-level caching technology.
It focuses on adaptive cache management by detecting cache access patterns. It can also self-dynamically evolves to achieve desired cache optimization goals.

2. Preliminaries
As a supplement to traditional caching solutions, application-level caching plays an important role in improving system performance. [5] describes four key problems that need to be solved when building application-level cache:

- How to choose the appropriate cached content.
- When to load or remove cached objects from the cache.
- Determine the effective storage location for cached content.
- How to achieve the appropriate caching logic.

Programs shall be responsible for the cache and persistent data, so developers often must manually insert content, as well as maintain the transformation and distribution between the raw data and cached object. These approaches usually adopt the static and dynamic method to keep consistency.

2.1. Memshare
Memshare [6] uses shared memory across applications to maximize cache hit ratio and provide isolated cache space for each tenant. With a unique log-based memory allocation model, it moves cached resources across multiple applications more efficiently than the slab allocator.

Memshare contains two key components, Arbiter and Cleaner. In particular, Arbiter manages the allocation of each program memory space.

According to the eviction property of cache items and the cache usage of each program, Cleaner evicts cache items that are considered as having multiple allocated space Memshare introduces the estimated value $\text{need}(\text{app})$, which is used to assess the degree requirement of memory space to each application, defined as where $\text{actualMem}$ is the actual size (unit bytes) of the application's storage, and $\text{targetMem}$ is the allocated size (unit bytes) of the cache space that the application should cache. When using static partitioning methods, $\text{targetMem}$ is a constant value. When $\text{need}(\text{app})$ is greater than 1, it indicates that the application is not allocated enough memory space and there is waste of memory resources. When $\text{need}(\text{app})$ is less than 1, the application consumes more memory than the allocated memory space, and the memory space is not allocated enough. Arbiter sorts your applications based on the value of $\text{need}$, and the Cleaner cleans up the log segments of those apps that overuse memory. The cache entries on the log segments are cleaned one by one. Removing the unreasonable caches and keeping the reasonable caches also leads to a large number of fragmentations.

$$\text{need}(\text{app}) = \frac{\text{targetMem}(\text{app})}{\text{actualMem}(\text{app})}$$ (1)

2.2. mPart
mPart [7] improves the global cache hit Ratio by constructing the Miss Ratio Curves (MRC) from Memshare. mPart uses working-set size allocation to model cache allocation in multi-rented environment.

The best way to allocate memory is $m = [m_1, m_2, ... , m_N]$. The goal is to optimize the global cache hit ratio. $M$ is the cache size, $N$ is the total number of applications (tenants), $\{\text{mrc}\}_{i(m)}$ is memory size when the cache-miss rate is $m_i$, $\{\text{NR}\}_i$ is the requests number of the application $i$, $(1 - \{\text{mrc}\}_{i(m)}) * \{\text{NR}\}_i$ is the cache-hit number of tenants $i$.

Memshare and mPart both assume the size of cache objects should be evenly distributed and fluctuate within a range. But the actual production environment is more complex, and the size of the cache objects can vary by orders of magnitude. Large cache objects can cause a sharp drop in the cache hit ratio of a single application. Take 1 GB cache for example. Assuming that there are only two cache objects in the memory space: 9999 cache objects with 100 KB for each one and 1 cache object with 900 MB. A large cache object request will eject approximately 9000 smaller cache objects, and the cache hit ratio is less than 10%. In the case of Memshare, if a program frequently requests large cache objects over a period of
time, many small but hot objects will be squeezed out and placed in shadow queues. Accessing small cached objects causes the shadow queue hit ratio to increase, resulting in an "over shortage" state. At this time, Arbiter can only throw out other application cache objects to meet the cache needs of this abnormal application. This approach directly results in a global hit ratio drop.

In addition, the generation and transmission cost of different cache objects are different. Some cache objects are accessed frequently after requesting with expensive computational overhead. For such cached objects, we prefer to keep them in the cache for a long time. Although this method cannot improve the overall hit ratio of cache, it can effectively reduce the overall access delay of production system. On the other hand, neither Memshare nor mPart has a more granular customization design on this dimension.

3. The Proposed Method
To solve the above problems, we propose a multi-rent application-level cache model based on adaptive access strategy CSER (Cost-aware, Size-aware, Efficient and Reliable Multi-tenant Key-value Cache).

3.1. Abbreviations and Acronyms
CSER is a multi-rent application-level cache model. Figure 1 shows a typical application scenario for a multi-tenant application-level cache system.

