MemSweeper: Virtualizing Cluster Memory Management for High Memory Utilization and Isolation

AmirHossein Seyri
University of Illinois at Chicago
USA
aseyri2@uic.edu

Abhisek Pan
Microsoft
USA
abpan@microsoft.com

Balajee Vamanan
University of Illinois at Chicago
USA
bvamanan@uic.edu

Abstract

Memory caches are critical components of modern web services that improve response times and reduce the load on backend databases. In multi-tenant clouds, several instances of caches compete for memory. The current state-of-the-art is to statically allocate memory for cache instances (e.g., based on cost-tier) but such allocation tends to be sub-optimal as memory demands of instances often vary with time and not known apriori. We propose *MemSweeper*, which dynamically manages memory between cache instances. *MemSweeper* uses a novel, score-based metric and an associated algorithm to identify cache instances whose working sets fit well within their allocated memory and thus can relinquish a portion of the memory without suffering appreciable loss in their hit rates. Using a combination of synthetic and production traces on a real implementation, we show that *MemSweeper* achieves 74% improvement (on average) in the miss rate of critical tenants without degrading the performance of other tenants.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Keywords: Cloud, Multi-tenancy, Cache, Memcached

ACM Reference Format:
AmirHossein Seyri, Abhisek Pan, and Balajee Vamanan. 2022. MemSweeper: Virtualizing Cluster Memory Management for High Memory Utilization and Isolation. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management (ISMM '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3520263.3534651

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISMM '22, June 14, 2022, San Diego, CA, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9267-9/22/06…$15.00
https://doi.org/10.1145/3520263.3534651

1 Introduction

Memory key-value stores are widely used in web services to reduce the load on backend databases and improve the latency of the requests [22]. While request latency is a key metric for user-facing services, backend databases often cannot handle the volume of requests and require in-memory caches to filter out most requests. By servicing most requests from main memory, these memory key-value stores drastically cut the number of accesses to disk (i.e., cache misses), which is inferior to main memory in both latency and bandwidth by multiple orders of magnitude. Consequently, even modest improvements in hit rates of these caches would likely achieve significant improvements in application performance. Therefore, maintaining near-optimal hit rates is a first-order concern for these applications. The key to achieving near-optimal hit rates lies in accurately estimating the working sets of applications and allocating space according to their working sets. Further, because working sets of applications tend to change over time, the system must learn and adapt to these changes.

Memcached [22] and Redis [30] are two popular caching solutions utilized by major web service providers such as Facebook [3, 25], Twitter [36] and Github [24] to maintain low web request latencies and improve the user experience. Additionally, In-memory key-value storage is offered by cloud providers such as Microsoft [23] and Amazon [1], as a service to customers who need a high-performance caching layer for their web applications. Multi-tenancy in cloud has been the focus of many recent papers [7, 10, 11, 17, 20, 28, 33]. To achieve high utilization, it is quite likely for multiple tenant applications to reside in the same physical machine and to share resources such as CPU, network bandwidth, and main memory. Protean [17] makes the case for co-locating VMs from multiple tenants to bin-pack as many VM instances as possible in a single machine. Moreover, operators spread VMs of one tenant across multiple servers to provide better fault tolerance. Doing so necessitates co-locating VMs from multiple tenants in the same physical server to improve utilization and reduce cost. Without such bin-packing, cloud providers will take a substantial hit in their profit margins. While the problem of achieving an optimal hit rate for a

single tenant, is well studied, we focus on achieving optimal hit rates by efficiently managing memory allocations in multi-tenant cache environments.

If multiple instances of memory caches are placed in the same machine, the common approach is to *statically* partition the physical memory based on some criteria (e.g., cost tier). However, because workloads exhibit significant diversity in terms of working set sizes and access patterns, *workload-agnostic* static partitioning leads to sub-optimal performance as cache instances with small working sets waste the space while instances with large working sets suffer. Further, the workloads are also dynamic as their working set sizes and access patters change over time. Thus, any static allocation will fail to adapt to tenants' demands and consequently suffer in performance.

Caches are known to exhibit *performance cliffs* in their hit rate curves (i.e., a plot of hit rate vs. cache size) [4, 10]. A performance cliff occurs when a small increase in memory results in a drastic improvement in the hit rate of the application and vice versa. Performance cliffs occur when a working set fits in the cache leading to improved locality (e.g., If there is an inner loop that iterates over some fixed subset of an array, a performance cliff would occur when the cache size equals the size of the subset). Thus, performance cliffs enable us to identify working sets of applications. While there is prior work [10] on identifying and removing such cliffs, they suffer from high overhead and they focus only on a single application. Further, they do not identify far-off cliffs and therefore suffer from poor performance when workloads exhibits such behavior (Figure 1). In this paper, we propose a novel algorithm to identify cliffs (working sets) with significantly lower overhead than prior work and leverage this information to allocate space among multiple cache instances that are co-located in the same server. Giving space to tenants proportional to their working sets will be optimal if the sum total of their working set sizes do not exceed the total memory capacity of the server. Unfortunately, most real applications tend to have large working sets that exceed the physical capacity of the machine when co-located with other applications. We propose a novel *score* metric that enables the system to achieve a good balance between fairness and utilization. Finally, *MemSweeper* adapts to changes to workload characteristics and achieves stable allocations.

MemSweeper allows cloud providers to co-locate clients in fewer machines and increase utilization which reduces the operational cost. Customers who pay for a large cache (a) will not experience a drop in hit rate, if they have unused space, and (b) if they do not have unused space, *MemSweeper* has minimal negative impact on the hit rate (even at peak loads). Our experiments confirm this behavior and show *MemSweeper* 's performance for the victims to be very close to static allocation (Section 5) — the victims are unlikely to experience a noticeable degradation.

On the other hand, customers who pay for a smaller cache will get only best-effort service — they can get better performance if there is spare capacity but this cannot be always guaranteed. We think that customers requiring a hard guarantee will find it acceptable to pay more for a larger cache.

There is a trade-off between utilization, isolation, and fairness. Perfect isolation and fairness results in poor utilization (discussed in Section 5 — Figures 5 and Figure 6). One of our contribution is to make this trade-off such that we achieve *disproportionate* gains in memory utilization while achieving isolation close to static (ideal) allocation. Lastly, our design avoids thrashing when more than one memory-hungry tenants are co-located on the same machine (Section 5.5).

In summary, we make the following contributions:

- By reducing the time complexity for calculating reuse distances from *O(N)* (prior approaches [10, 21, 31]) to *O(Log N)*, our design is able to find far-off cliffs with high accuracy and low cost (Sections 2.2 and 3.1)
- We propose *MemSweeper*, a memory management mechanism that effectively performs memory management among multiple cache tenants. *MemSweeper* accurately estimates the working sets and allocates memory to the tenants based on utility and fairness.
- We introduce a *score* metric that succinctly incorporates both fairness and utility in a closed form. Our evaluations demonstrate the effectiveness and robustness of this metric.
- We evaluated a real implementation of *MemSweeper* on *CloudLab*, using a combination of synthetic and production traces. Our evaluation shows that *MemSweeper* achieves about 74% improvement (on average) in the miss rate of critical tenants without negatively affecting other tenants.

