Faster Slab Reassignment in Memcached

Daniel Byrne
djbyrne@mtu.edu
Michigan Technological University
Houghton, Michigan

Nilufer Onder
nilufer@mtu.edu
Michigan Technological University
Houghton, Michigan

Zhenlin Wang
zlwang@mtu.edu
Michigan Technological University
Houghton, Michigan

Abstract

Web applications, databases, and many datacenter services rely on in-memory key-value stores to cache frequently accessed data. In this work, we focus on a commonly used system, memcached, where even small performance improvements can result in large end-to-end speed ups in request latency. memcached organizes its memory into slabs that belong to different classes corresponding to object sizes. Many prior works have explored the problem of how many slabs should each class be assigned in the face of dynamic workloads, typically reassigning hundreds of slabs during a reassignment. However, we find that as workloads scale and applications use increasing amounts of memory, the current reassignment mechanism in memcached is inefficient. In fact, we measure that reassignments can take millions of requests to complete.

Motivated by these findings, we introduce a faster slab reassignment mechanism in memcached with minimal changes to existing source code. In our experiments, we show that the time needed to reassign a slab reduces by over 99% resulting in the ability to reach workloads' steady state miss ratio by 53% to 75% faster. By arriving at the steady state miss ratio faster, we reduce the overall average miss ratio by 3.42% to 11.5%.

CCS Concepts • **Information systems → Application servers**; **Cloud based storage**.

Keywords memcached, Memory Allocation, In-Memory Key-Value Stores, Memory Caches

ACM Reference Format:
Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2019. Faster Slab Reassignment in Memcached. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357526.3357562

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357562

1 Introduction

In-memory key-value stores, such as memcached [12], play a critical role in serving requests in the datacenter and cloud environment. These systems reside in the server's main memory and are used to cache data from backend systems, such as databases or recommendation systems.

In-memory key-value stores are deployed in large environments such as Facebook, Twitter, and Microsoft.com [2, 13, 14]. Given the scale and scope of these deployments, even a small improvement in their cache hit ratio can have a large impact on their performance. Consider the following example: A cache has 98% hit ratio, average cache latency of 100μs, and a database access time of 10ms, the expected end-to-end application latency is 298μs (= 0.98 × 100μs + 0.02 × 10000μs). Now increasing the application's hit ratio to 99%, we get an expected end-to-end latency of 199μs, resulting in over a 33% speedup.

As workloads change over time, memcached will reassign memory internally in order to best serve the changing workload. In this work, we will focus on the mechanism that memcached uses to reassign memory. memcached stores data in logical units of memory called slabs. Each slab belongs to a given class corresponding to a fixed size *chunk*. A chunk stores the actual data: key and value pair along with meta information. Figure 1, on the next page, illustrates the layout of memory in memcached system.

The importance of quickly reassigning slabs in memcached is motivated by several recent studies. Locality Aware Memory Partitioning (LAMA) by Hu et al. [10, 11], is a memory allocation scheme implemented in memcached that attempts to maximize the overall hit ratio of the system using miss ratio curves. To arrive at optimal miss ratios, the system relies on reassigning slabs to classes that will benefit the most from additional slabs. Reducing the time it takes to reassign slabs among classes will result in a lower number of total misses. This is because the system will arrive at the steady state miss ratio in a shorter period of time.

A second study that motivates a fast slab reassignment mechanism is Robinhood [2]. Robinhood attempts to minimize the 99th percentile latency of memcached requests by periodically taxing memory from cache rich applications and redistributing to cache poor applications. In this system, the faster it can assign memory to a cache poor application, the lower the number of Service Level Agreement (SLA) violations at the 99th percentile will be.

With multi-tenant workloads (single cache, multiple applications [4]) and changing object size distributions common in most datacenter environments, it is also necessary to move memory between classes as quickly as possible so that application performance is not disturbed. We improve the slab reassignment mechanism in memcached to enable fast slab movement between classes.

In order to illustrate the impact on performance, we give a case where a new size of items is introduced to a memcached instance that has miss ratio curve-based partitioning already enabled [10, 11]. The workload is divided into two phases: phase 1 is 100% items with size 200 bytes, phase 2 is split between items of size 200 bytes (33%) and items of size 250 bytes (66%). Items sized 200 bytes are in class 1 and items sized 250 bytes are in class 2. Phase 1 lasts 13 million requests and phase 2 lasts 87 million requests. Each size distribution belongs to its own class within memcached and has Zipfian access pattern $\alpha = 1.0$, consistent with measured access patterns in typical data center workloads by Breslau et al. [3]. The long transition to steady state in phase 2 of the workload is shown in Figure 2 and the slab allocation is shown in Figure 3 on the next page. Ideally, memcached should be able to reassign a slab as soon as the command is issued. However, it takes nearly 60 million requests to reach steady state miss ratio.