Multiple tenants (applications, Web, etc.) obtain hot data objects to avoid frequent access (such as databases, file systems, etc.) in the current production environment by sharing and using cached resources.

Compared with traditional static caching solution, multi-rent application-level cache model can be adjusted according to the current access patterns of all tenant’s dynamic cache object access parameters, and the memory resources quota of each tenant, make memory resources can be properly used to where it is most urgent and eventually improve the overall performance of system.

![Figure 1. A typical scenario that the initial type of the application is unique.](image)

3.2. Functional Design Goals

3.2.1. NoSQL Key Value Caching System. In the actual environment, the structure and form of data objects will change with the time, and it is a common situation that the structure of data objects deviates from the original intention of design. Multi-tenant caching system needs to cache and manage data objects among multiple tenants, while different tenants will have many differences in the form of data object design and construction due to great differences in business logic and application scenarios. Traditional relational databases, due to data structure, strict ACID and other designs, cannot meet some high-performance application requirements in terms of performance, nor can they meet diverse storage requirements in terms of flexibility. Key value storage based on NoSQL (Not Only SQL) is a popular new storage model, which does not need fixed data mode and can be easily extended horizontally. In
order to meet tenants’ strict requirements on the flexibility and performance of data form, CSER uses key-value Storage to build the Shared Storage model.

3.2.2. Self-Adaptive Strategy. Considering that caching is only a performance requirement for production applications, CSER is designed to run on isolated data nodes to provide consistent and stable service for production environments. Currently, multi-tenant caching systems are usually memory-bound programs with underutilized CPU resources and bus bandwidth. Therefore, this paper learns from the excellent design ideas of Memshare and mPart, and makes full use of idle resources (CPU, bus bandwidth, etc.), so that it can dynamically adjust partition decisions according to the workload of tenants. In addition, we make a more fine-grained adaptive design for cache admittance.

3.3. Non-functional Design Goals

3.3.1. Data Consistency. Different business scenarios have inconsistent requirements for consistency. For example, financial business systems usually need strong consistency, while social application systems have lower consistency requirements. According to Brewer's CAP theory [8], consistency, availability and partition tolerance of distributed systems cannot be satisfied at the same time. Therefore, a multi-tenant cache should give developers the flexibility to customize the consistency model based on actual requirements.

3.3.2. Transaction and Isolation. A cached transaction should have atomic semantics, that is, all or none of the transaction operations succeed or fail. At the same time, transactions also need to be isolated, that is, the execution process of a single transaction is not disturbed by other transactions.

3.3.3. Reliability. Rather than “making applications more robust”, we think it’s more important to recover quickly after a service outage. To prevent cache data loss due to unexpected service outages, the CSER model persists cached data. When CSER goes down unexpectedly, after restarting the application service, the application system can quickly and accurately restore to the state before the outage and continue to provide stable service for the front-end program.

3.3.4. High Availability. CSER model adopts modular design, including the functions of arbitration, cleaning (expulsion), persistence, monitoring and so on. The CSER model needs to serve multiple clients at the same time, which requires the throughput of the model itself to meet the requirements, and provides high-availability, low-latency caching services for tenants. CSER cache requests must be processed quickly, and the modules of the client cache requests must not interfere with each other to ensure the reliability of the model.

4. Framework of The Model

CSER’s adaptive access model selects which cache objects to access based on the size of the cache and the cost of reloading. Cache objects with small cache space, high reload cost and high access frequency should be given priority access. CSER also USES probability-based access function to build an adaptive access model, realizing the possibility of access for any cache object.

We use $b_i$ to represent the gain from the cache object $I$ staying in the Shared cache, which is inversely proportional to the size $S_i$ of the cache object $I$ and directly proportional to the cost $w_i$ of reloading. The reload cost $w_i$ is the cost of obtaining the original data content again after the cache object is expelled, including calculation, query, transmission and other costs. $f_c(b_i)$ is used to indicate the possibility of cache object $I$ being allowed into CSER system, and parameter $c$ is the adaptive access parameter. Our goal is to find the optimal access parameter $c$ so that CSER is balanced in cache object hit ratio and total access delay. Because there are too many factors affecting access parameter $c$, it is difficult for cache operators to set it in advance through prior experience. The optimal access parameter $c$ is not a constant, but changes continuously over time.
The adaptive access model of CSER prototype system is the extension and generalization of Markov chain model proposed in [9]. Let cache objects be organized in cache space by LRU [10], $K$ is the total size of cache resources, $P_{in}(i)$ is the probability of cache object $i$, and $s_i$ is the size of cache object $i$. Then the total size of all cached objects should be expected to be $K$:

$$K = \sum_{i=1}^{N} P_{in}(i) \cdot s_i$$  (2)

where, the total number of cached objects is $N$. Let $r_i$ be the probability that cache object $i$ is put into LRU header (i.e. accessed), which can be obtained by statistics. $\mu_c$ is the probability that a cached object I moves a step back in the LRU cache. First of all, cache objects I can only move one step back because other cache objects in the cache space are accessed again or new cache objects are allowed into the cache space. When the total number of cache objects is large enough, the push probability $\mu_c$ is statistically approximately equal for any cache object $i$. Secondly, access parameter $c$ is closely related to push down probability $\mu_c$ and $P_{in}(i)$. When the access parameter $c$ is tightened, the access probability decreases and fewer cache objects are ejected, resulting in lower push rate $\mu_c$ and higher $P_{in}(i)$.

Therefore, $P_{in}(i)\cdot r_i$ can be used to approximate the hit times of cache object $i$, and then $OHR(c)$ is:

$$OHR(c) = \frac{\sum_{i=1}^{N} P_{in}(i) \cdot r_i}{\sum_{i=1}^{N} r_i}$$  (3)

where, $OHR(c)$ is the cache object hit ratio under the condition that the access parameter is $c$. Apparently $OHR(c) \in [0, 1]$.

If the cache object $i$ cannot be retrieved from the cache space, it must be retrieved again at the cost of $w_i$. If the cache object I can hit and fetch data from the cache space, the cost is negligible compared to $w_i$. Therefore, the expectation of $TC(c)$ of the total retrieval cost is:

$$TC(c) = \sum_{i=1}^{N} (1 - P_{in}(i)) \cdot r_i \cdot w_i.$$  (4)

Set $TC_{max}$, $TC_{min}$, as the maximum and minimum values of $TC(c)$, and then normalize $TC(c)$ to:

$$TC_N(c) = \frac{TC(c) - TC_{min}}{TC_{max} - TC_{min}}$$  (5)

Mapping $TC_N(c)$ to region $[1 - \alpha, 1]$ yields, we get:

$$TC_\alpha(c) = \frac{TC(c) - TC_{min} + (\alpha - 1)(TC_{max} - TC_{min})}{\alpha(TC_{max} - TC_{min})}, 0 < \alpha \leq 1$$  (6)

$$OG(c) = \frac{OHR(c)}{TC_\alpha(c)}$$  (7)

where, the tradeoff factor alpha represents the width of the region mapped by $TC_N(c)$, controlling the sensitivity of the adaptive access model to the cache object retrieval cost $w_i$. The closer alpha is to 1, the greater the impact of acquisition cost $w_i$ on the adaptive model. When alpha is close to 0, it means that $TC_N(c)$ maps to a fairly narrow interval, and $TC_N(c)$ is close to 1. In particular, when alpha is 0, this adaptive access model is reduced to Markov chain-based access model of [9] (i.e., $w_i$ of cache object retrieval cost is no longer considered).

It is worth noting that if the cost of reacquiring object A $w_A$ is greater than the cost of reacquiring object B $w_B$, but object B is hot data and object A is cold data, the value of object A and object B residing in the cache is not clearly superior. However, during the experiment, it was found that the performance did not improve if the hot spot degree of the object was considered in the access model (such as the frequency of access of the object was introduced to approximate). The main reason is that cached objects mostly use LRU or LFU expulsion strategies, while either strategy tends to expel cold data.

In order to obtain $OG(c)$ function curve of adaptive access parameter $c$, the most critical place is the derivation of $P_{in}(i)$. Intuitively, $P_{in}(i)$ is directly related to the adaptive access parameter $c$. That is, when the access parameter $c$ decreases, the probability of cache objects being admitted decreases, and fewer cache objects are expelled, resulting in lower push rate $\mu_c$ and higher $P_{in}(i)$. [9] gives the results
of \( P_{in}(l) \) under the exponential probability access function through Markov chain model. Based on this, this paper will extend the model to a more general form and give a detailed derivation process.

Assume the length of LRU linked list is \( l \), then there are a total of \( l+1 \) states for any cache object \( l \), namely, the first node and the second node in the LRU linked list...The LTH node, and it's not in the cache. There are three transition states in a cache: a) the probability \( \mu_c \) of a move down if a subsequent cache object is accessed or a new cache object is allowed to enter; B) the probability of being moved to the head because of being accessed is \( r_i \); C) hold still, such as the probability that a cached object in the LRU list is accessed closer to the head than the cached object I itself. For non-cached states, there are two transition states: a) the probability is \( r_i \cdot f_c(b_i) \) for being accessed and granted access; B) hold for a probability of \( 1 - r_i \cdot f_c(b_i) \).