2 Motivation

Memcached is an in-memory cache that can be distributed among hundreds of servers receiving queries e.g., SET, GET, UPDATE etc., from clients using a key-value API. The servers are divided into different pools with each one handling a specific category of client applications (i.e, object sizes, access patterns). Servers use a hash table to track and access the items stored on their allocated space in the main memory of the system. The *O(1)* access time for the stored items, make the key-value memory caches desirable for servers that handle thousands of requests every second. To prevent fragmentation, memory is divided into different slabs; items are categorized into slab classes based on their size, and stored on the memory pages assigned to that class. Furthermore, each page is divided into smaller chunks to store the objects and their metadata. The size of the these pages is usually 1 *MB* and they are allocated to a class only if the cache has empty space. Otherwise, the slab class will have to empty one of its owned pages by evicting items using a heuristic

e.g., Least Recently Used (LRU). Items might also get evicted from the queue when not accessed for a long time.

Prior work [10, 11] has shown that reallocating pages from one slab class to another inside a tenant, can improve the hit rate, as the demand of each slab class might change over time. Thus, having the lower-demand large classes give their pages to higher-demand smaller classes results in better memory utilization and a higher hit rate. However, lots of opportunities for more improvements are missed in previous work due to restricting the pool of available pages to the same tenant. We make the key insight that in a multi-tenant environment, we can reassign memory not only between classes of a tenant, but also among tenants e.g, take memory from tenants who need lower space at the moment and give it to more demanding tenants in real time.

2.1 Multi-tenant Space Sharing

The allocation mechanism in Memcached is slab-based. Memory is divided into 1 MB pages (slabs) and each page is assigned to a certain slab class (queue) of a tenant using a first-come-first-serve basis. The pages are given to the classes requesting memory before others. As a result, the classes that are popular during the cache startup will occupy most of the space, even if their popularity changes afterwards. This well-known problem is referred to as *slab calcification* [2, 18]. Once a slab (i.e., 1 MB) is allocated to a class, it cannot be assigned to a different class or tenant. Although, Memcached has a reallocation mechanism in form of a thread, which automatically moves pages between classes, it's been shown that it is not able to efficiently adapt to dynamic workloads or prevent memory fragmentation [37].

In addition to slab calcification, performance cliffs are common in real-world production workloads, in which a sudden change in memory allocation leads to significant increase or decrease in the hit rate [4, 10] e.g., when the working set of an application suddenly fits in the cache. Talus [4], proposed an algorithm to remove a performance cliff in the hit rate curve of a single slab class of a single tenant. It needs to have the knowledge of the entire hit rate curve and the exact points where the cliff starts and ends, so it can simulate the behavior of the cache where the cliffs does not exists, by using a technique called *shadow partitioning*.

Based on Talus and Dynacache [9], Cliffhanger [10] assumes it does not have the entire hit rate curve and instead, tries to find the cliff's start and end points by looking ahead and approximating the gradient of the local hit rate curve. It can improve the hit rate by reallocating memory from larger lower-demand classes to smaller higher-demand classes. Cliffhanger does all the reallocations internally without considering the other tenants. Enabling the cache to move memory among tenants, creates this opportunity to improve the hit rate even more (Figure 1). By looking ahead in the hit rate curve, Cliffhanger and similar methods detect only those changes that are close to the current position

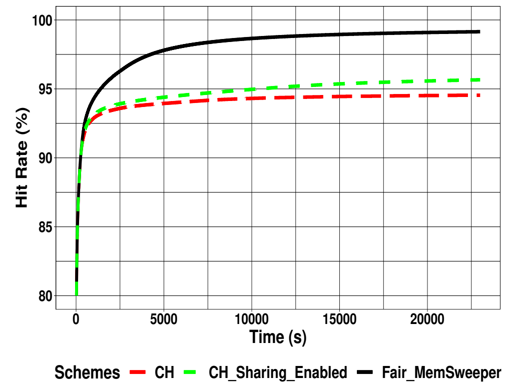

Figure 1. Cliffhanger vs. Cliffhanger with sharing enabled vs. *MemSweeper*

in the curve i.e., small size difference. In case of a performance cliff, it is undetectable for these systems when it is far away in the curve i.e., there is a huge difference between the current size and the size in which the cliff will occur.

Memshare [11] utilized Cliffhanger's approach to dynamically share memory between applications in a log-structured cache. By using a log-structured model, Memshare tries to avoid the problems that might arise in slab-based caches when tenants exchange memory such as low flexibility in size of the moved memory. However, it still suffers from inability to detect far-away cliffs and the overhead from managing a log-structured cache. To demonstrate how these opportunities i.e, memory sharing among tenants, and far-away cliffs are missed, we did an experiment with a workload having both within reach and far-away demand changes (Figure 1).

2.2 Reuse Distance Calculations

In order to look ahead in the MRC (miss ratio curve) and find the demand changes correctly, we needed an efficient and accurate method to track the reuse distance of cached items. In caching systems, *reuse distance* is defined as the number of unique items accessed between two consecutive accesses to the same item [13]. In an LRU queue, it is the same as the concept of stack distance (introduced in [21]), which can be measured by counting the items between the head of the queue and the accessed item. The reuse distances can be utilized to predict the curve in the miss rate vs. cache size (number of cached items) i.e., MRC. A distance value *larger* than the cache size implies that item was once in the cache and it was evicted, which means in a larger cache (a future point in the x-axis of the MRC), this item would be a hit. Depending on the number of found reuse distances, we can approximate the MRC.

A simple reuse distance calculation approach would take O(N) time for an LRU queue i.e, a linked list of size N , as it has to traverse the queue from the head until it reaches

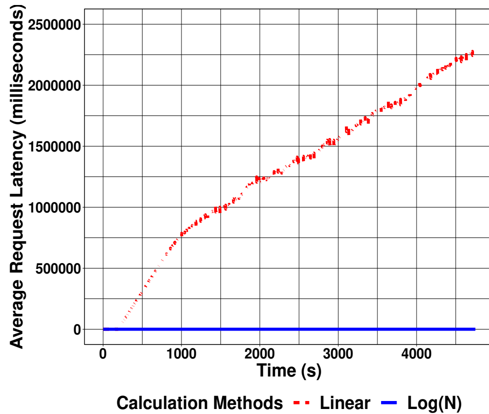

Figure 2. Reuse Distance Computation - Linear vs. Log N

the item [21]. A linear time complexity is extremely prohibitive for a cache with millions of items and thousands of requests per second (Figure 1). New approaches have been proposed to improve the reuse distance calculations [13, 31, 38]. Mimir [31] presented a bucket algorithm that groups references to the items in a specific number of buckets(B), reducing the time complexity to $O(\frac{N}{B})$. This approach is not accurate for large caches [10]. By using *search trees* to keep track of references to the items [13], the time complexity of reuse distance calculations were reduced from linear to logarithmic. In this approach, a balanced search tree, for example, an AVL tree is maintained in addition to the main data structure holding the items. In our implementation, the AVL tree is used to organize and keep track of the last-access time of the items. Each node in the tree would also store a *weight* attribute showing the number of older nodes in its sub-tree. For a queue size of N, the reuse distance computation takes O(Log N) time, as the item and its reuse distance are found in *Log N* iterations (more details in section 3).

Figure 2 demonstrates an experiment on linear and logarithmic reuse distance calculation approaches. The experiment compares the average latency of requests for two equal-sized Memcached tenants that are receiving the same RPS (requests per second) and calculating the reuse distance of every missed item. In order to create the worst-case scenario, all requests are missing the cache, the size of the tenants were set smaller than the workload size, and the benchmark followed a sequential access pattern. As shown, the linear approach was extremely slow, causing the tenant to become unresponsive, servicing significantly lower number of requests, and in some cases, zero requests i.e., the numerous breaks in the red curve, whereas the logarithmic approach suffered no loss in the quality of service.