The slow reassignment of slabs from one class to another is due to long waiting times for clearing active items from a slab. To clear an item from a slab, we must unlink it from the LRU queue and then remove the item from the class's list of available item slots. Currently, this is a two-step process that involves unnecessary waiting. Our solution combines the unlinking and removal from the freelist into one step in the slab reassignment algorithm.

This paper makes the following contributions. First, we present a detailed description of the slab reassignment process in memcached; second, we identify why slab reassignment can take so long; and third, we provide an improvement to the base reassignment algorithm. We show that our fast implementation reduces the time needed to reassign a slab by over 99%, resulting in the ability to reach workloads' steady state miss ratio by 53% to 75% faster. By arriving at the steady state miss ratio faster, we reduce the overall average miss ratio by 3.42% to 11.5%.

2 Background

memcached is a popular open-source, in-memory key-value store. memcached is often used as a cache for backend systems in a datacenter environment. Requests to memcached are in the form of GET key and SET key value. When a requested key is found in memcached (cache hit), the corresponding data is returned to the client. In the case that a requested key is not found (cache miss) in memcached, a SET request adds the (key,value) pair to memcached after the data has been retrieved from the corresponding backend system. Backend systems can range from large databases, advertising platforms, and web applications.

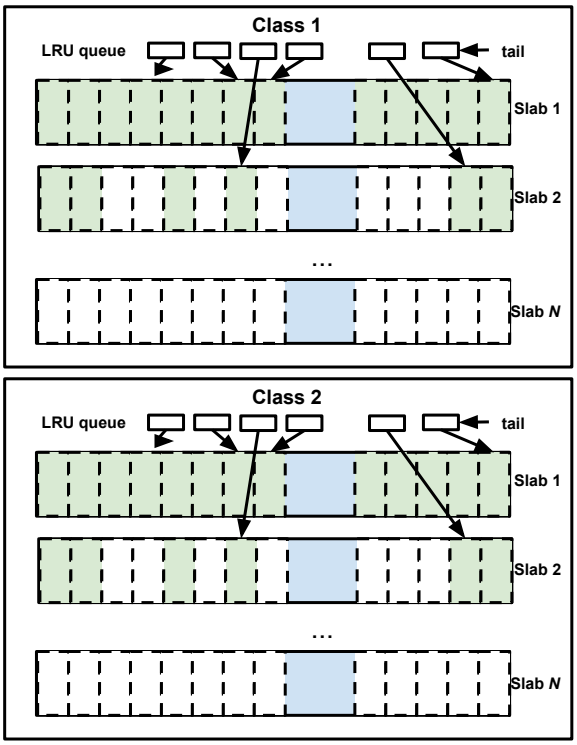

Figure 1. memcached internal memory layout. Each class of items has their own set of slabs and their own LRU queue for eviction. We illustrate two classes but the system may be configured for more. Dark shaded boxes represent active item allocations. Keys and values are stored together in memcached.

2.1 Memory Layout

memcached stores data in logical units of memory called slabs. Each slab by default is 1MB and belongs to a class. Slabs store fixed size chunks according to the slab's class. For example class 1 stores chunks of 104 bytes, class 2 stores chunks of 136 bytes, class 3 store chunks of 176 bytes, and so on. Chunks contain the item's key, value, and header information (last access time, next and previous item pointers, slab class, number of bytes, etc.). Figure 1 illustrates this layout of memory in memcached.

memcached maintains the items in a class as a least recently used (LRU) list. When a class is full, it will evict the LRU item from the class in order to allocate new data. Ideally, if a class requires more slabs than currently assigned it can acquire more through dynamic slab reallocation. Dynamically reallocating slabs attempts to maximize the overall

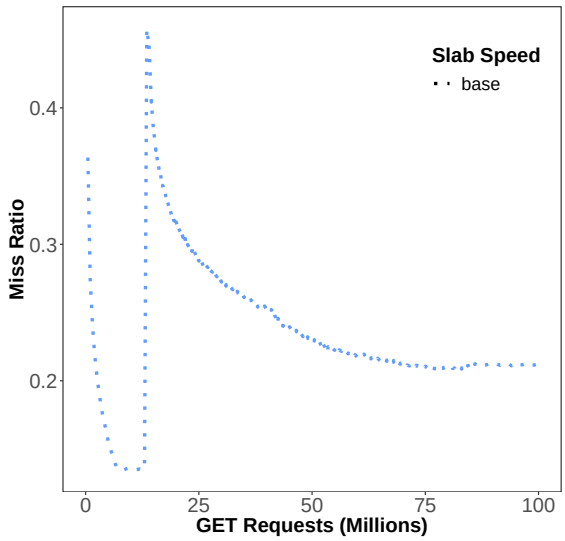

Figure 2. Miss Ratio for Two-Phase Workload under the base reassignment algorithm using LAMA allocation policy. After transitioning to phase 2 of the workload, it takes nearly 60 million requests to reach steady state miss ratio.