The length of the LRU linked list \( l \) will change as large or small cached objects are entered or expelled. However, it can be observed that if the cached object \( i \) is not accessed again, the expected time from state 1 to state \( l+1 \) is constant and independent of \( l \). Meanwhile, the expected time is closely related to the admission probability function (namely, the value of admission parameter \( c \)). More specifically, when the access probability function relaxes the access restrictions, more cache objects are allowed in, which allows the cache object \( i \) to be ejected from the cache space faster (if not accessed again). In addition, for cache object \( i \) in state 1, the path length that needs to be moved to be expelled is always \( l \), since no other object can be inserted. Therefore, when the Markov chain is modified, the expected time for the cache object \( i \) to be expelled is a constant value of \( \frac{1}{\mu_i} \), and its transfer matrix is:

\[
\begin{pmatrix}
1 - l \mu & l \mu & 0 & 0 \\
0 & 1 - r_i - l \mu & l \mu & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1 - r_i \cdot f_c(b_i)
\end{pmatrix}
\]

(8)

The transfer matrix \( \mathbb{P} \) only \( l+1 \) limited state, each state transition probability of the fixed, from any state can be transferred to other states, and is not a simple loop, and therefore the Markov process can converge to a steady state. Assume that arrived in a smooth, the steady-state probability is \( \xi = (\xi_1, \xi_2, ..., \xi_{l+1}) \), which \( \xi_l \) steady-state probability of \( i, 0 \leq i \leq l+1 \) or less.

\[
\begin{align*}
\xi_1 + \xi_2 + \cdots + \xi_{l+1} &= 1 \\
\xi \cdot \mathbb{P} &= \xi
\end{align*}
\]

(9)

The cache object \( i \) LRU list in the steady state probability is:

\[
\xi_{in} = \left( \frac{f_c(b_i)}{l \cdot r_i + f_c(b_i)} \right) \frac{1 - \left( \frac{i}{l \cdot r_i} \right)^l}{1 - \left( \frac{i}{l \cdot r_i} \right)^l}
\]

(10)

Note that \( l \) is usually large and related to the average size of cache resources and objects. The expected probability of cache object \( i \) in the cache space can be obtained by taking the limit of \( l \):

\[
P_{in}(i) = \frac{f_c(b_i) \cdot e^{r_i / \mu_c - 1}}{1 + f_c(b_i) \cdot e^{r_i / \mu_c - 1}}
\]

(11)

Two problems need to be solved to integrate the above adaptive access model into the CSER system. The first is the solution of equation (2). The fixed-point iteration technique mentioned in [11] can be adopted. The general idea is to change the equation into the form of \( x = g(x) \) and solve it by iterating \( x_{n+1} = g(x_n) \). The second problem is the extreme value solution of the function curve represented by equations (4) and (7).

In addition to the two issues mentioned above, note that since \( b_i \) approximately represents the benefit to cache object \( i \) staying in the Shared cache, and this value \( s_i \) is inversely proportional to the size of cache object \( i \), and proportional to the cost of reacquiring \( w_i \). When a cache object is submitted to the CSER system by a put request, the CSER system collects statistics and the \( b_i \) estimate value is now...
determined and is set to $w_i/s_i$. The size of the cache object $s_i$ is determined by its content, such as the large difference between the static web cache and video fragment cache. Since CSER system is limited by the application background and cannot be aware of the generation process of cache objects, the acquisition cost $w_i$ is often explicitly provided by the tenant.

5. Experimental Evaluation

5.1. Dynamic Partitioning Test
Firstly, after the introduction of adaptive access module, the performance of CSER in partitioning decision was tested in the past scenarios compared with the current multi-rent application-level cache system. Like Memshare and mPart, it was considered that the values of $s_i$ and $w_i$ of cache object $i$ were roughly equal and irrelevant.

![Figure 2](image)

**Figure 2.** The cache hit ratio of each system varies with the number of visits: (a) when the initial type of the application is unique; (b) when the initial type of the application is equal.

Figure 2 show that whether in the initial setting of 75% Unique or Equal, the CSER system can adjust itself well, improve the total hit rate by dynamically pooling resources to urgently needed tenants, and not lag behind mPart and Memshare in the stable time. Figure 3 shows the change trend of cache hit ratio of each system with time when the tenant adopts Unique initial setting. The curve change of CSER is roughly the same as that of mPart. The initial setting of Equal tenants is consistent with the Unique type in the trend. The experimental results show that the adaptive access module does not worsen the performance of dynamic partitioning decisions in the scenarios.