3 Design and Implementation

In *MemSweeper*'s environment, all tenants reside in a pre-allocated POSIX shared-memory. We replaced Memcached's memory allocation routine i.e., malloc() with a function that uses mmap() to allocate memory in the shared-memory. This memory is managed using an entity called the *tracker*, which performs memory allocations and stores management metadata (i.e., one page) for inter process communication (IPC). Normally, Memcached tenants are allocated memory only if they need to store new items and have no available page. We anticipated a preset share size from the shared-memory for each tenant that guarantees the availability of new pages for the tenants that have paid for a certain memory size, but have not requested to that amount yet. By default, all tenants have an equal share from the shared-memory, unless they paid for more space. When a tenant asks for a page, the next available page in the shared-memory is given to the requesting tenant, as long as tenant's current size is less than or equal to its share. Neighboring pages might belong to different tenants.

In *MemSweeper*, from the memory management point of view, we look at the multiple tenants sharing the space, as one large tenant allocated a large piece of memory i.e, the shared-memory. In Memcached, each tenant has its own address space. In MemSweeper, tenants are still separate processes, but they operate within a single address space, which is the shared memory. Tenants cannot access (Read/Write) the pages allocated to other tenants due to existing OS page protection mechanisms (i.e., SIGSEGV).

This approach leads to much more flexibility for performing reallocations when tenants see changes in their workloads. We propose an efficient reallocation mechanism that is able to accurately detect the changes in the demand of different tenants and slab classes. Not only it is able to reallocate the memory page between the classes of one tenant, it can also move them among different tenants, and find far-away performance cliffs. To prevent tenants from reading the data of other tenants when pages are exchanged, *MemSweeper* explicitly zeros out the contents twice: once when a tenant is giving the memory away, and again when a new tenant receives the ownership. We rely on existing system support for efficiently zeroing out memory.

The reallocation mechanism consists of two main steps: 1) calculating the reuse distances for all data items, and 2) reallocation algorithm. Using *shadow queues* and a self-balancing search tree i.e, AVL tree, we track all the misses and calculate the reuse distances for the missed items. The shadow queues are extensions to the main LRU queues of the cache, keeping the items evicted from the main queues in the same LRU format [10]. To reduce the memory overhead, items in the shadow queues (i.e., shadow items) only store the keys and not the values. When a GET request misses the main queue, and hits the shadow queue, it means the request would have been a hit, if the item was not evicted from the main queue (if the queue was larger). Each slab class has a shadow queue which is divided into pages that are not equal in size to the pages of the main queue, but in the number of items stored

in each page. Depending on the position of the shadow page receiving a hit, the algorithm might decide to allocate more memory to the corresponding slab class in the owner tenant.

These decisions are based on several factors, one of which is the *marginal utility*. We define Marginal Utility(MU) [29] as the per-page hit gain for a continuous block (a group of one or more adjacent pages) in the shadow queue. In the initial version of our algorithm, once the marginal utility of a shadow page reaches a certain threshold, the tenant will request for more space to the amount of the position of the aforementioned page in the shadow queue. These values are simply calculated by dividing the total number of shadow hits in a block by the size of the block (Algorithm 1).

Algorithm 1 Marginal Utility Calculation

```
1:  slabclass[i].shadowq_hits[]    /* latest hit counters of all shadow pages */
2:  slabclass[i].mu[] ← 0
3:  while miss_rate != 0 do
4:      If there is no reallocation in progress
5:      for i = 1, 2, . . . number of slab classes do
6:          for j = 0, 1, . . . shadow queue size do
7:              slabclass[i].mu[j] =
                Sum(slabclass[i].shadowq_hits[1,..,j]) ÷ j
8:          end for
9:      end for
10:     Sleep(1)
11: end while
```

In order to allocate the requested memory to the tenant, the algorithm needs to find a suitable candidate to take the memory from. This tenant should either have unused memory, in which case, it will not suffer at all (in terms of hit rate), or suffer the least from releasing one of its old pages. In the first version of our system, the algorithm finds the page with the lowest marginal utility in the shared-memory, which is calculated based on the actual hit rate of each page in every slab class. These values are not normally tracked in Memcached, so we added them as an extra attribute to Memcached's slab class structure. The page with the lowest marginal utility among all pages of all tenants is de-allocated from the owner i.e, the *victim* and given to the *victor* tenant. The pages are moved one by one (1-page granularity), meaning that if the victor tenant needs X pages, all X pages will not be taken from a single tenant, and a victim is found for each page (might be the same victim). This approach of having a (possibly) new victim for each page is more fair and precise than choosing one victim for all X pages, since releasing even a single page can lead to a notable change in the hit rate of a tenant, making it a less suitable victim candidate. In other words, not finding a new victim for each page can cause additional and avoidable reallocations in the future, as the same tenant will ask for more memory i.e, wants its pages back, due to suffering too much during the first reallocation in which it was the victim.

This approach has a few shortcomings. If there is an aggressive tenant with a high RPS (requests per second) and miss rate, it will take all the memory from other tenants with lower RPS, causing them to suffer too much. In order to prevent this problem and control the amount of possible reallocations between the tenants, we anticipated two kinds of thresholds: 1) Threshold on size, in which the number of moved pages is limited for both victims and victors, and 2) Threshold on hit rate, in which the victims are allowed to lose pages as long as their hit rate has not dropped more than a certain value. All these systems are evaluated in section 5.

We made the algorithm more efficient and fair by considering more factors such as the number of pages a tenant has gained or released prior to the reallocation, or how long a tenant has waited for a new page, in the decision making process. All these factors are incorporated in form of an *score* metric that is used to perform reallocations in the newer version, *Fair_MemSweeper*. We believe *Fair_MemSweeper* is efficient and more fair, while performing slightly worse in terms of utilization compared to *Max_MemSweeper*. In *Fair_MemSweeper* aggressive tenants cannot hurt the ones that have a lower RPS (discussed in section 5.5). In the following sections, we will discuss this algorithm in more details.

3.1 Calculating Reuse Distances

Similar to Cliffhanger [10], we utilize shadow queues: extensions to the main LRU queue of each slab class that contain the keys of evicted items. When a GET request misses the main queue, but hits the shadow queue, it means if the main queue was larger, this request would have been a hit. Prior systems such as Cliffhanger use very small shadow queues e.g., one page, and approximation techniques to avoid the cost of reuse distance calculations. As discussed before (Section 2.1), this approach misses the far-off cliffs. On the other hand, a large queue implemented as a linked list with a linear approach for the calculations would result in an extremely slow, and in some cases, unresponsive system (as discussed in section 2.2), since the whole list has to traversed for every access to a distinct data item, with the time complexity of *O(N)* (N is the number data items). *MemSweeper* does not evict items from the shadow queues, enabling it to keep track of all past items. However, this approach might lead to large queues (in terms of number of items, not memory) for popular slab classes depending on the workload. Thus, we had to think of a better approach to calculate the reuse distances.

We replaced the linked list with an AVL tree, in which every item (node) contains the number of their older items (size of their right sub-tree) as a *weight* value. Weights are updated on insertions and deletions. To find the reuse distance, we simply traverse the tree from the accessed node toward the root and if the current node is the left child of its parent, we add up the parent's weight value. The total sum would be the reuse distance of the requested item found in Log N iterations (height of the tree). This results in the

time complexity of *O(Log N)* for the reuse distance calculations. The reuse distances are then utilized to determine the number of hits on every shadow queue page.

The *Lookahead Algorithm* introduced in [29], can partition shared CPU caches according to the utility(hit) gain of extra cache blocks for different applications sharing the cache with a low overhead. Inspired by this algorithm, *MemSweeper* takes advantage of the knowledge of hit rate of the shadow pages to find the marginal utilities (potential hit gain from all possible additions to the main queue), which subsequently enables it to reallocate memory to the tenants that would benefit from more space from those who can afford to lose space. Additionally, *MemSweeper* detects the performance cliffs occurring far away from the current position in the hit rate curve. A dedicated thread is tasked with periodically finding the marginal utilities of all shadow queues.