![Figure 3](image)

**Figure 3.** With the Unique initial setting, the cache hit ratio changes over time when the object size is uniform to 200 Bytes.
5.2. Adaptive Access Test
Test the performance of each cache system in different sizes of cache object data sets and calculate cache hit ratio. In order to control irrelevant variables, the default value of \( w_i \) for each cached object in the dataset is 1. Figure 4 is the size distribution diagram objects are fully utilized, so that the global cache hit ratio is maintained at a high level.

Performance of each cache system are shown in figure 4, the same obvious conclusion can be drawn: (a) the time taken by mPart and Memshare to complete all workflows is quite close; (b) CSER prototype system completed all data set requests earlier than mPart and Memshare, and its total time spent was 17.4\% less than mPart and Memshare on average. Within the first hour, the remaining workload of each cache system was fairly similar, mainly because the CSER system was adjusting the adaptive access parameters according to the access pattern of the data set. Over time, however, the access module continues to work, with a probability of filtering out those cache objects with less “value” in order to preserve those with more “value”. Each tenant will find that when a miss has to occur, the missing cache object is more likely to be the cache enables the tenant to re-acquire the missing cache object without waiting too long. object that has a lower cost of re-acquisition.

![Figure 4](image_url)

**Figure 4.** With the Unique initial setting, the cache hit ratio changes over time when the object size is uniform to 200 Bytes.

The performance of CSER system, such as cache hit ratio, does not benefit from the adaptive access module of cost-aware. However, if the perspective is global and the whole production environment of front-end business, cache system and back-end storage is considered, CSER system can generate potential gains compared with other multi-application caches.

6. Conclusion
This paper designs and implements CSER, a multi-rent cache system based on adaptive access strategy. The access parameter of optimization target is derived by introducing the adaptive access model. Due to the pipeline design, CSER can quickly reflect the collected data on Tuner’s adaptive decision, so that the CSER system can always find the optimal transfer parameter according to the tenant’s access mode and workload. The effectiveness of adaptive access model in CSER prototype system is proved by experiments, and the specific principle is given. The proposed CSER model solves the problem of concurrent access efficiency of multiple users in mimicry distributed storage system. It can also effectively improve the user response efficiency of WEB, cloud and other service terminals. In our future work, the malicious tenant problem of multi-rented cache system in the public application scenario will be considered.

References
[1] The 47rd China Statistical Report on Internet Development
http://www.cnnic.net.cn/hlwzyj/hlxwzxgl/hlwztjbg/202102/P020210203334633480104.pdf.
[2] Ravi J, Yu Z and Shi W 2009 A survey on dynamic Web content generation and delivery techniques *Journal of Network and Computer Applications* **32** (5) 943-960.

[3] Azim T, Nadeem A and Ailamaki A 2018 Adaptive cache mode selection for queries over raw data *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)* (ACM) pp 51-65.

[4] Radhakrishnan G 2004 Adaptive application caching *Bell Labs Technical Journal* **9** (1) 165-175.

[5] Mertz J, Nunes I. 2017. A qualitative study of application-level caching *IEEE Transactions on Software Engineering* **43** (9) 798-816.

[6] Cidon A, Rushton D, Rumble S M, et al. 2017 Memshare: A dynamic multi-tenant key-value cache *2017 USENIX Annual Technical Conference (USENIX ATC)* pp 321-334.

[7] Byrne D, Onder N and Wang Z 2018 mPart: Miss-ratio curve guided partitioning in key-value stores *ACM SIGPLAN International Symposium on Memory Management (ISMM)* (ACM) pp 84-95.

[8] Brewer E A 2000 Towards robust distributed systems *Annual ACM Symposium on Principles of Distributed Computing (PODC)* (ACM) p 7.

[9] Berger D S, Sitaraman R K and Harchol-Balter M 2017 AdaptSize: Orchestrating the hot object memory cache in a content delivery network *Symposium on Networked Systems Design and Implementation (NSDI)* pp 483-498.

[10] 2019 Let Cache Objects Be Organized in Cache Space by LRU https://varnish-cache.org/trac/wiki/ArchitectureLRU.

[11] Fofack N C, Towsley D, Badov M, et al. 2014 Single cache approximation *An Approximate Analysis of Heterogeneous and General Cache Networks* (The National Institute for Research in Computer Science and Automation (INRIA)).