One tenant has to give up a page. *Max_MemSweeper* picks the tenant that has the lowest marginal utility value among all the tenants as the victim. For doing so, the system needs to keep track of the hits on each individual page in the main queue, which are updated right after each hit request. The thread responsible for calculating the marginal utility of shadow pages, does the same for the the main pages without any additional overhead. No reuse distance calculation is needed for the main queues as the hit page is identified simply by accessing the item.

3.2 Reallocation Algorithm

The goal of this step is to identify two candidates: one that would benefit the most from extra memory, and one that would suffer the least from giving one page away. Considering the marginal utility values of all pages of a tenant, *Max_MemSweeper* reallocates the page with the lowest number of hits to the tenant that would (potentially) gain the highest number of hits. The tenant with the lowest S_0 of main queues (Formula 1.2), and the one with the highest S_0 of shadow queues (Formula 1.1), are chosen as the victim and the victor, respectively.

$$Shadow\ Queue: \quad S_0 = \max\left(MarginalUtility\right) \tag{1.1}$$

$$Main\ Queue: \quad S_0 = \min\left(MarginalUtility\right) \tag{1.2}$$

Although, exchanging a low hit, under-utilized page for a potentially high hit and fully utilized page would result in reallocations that would maximize the overall memory utilization and hit rate, it might be unfair to some of the tenants. As it will be discussed further in section 5.5, victims might get over-penalized or taken advantage of by the victors. To solve these issues, *Fair_MemSweeper* extends S_0 to consider more factors in deciding the reallocation sides, aggregated in form of a *score* attribute. Each tenant calculates S_0 and S_1 for the main and shadow queues periodically. The system reallocates memory from the tenant with the lowest main queue S_1 to the tenant with the highest shadow queue S_1.

$$Score: \quad S_1 = S_0 \times \left(\frac{N_r}{N_g}\right) \times MR_{ins} \times t \times \frac{1}{M} \tag{2}$$

Marginal utility is measured for all pages of a tenant x: 1 < x < n (n: size of the queue), which basically shows the per page hits, the tenant would gain by obtaining x pages, or lose by releasing one page. N_r and N_g are the total number of pages released and gained by the tenant (respectively), since the beginning which are never reset. A decay factor can be utilized to reduce the impact of these elements depending on the policy. The ratio of released to gained memory i.e, N_r / N_g is a critical factor meant to prevent tenants from suffer or gain too much, and hold a balance in the reallocations (discussed in section 5.5).

MR_{inst} denotes the instantaneous miss rate i.e, number of misses in the last second. The time interval for score calculations and reallocations is 1 second, and by adding the miss rate to the score calculations, we ensure that the tenants experiencing more misses are prioritized over those having fewer misses in the last time interval. The time window is set to the minimum (1 second), since longer periods might lead to overlooking the sudden misses caused by the workload behavior e.g, performance cliffs. T is the time elapsed since the last time this tenant gained a new page, and M is the total size of the tenant. The longer a tenant has waited for extra pages, the higher would be its chance of receiving one. Algorithm 2 is the pseudo-code of *Fair_MemSweeper*.

In order to show that the score is a *robust* metric able to *fairly* identify the victors and victims, we will demonstrate a simplified example with 3 tenants and a small number of pages. Then, we will present the performance of this metric in benchmarks testing the extreme cases i.e, the workload size much smaller or larger than the available memory.

3.2.1 Step-by-step Demonstration.

In this example, we consider 3 tenants sharing 60 pages. All tenants need 20 pages each, but tenant A is initially allocated 10 pages, tenant B is allocated 15 pages, and tenant C has 35 pages. We run the algorithm on these tenants, calculate the score values in each round, and demonstrate how this metric can correctly identify the victim and the victor (shown in table 1).

We assume the workload size is 20 pages, all items belong to a single slab class, and each page has 10 accesses per second. The first 20 pages of all tenants are accessed 10 times each, and extra pages (more than 20) will have zero accesses. If a tenant is missing 10 pages, then it will have 10 * 10 = 100 hits, and 10 * 10 = 100 misses per second, which is equal to 100 hits on the shadow queue. Therefore, in this example, the scores for both main and shadow queues will be equal for each tenant, as the marginal utility of all pages (shadow and main) would be 10 hits per page for tenant A and B. Tenant C has 15 pages that are not receiving an hits, so the minimum marginal utility among pages of this tenant would be 0, while it has free space. For the sake of this demonstration, we set

Table 1. Step-by-step Demonstration

		ShadowQueue Score			MainQueue Score			Memory Allocation (MB)		
		Tenant A	Tenant B	Tenant C	Tenant A	Tenant B	Tenant C	Tenant A	Tenant B	Tenant C
Rounds	0	0	0	0	0	0	0	10	15	35
	1	**100**	33.3	0	100	33.3	0	**11**	15	34
	2	40.9	**66.6**	0	40.9	66.6	0	11	**16**	33
	3	**81.81**	12.5	0	81.81	12.5	0	**12**	16	32
	4	22.2	**25**	0	22.2	25	0	12	**17**	31
	5	**100**	5.88	0	100	5.88	0	**13**	17	30
	6	**13.46**	11.76	0	13.46	11.76	0	**14**	17	29
	7	8.57	**17.64**	0	8.57	17.64	0	14	**18**	28
	8	**17.14**	2.77	0	17.14	2.77	0	**15**	18	27
	9	**5.5**	5.5	0	5.5	5.5	0	**16**	18	26
	10	3.57	**8.33**	0	3.57	8.33	0	16	**19**	25
	11	**7.14**	1.05	0	7.14	1.05	0	**17**	19	24
	12	**2.20**	2.10	0	2.20	2.10	0	**18**	19	23
	13	1.23	**3.15**	0	1.23	3.15	0	18	**20**	22
	14	**2.46**	0	0	2.46	0	0	**19**	20	21
Final		**0.52**	0	0	0.52	0	0	**20**	20	20

the reallocation granularity to one page, which means only the hits on the first shadow page matter. Round 1:

- A: $Score_{\text{shadow, main}} = (10 \times 1 \times\ 100 \times 1/10) = 100$
- B: $Score_{\text{shadow, main}} = (10 \times 1 \times 50 \times 1/15) = 33.3$
- C: $Score_{\text{shadow, main}} = (0 \times 1 \times 0 \times 1/35) = 0$

After this round, tenant C and A are chosen as the victim and the victor, respectively, since they have the lowest main queue and highest shadow queue scores. Tenant C gives one page to tenant A. Round 2:

- A: $Score_{\text{shadow, main}} = (10 \times (1/2) \times\ 90 \times 1/11) = 40.9$
- B: $Score_{\text{shadow, main}} = (10 \times 1 \times 50 \times 2/15) = 66.6$
- C: $Score_{\text{shadow, main}} = (0 \times 1 \times 0 \times 2/35) = 0$

After this round, tenant C is chosen as the victim again, since it has the lowest main queue score and gives 1 page to tenant B, which currently has the highest shadow queue score. In the next rounds, tenant C has a main queue score of zero, and will always be the victim, due to having free space and a zero miss rate. The other two tenants will have non-zero main queue scores, since they do not have free space and releasing a page will hurt them by losing 10 hits per second. The rest of the steps are shown in Table 1.

In practice, the victor can ask for more than one page in each round. In the first round, tenant C would give 10 pages to tenant A (Tenant C will be the victim for every page). The working set of tenant A will fit in the cache, resulting in a zero miss rate after round 1. In round 2, tenant B would receive 5 pages from tenant C. The reallocation finishes in just two rounds, and unless the workload changes, all tenants reach a steady state.

3.2.2 Extreme Case Benchmarks. Next, we discuss two scenarios with 2 tenants having different workload sizes,

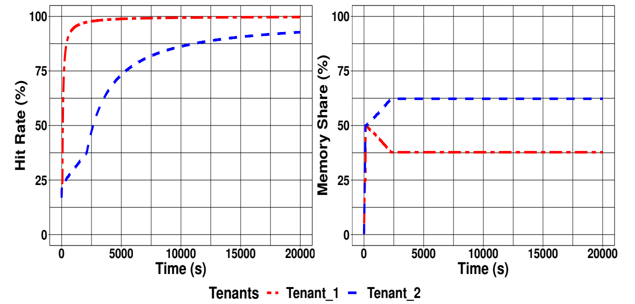

Figure 3. Tenant 2's demand fits in the memory. Left: Hit Rate over time, Right: Memory share over time.

and show how the system correctly reallocates memory by the use of the score metric. In the first scenario, tenants 1 and 2, each had a 50% share of the memory. But, tenant 1 only needed 40% and had 10% extra space, whereas, tenant 2 needed 60% of total space. As shown in Figure 3, the memory shares of the tenants are adjusted to their working set size.

In the second scenario, tenants initially started with a 50% share of the total cache space. Same as the previous scenario, one tenant needed memory less, and the other needed memory more than its already owned share. In this case, tenant 2's demand was more than the total available memory i.e, size of the shared-memory. Figure 4 shows that no memory was reallocated during the benchmark.

4 Methodology

We implemented our proposals on top of Memcached.[1] These include the *Max_MemSweeper*, using which the memory

[1]Source code is available at: https://github.com/AAMH/memcached.

Algorithm 2 Victor-Victim Selection

```
1: tenants[i]
2: victor_scores[i] = 0, victim_scores[i] = 0
3: for iteration = 1, 2, ... do
4:   for each tenant i do
5:     max_mu[i] = get_max_mu(i)
6:     NumberOfMisses[i] = get_inst_misses(i)
7:     update_values(i, M[i], Nr[i], Ng[i], T[i])
8:     victor_scores[i] = calculate_score(M[i], Nr[i], Ng[i],
       T[i], max_mu[i], NumberOfMisses[i])
9:   end for
10:  victor = find_highest(victor_scores)
11:  for j = 1, 2, ..., requested_size do
12:    min_mu[i] = get_min_mu(i)
13:    update_values(M[i], Nr[i], Ng[i], T[i])
14:    victim_scores[i] = calculate_score(M[i], Nr[i],
       Ng[i], T[i], min_mu[i], NumberOfMisses[i])
15:    victim = find_lowest(victim_scores)
16:    Reallocate 1 page from the victim to the victor
17:  end for
18: end for
19: calculate_score (M, Nr, Ng, T, mu, mr):
20:   return mu × (Nr ÷ Ng) × mr × T ÷ M
21: update_values(i, M, Nr, Ng, T):
22:   M ← Latest memory size of tenant i
23:   Nr ← Latest number of released pages for tenant i
24:   Ng ← Latest number of gained pages for tenant i
25:   T ← Time elapsed since tenant i received a new page
26: get_max_mu(i): /* finds the max mu value among all slab classes */
27:   max = max marginal utility of all pages of tenant i
28:   return max
29: get_min_mu(i): /* finds the min mu value among all slab classes */
30:   min = min marginal utility of all pages of tenant i
31:   return min
```

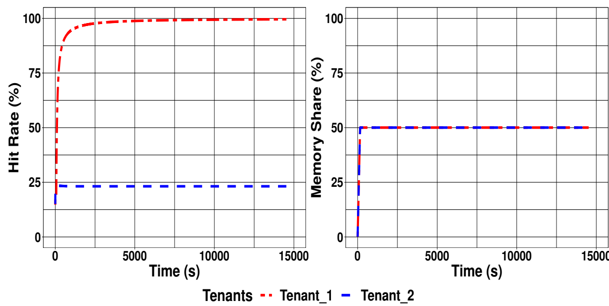

Figure 4. Tenant 2's demand does not fit in memory. Left: Hit Rate over time, Right: Memory share over time.

pages are reallocated between tenants solely based on the marginal utility of each page, and the *Fair_MemSweeper*, using which the memory is moved between tenants based on a score metric that considers more factors to incorporate fairness in the process. The reallocations between tenants will result in positive or negative changes in the hit rate of the involved tenants. The hit rate of the victor i.e, the receiver will improve, but, the victim might suffer from having a lower amount of memory and its hit rate might deteriorate. Using two thresholds i.e., size (MTh) and hit rate (HRTh), we enforce a simple constraint to prevent tenants from suffering too much when losing memory. Both thresholds have to be manually set before starting the system, and in most cases, they limit the potential of the reallocation mechanism. We consider them as two baselines for comparison against Fair and *Max_MemSweeper*, in addition to the base Memcached and Cliffhanger's *Hill Climbing* implementation.

4.1 Workloads

The primary trace we use in our experiments, is a scalable pre-built Twitter dataset as a part of the Cloudsuite benchmarking tool [15]. In the first experiment, the keys and values are selected randomly using a uniform distribution. We refer to this workload as the *Random* workload. To evaluate the performance of our systems in presence of a performance cliff, we added additional features to the Cloudsuite client, so it could generate these cliffs by utilizing a sequential access pattern. We use a combination of random and sequential access patterns in the third workload. We ran additional benchmarks using the Twitter dataset in Cloudsuite, and Mutilate [19], a workload generator that is able to simulate a request stream from the Facebook's ETC pool [3]. These benchmarks focus on studying the tenants' behavior when running on two different workloads with contrasting characteristics. We also studied the case in which more than one tenants were memory-hungry.

All the benchmark settings used in our experiments were designed to simulate a *multi-tenant* cache environment. In each benchmark, *four* tenants shared the underlying resources such as CPU, network bandwidth, and most importantly the main memory. We allocated a 16 GB shared-memory as the total space for the cache environment equally divided between the tenants (Table 2). In all benchmarks, one tenant, designated as the *demanding* tenant, was assigned a working set size (WSS) larger than its initial memory share, so it would request for more memory during the benchmarks. Other three tenants had an initial memory of larger than or equal to their working set, so they did not need extra space.

In the last experiment, we compared the performance of Fair and *Max_MemSweeper*, by having two memory-hungry tenants share memory and receive request in different rates. This experiment demonstrates how an aggressive tenant can hurt another tenant that has a similar working set but a lower request rate. We show how *Fair_MemSweeper* and the score metric successfully prevent such situations that are common in cloud environments.

Table 2. Workloads - WSS of All Tenants (GB)

Workload	Tenant 1	Tenant 2	Tenant 3	Tenant 4
A	4	4	5	3
B	4	4	6	2
C	4	3	7	2
D	4	2	8	2
E	3	2	9	2
F	2	2	10	2
G	4	4	6	4
H	4	3	7	4
I	4	4	8	4
J	3	4	9	4
K	4	4	10	4

4.2 Metrics

We mainly use the individual hit rate of tenants, as well as the overall hit rate of the system to evaluate our proposal against the baselines. We will discuss the memory, CPU overhead and the fairness of our system.

5 Evaluation

In this section, we will present the results of the evaluation of our system, using CloudSuite benchmarking tool and CloudLab, a research cloud infrastructure [14]. We ran our benchmarks on rs630 physical nodes, having a 10-core 2.60 GHz Intel(R) Xeon(R) E5-2660 processor, 256 GB of RAM, and running Ubuntu Server 18.04. We used a total number of 44 tenants in 11 benchmarks for each workload category.

5.1 Random Workload

In this workload, both the key and the value of every request are selected randomly using a uniform distribution. We ran a total of 11 benchmarks. The tenant caches were warmed up (i.e., items were pre-loaded into the cache) and all of them were initially assigned an equal share of one-fourth of the shared-memory. Each tenant was connected to a single Cloudsuite client using 25 TCP connections receiving 25K requests per second (RPS). In all benchmarks, the memory size and the hit rate of the tenants were recorded over time until they reached a stabilized state.

Fair_MemSweeper results in an average of 62% of miss ratio improvement over the base Memcached, followed by *Max_MemSweeper* which gives a 58% improvement over the base. *Max_MemSweeper* with a threshold on size improves the miss ratio the least by 30%.

5.2 Sequential Workload

In order to demonstrate the *MemSweeper*'s ability to detect and overcome performance cliffs, we utilized a customized workload in which items were accessed sequentially rather than randomly. In caches smaller than the length of the sequence, all GET requests will be *missed*, due to the fact that Memcached's LRU (Least Recently Used) policy evicts the items before they are requested. Unless the size of the cache is adjusted to contain all of the items of the sequence, there would not be any hits. We modified the CloudSuite's client to produce these cliffs for this workload. All benchmark settings were similar to the random workload, except the tenants were initially *cold* (not warmed up), and the items were loaded into the cache manually after they were missed in a GET request. The length of the sequence was set large enough for creating a far-off performance cliff in hit rate over time curve.

The base Memcached did not get any hits, since it is not able to adjust the cache size in presence of these behaviors such as cliffs. The working set could not fit in the cache, resulting in a near zero hit rate. Cliffhanger exhibited a similar behavior, as it is not able to detect and mitigate the performance cliff happening far-off in the hit rate curve. All *MemSweeper* systems were able to detect the cliff and adjust memory of the tenants, adapting them to their working sets which resulted in a significant improvement in the hit rate of the demanding tenant. *Max_MemSweeper* decreased the miss ratio of the demanding tenant by an average of 85%. This percentage for *Max_MemSweeper* with a threshold on hit rate and *Fair_MemSweeper* were 80% and 78%, respectively.

5.3 Random-Sequential Combination

We combined the random and sequential workloads to study *MemSweeper*'s behavior when performance cliffs are steep. In this workload, out of every four keys, three are picked from the sequence, and one is chosen using a uniform random distribution. The benchmarks settings were the same as previous sets. The Memory size and the hit rate of the tenants were recorded from the start to the steady state.

Hit rate of the demanding tenant and the overall hit rate are shown in Figure 5 and Figure 6, respectively. Workload A has the smallest Working Set Size (WSS) for the most demanding tenant (5 GB) among all the workloads and the cliff happens to occur at a size less than the WSS. So, the tenant already starts with enough memory (4 GB) to fit most of its WSS in the memory and, therefore, achieves a high hit rate for all schemes. In all other workloads, the base Memcached reached an approximate hit rate of 25%. Similar to the previous case, the requests picked from the sequence were all misses, and the only hits were the requests coming from the random distribution. A similar behavior could be observed from Cliffhanger, as its small shadow queues are unable to find the far-off cliffs in the hit rate curve. On the other hand, all *MemSweeper* versions could detect the far-off cliffs and adjust the memory sizes, boosting the overall hit rate of the system. As shown in Figure 5, *Max_MemSweeper* achieves the highest improvement for the demanding tenant by decreasing its miss rate by 82%. *Max_MemSweeper* with a threshold on hit rate and *Fair_MemSweeper* are next with 79% and 74%, respectively. Overall, *Max_MemSweeper* decreases

the miss rate by 75% (Figure 6). Figure 7 shows the hit rate curves of the demanding tenants in all benchmarks.

5.4 ETC Workload

This benchmark shed lights on *MemSweeper*'s performance when running on a simulated production trace, as well as an important scenario that is standard in cloud environments: tenants placed on the same physical server receiving workloads with totally different characteristics. For this purpose, we utilized Mutilate [19], a workload generator that can simulate a request stream from the ETC pool of Facebook's 2012 study [3], and CloudSuite. We placed two tenants on a 8 GB shared-memory. One tenant received the ETC workload, and the other received the random-sequential request stream from CloudSuite's Twitter dataset. The working set of the ETC tenant was 6M records, filling up to 4 GB. The second tenant was tested with two working set sizes of 6 GB and 8 GB. In both cases, *Fair_MemSweeper* was able to adjust both tenants' allocated space, and improve the overall and the demanding tenant's hit rate significantly without any loss in the hit rate of the less demanding tenant. (Figure 8).

5.5 Fairness Study

Approaches such as *Max_MemSweeper* that try to the maximize the overall hit rate and the memory utilization of the system would likely hurt the tenants that are receiving requests on a lower rate or have higher hit rate, favoring aggressive tenants. Meaning, an aggressive tenant with a sufficiently high request per second (RPS) which could be on a path to a large hit rate gain, can hurt the other tenants by bullying them and taking all their extra memory. *Fair_MemSweeper* prevents such problems by considering factors such as the size of the tenant, ratio of released to gained pages, time elapsed since last allocation, etc., in addition to the marginal utility which is crucial in performing the best reallocation and adjusting the memory sizes.

In this benchmark, two memory-hungry tenants compete with each other for memory. Both tenants were initially allocated 4 GB in an 8 GB shared memory. We used our most realistic synthetic workload i.e., random-sequential combination with a cliff happening before 6 GB. The RPS of one tenant (B) was half the rps of the other (Tenant A) e.g, 10k - 20k. Figures 9 and 10 demonstrate their memory size and hit rate over time during the benchmark. *Max_MemSweeper* let the aggressive tenant (shown in the right) to take all memory it needs from the slower tenant, even though the slower tenant received a similar workload but in a lower rate. As shown in Figure 10, not considering the impact of memory loss on the slow tenant leads to a significantly lower hit rate. On the other hand, *Fair_MemSweeper* was not affected by the difference in the RPS, and kept a balance in the reallocations. Table 3 shows the final size and hit rate for both tenants.

Because *Max_MemSweeper* performs allocations based on a single factor (i.e., marginal utility), it tends to allocate more memory to aggressive tenants over time, which may lead to unfair allocation and starvation. To remedy this problem, we designed *Fair_MemSweeper*, which considers others factors such as allocated memory and recency, in addition to utility.

From our experiments, we observed that even though the *Fair_MemSweeper* improves the overall hit rate less than *Max_MemSweeper*, it penalizes the lower demanding tenants (majority of the tenants, 3/4) much less, which is a justifiable trade-off. In this section we captured this trade-off —Figure 10 and Table 3 show that *Fair_MemSweeper* ensures that the two tenants achieve better parity in hit rates, whereas *Max_MemSweeper* does not. We designed *MemSweeper* to achieve a sweet spot in this trade-off between overall hit rate (utilization) and fairness. In environments in which fairness is a priority (e.g., multi-tenant Clouds), *Fair_MemSweeper* is the best option, but in those aiming to maximize utilization (e.g., Datacenter of a single provider), *Max_MemSweeper* might be acceptable.

Table 3. *Max_MemSweeper* vs. *Fair_MemSweeper*

	Max_MemSweeper		*Fair_MemSweeper*	
	Memory	Hit Rate	Memory	Hit Rate
1	2632 MB	60.84%	4852 MB	86.40%
2	5373 MB	96.65%	3153 MB	88.38%

5.6 Memory and CPU Overhead

MemSweeper's reallocation algorithm is based on keeping the evicted items in the shadow queues. The *shadow* items are much smaller than normal items, due to not containing the largest part of the items (the value). In our benchmarks, 10 slab classes covered all the traces, and the largest of these classes was for storing the 480-byte items. The number of keys in each shadow page is equal to the number of keys in a real 1 MB page. Meaning, each shadow page of this class holds 1 MB / 480 bytes = 2184 keys. This class required less than 1000 shadow pages at the peak of the reallocations. For an average key size of 14 bytes [10], the overhead memory for this class would be 1000 * 2184 * 14 = 30.5 MB. In the worst case, the total overhead memory for a demanding tenant was less than 300 MB, 2% of the total size (16 GB).

A High CPU overhead may lead to a drop in the request throughput and increase in the request latency. Fortunately, we observed minimal CPU overhead imposed by *MemSweeper* during our experiments. *MemSweeper* utilizes *multi-threading* to maximize the efficiency and minimize the impact of the reallocation on Memcached's main operations such as processing the client's requests. All the major operations required by *MemSweeper*'s algorithm are performed by separate threads. The eviction of items into the shadow queues is done as an extra step in Memcached's LRU eviction policy. Tracking the hits on the main queue is done as an additional step in the Memcached's STATS thread. For the shadow queues, this

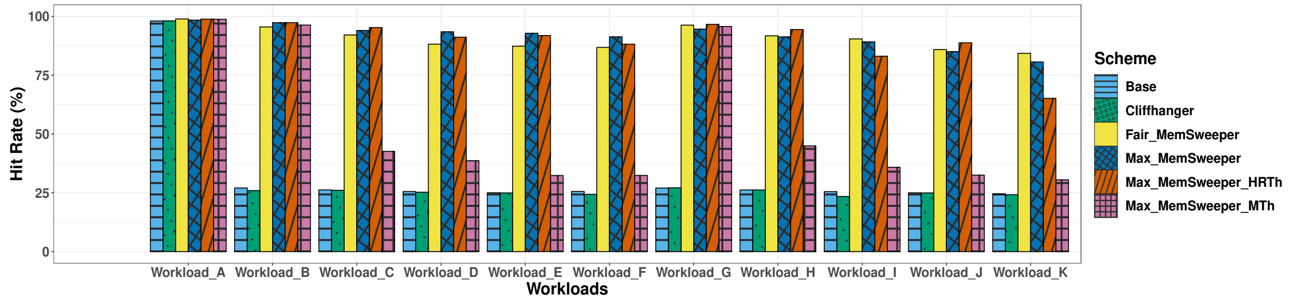

Figure 5. Hit Rate of the Demanding Tenant - Random-Sequential Combination

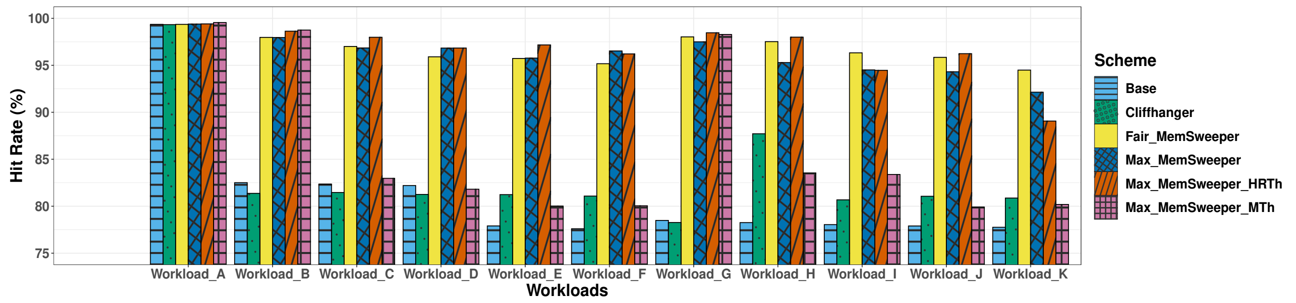

Figure 6. Overall Hit Rate - Random-Sequential Combination

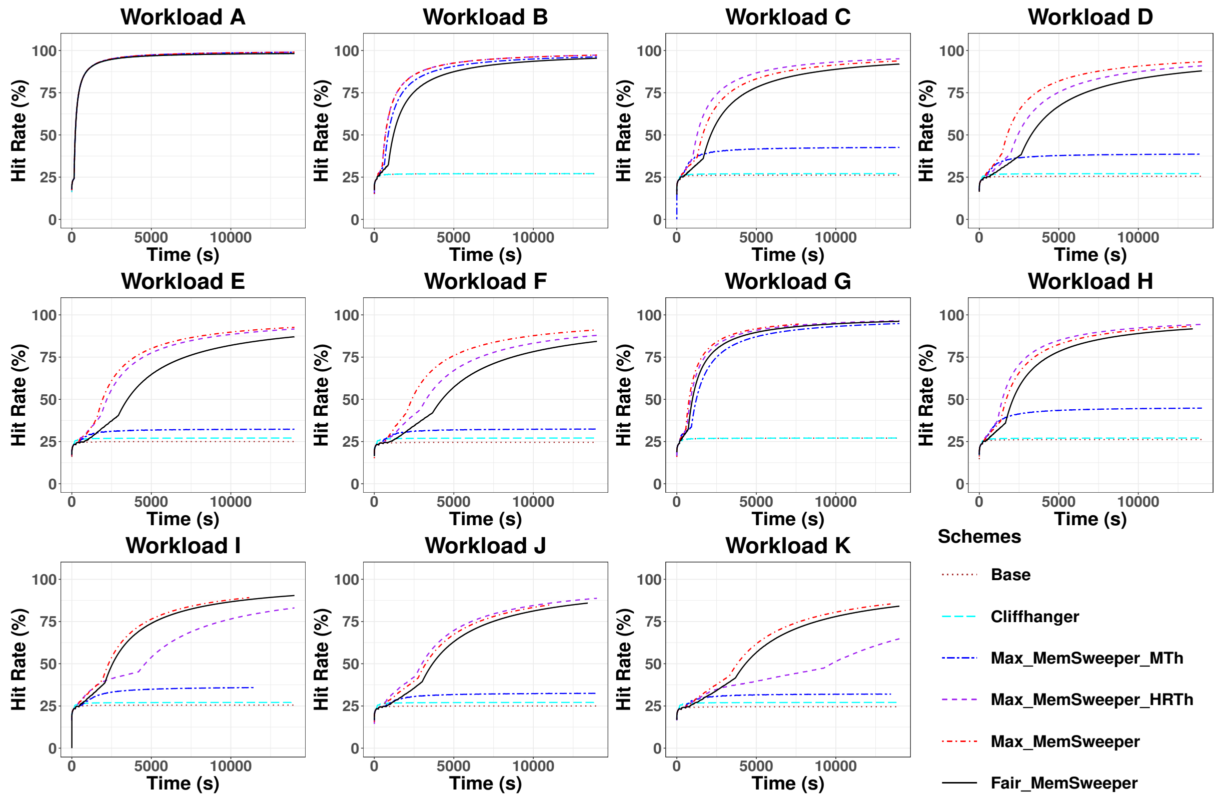

Figure 7. Hit rate of the Demanding Tenant over Time - Random-Sequential Combination

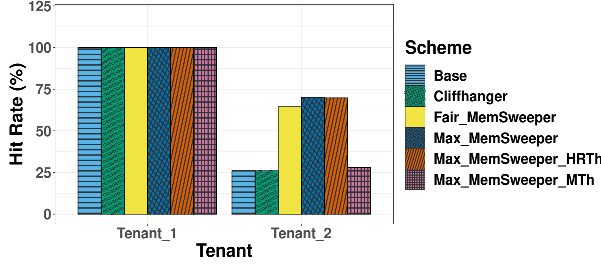

Figure 8. Hit Rate - Tenant 1: ETC, Tenant 2: Random-Sequential Combination (8 GB)

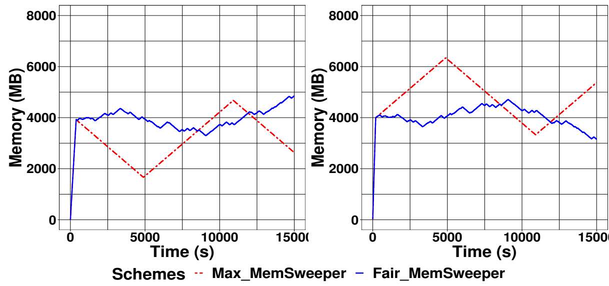

Figure 9. Memory vs. Time. RPS (Left) = 10K, RPS (Right) = 20K

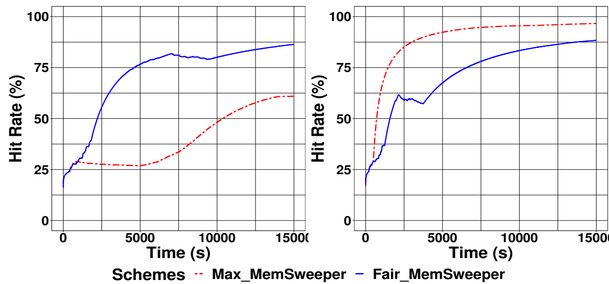

Figure 10. Hit rate vs. Time. RPS (Left) = 10K, RPS (Right) = 20K

step requires finding the reuse distance value of each missed item. Our approach reduces the time complexity of this step to *O(Log N)*, and minimizes the impact of this operation on the latency. The only thread running in addition to the Memcached's routine, is created to calculate the marginal utilities and scores. The reallocations (if needed) are done using the Memcached's self re-balancing thread, augmented to perform inter-tenant reallocations.

6 Related Work

Multi-tenant Caches. Multi-tenant caches have been studied extensively in previous years [7, 8, 10, 11, 20, 33, 37]. We have already discussed Cliffhanger [10]. Memshare [11] is a multi-tenant cache that, similar to Cliffhanger, utilizes the local hit rate gradients to optimize the memory allocation between applications in a log-structured cache. It improves the overall hit rate at the cost of throughput and latency. Segcache [37] has a similar log-structured design that utilizes TTLs (expiration) to group and sort objects in different segments and reduces the metadata of cache objects by sharing them in the segment header, thus reducing the memory footprint. It focuses on improving the eviction mechanism through the TTL-sorted segments and objects, with potentially high computation overhead. It is suitable only for small objects and needs parameter tuning.MPart[7] and eMRC[20] rely on different methods to periodically construct the MRC with costly computations, and optimize the memory allocation among the tenants. The latter only focuses on finding the cliffs in multi-dimensional miss ratio curves and multi-tier caching. Unlike these systems, *MemSweeper* can find all performance cliffs and adapt tenants to their working sets while achieving balance between fairness and memory utilization, with a low computation and memory overhead.

Resource Management. In a multi-tenant environment, resources are shared among tenants. Several previous work has studied the fair allocation of these resources. FairRide [28] utilizes a form of probabilistic blocking to prevent cheating in shared-file environments e.g, Hadoop file system. Dominant resource fairness, proposed in [16], is based on max-min fairness to fairly allocate multiple resource types.

MRC Estimation. Miss ratio curve (MRC) measurement in caches is generally performed by analyzing reuse distances of the cache objects. Tracking the reuse distance is a challenging problem examined in several previous work both in software and hardware caches that utilize sampling [5, 6, 32, 34, 39, 41], parallelization [12, 26, 27, 32], or improved data structures [13, 35, 40]. We take a similar approach to [13] by using AVL trees which results in O(Log N) time complexity. Cache partitioning and resource allocation based on the MRC is studied in [7, 18, 20].

7 Conclusion

We presented *MemSweeper*, a memory management scheme for multi-tenant cache environments that shares memory among cache tenants, performs fast reallocations, and achieves a good balance between memory utilization and fairness. *Fair_MemSweeper* can identify the working sets of the tenants and detect any changes (e.g., far-off cliffs) in these working sets in real-time and adjust the memory sizes accordingly. Using a score-based metric, *Fair_MemSweeper* reallocates extra space from the less demanding tenants to higher demanding tenants, improving the overall and critical tenant's hit rate. *Fair_MemSweeper* has the potential to replace the off-the-shelf caching systems that use static allocations or costly MRC estimation mechanisms.

References

[1] Amazon. 2021. AWS Caching Solutions. https://aws.amazon.com/caching/aws-caching/ [Online; accessed 15-July-2021].

[2] Chris Aniszczyk. 2012. Caching with Twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache [Online; accessed 15-July-2021].

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[4] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 64–75.

[5] Erik Berg and Erik Hagersten. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004.* IEEE, 20–27.

[6] Erik Berg and Erik Hagersten. 2005. Fast data-locality profiling of native execution. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 169–180.

[7] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 84–95.

[8] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2019. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*. 353–362.

[9] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[10] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 379–392.

[11] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 321–334. https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon

[12] Huimin Cui, Qing Yi, Jingling Xue, Lei Wang, Yang Yang, and Xiaobing Feng. 2012. A highly parallel reuse distance analysis algorithm on gpus. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 1080–1092.

[13] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 245–257.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[15] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (March 2012), 37–48. https://doi.org/10.1145/2248487.2150982

[16] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[17] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 845–861.

[18] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 57–69.

[19] Jacob Leverich. 2021. Mutilate. https://github.com/leverich/mutilate [Online; accessed 15-July-2021].

[20] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. 2021. eMRC: Efficient Miss Ratio Approximation for Multi-Tier Caching. In *19th USENIX Conference on File and Storage Technologies (FAST)*. 293–306.

[21] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.

[22] Memcached. 2021. memcached - a distribtued memory object caching system. https://memcached.org [Online; accessed 15-July-2021].

[23] Microsoft. 2021. Azure Cache for Redis. https://azure.microsoft.com/en-us/services/cache/ [Online; accessed 15-July-2021].

[24] Robert Mosolgo. 2021. How we scaled the GitHub API with a sharded, replicated rate limiter in Redis. https://github.blog/2021-04-05-how-we-scaled-github-api-sharded-replicated-rate-limiter-redis/ [Online; accessed 15-July-2021].

[25] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 385–398.

[26] Qingpeng Niu, James Dinan, Qingda Lu, and Ponnuswamy Sadayappan. 2012. PARDA: A fast parallel reuse distance analysis algorithm. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 1284–1294.

[27] Abhisek Pan and Vijay S Pai. 2015. Runtime-driven shared last-level cache management for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

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

[29] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.

[30] Redis. 2021. https://redis.io/ [Online; accessed 15-July-2021].

[31] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. 2014. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.

[32] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 53–64.

[33] AmirHossein Seyri, Abhisek Pan, and Balajee Vamanan. 2019. Dynamically Sharing Memory between Memcached Tenants using Tingo. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*. 40–42.

[34] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST)*. 95–110.

[35] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. 2014. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 335–349.

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

[37] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects.. In *NSDI*. 503–518.

[38] Ting Yang, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. 2006. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 103–116.

[39] Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*. 91–100.

[40] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. *ACM SIGPLAN Notices* 39, 6 (2004), 255–266.

[41] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 6 (2009), 1–39.