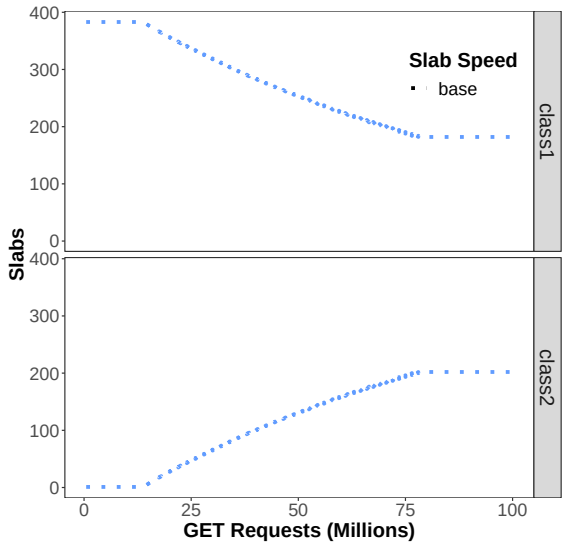

Figure 3. Slab Allocation for Two-Phase Workload under the base reassignment algorithm using LAMA allocation policy. Each slab can take millions of requests to reassign.

hit ratio of memcached. Many works introduce allocation strategies, from measuring class request rates to constructing full miss ratio curves for each class [5, 7, 10]. With those studies already in place, we will not focus on the problem of how many slabs to assign to each class. But rather how does the slab reassignment mechanism affect the miss ratio and latency of memcached.

2.2 memcached Slab Reassignment Mechanism

The process of reassigning a slab from one class to another in memcached is complex since it involves acquiring various locks and waiting for chunks to be freed. In fact, Berger et al. [2] point out that the memcached slab reassignment process typically takes 5 seconds when reassigning hundreds of slabs, which is not uncommon in today's datacenter environments.

Before introducing the slab reassignment algorithm, we must first explain how items are locked in memcached. There are two ways of locking an item in memcached: *mutex* and *reference count* locking. In mutex locking, an item is locked by holding the mutex corresponding to the hash value of the item's key. The second is by incrementing the item's reference count. To increment the reference count, the mutex lock must be held. After incrementing the reference count of the item, the mutex lock can be released. The reference count value refers to the number of current threads that reference the item.

On a GET request, an item must be locked in order to increment the reference count. On a SET request, the slab that the item is allocated from will be locked, with a global slab lock, and the reference count of the newly allocated item is initialized to 1, and the single reference is to the LRU thread.

Therefore, memcached defines a *busy item* as an item with a reference count greater than 0. This means that at least 1 thread is currently referencing that item. We call an item a *locked item* if its mutex is currently held by a thread. In the context of slab reassignment, we are given a slab to reassign from a source class to a destination. As outlined in Algorithm 1 (base implementation), we first must remove all items from the slab. We can remove items with a reference count of 0 safely since there are no outstanding references to that item. Then the following cases can occur for a given item:

1. Item is locked (by mutex), then we must wait on that item to free up. Its data is actively being updated (pointers, meta-information, etc.).
2. Item has a reference count greater than 2, then we must also wait on that item. At least 2 other threads (in addition to the reassignment thread) are currently referencing that item. This is the case when the LRU maintainer and an item crawler thread is also holding a reference to that item.
3. Item has a reference count of 2, then there is only one other thread that can modify some portion of the item. If the item only has the LINKED flag (linked in the LRU queue), then the only other thread that is referencing the current item is the LRU thread.

Cases 1 and 2 can slow down slab reassignment since the reassignment thread must wait for those items to become free before they can be unlinked, deallocated, and removed from a slab's freelist. In cases 1 and 2, the *wasBusy* counter is incremented and the sleep statement in line 29 will be

Algorithm 1 Base Implementation of Slab Reassignment

```
1:  for item ∈ srcSlab do
2:    wasBusy ← 0
3:    if item == unallocated then
4:      cutFromFreelist(srcSlab, item)
5:    end if
6:    if item == linked then
7:      haveLock ← aquirelock(item)
8:      if haveLock == false then
9:        // Case 1: mutex lock
10:       wasBusy ← wasBusy + 1
11:     else
12:       item.refs ← item.refs + 1
13:       if item.refs > 2 then
14:         // Case 2: addt'l threads have ref. to item
15:         wasBusy ← wasBusy + 1
16:       else
17:         // Case 3: only reassignment thread and LRU
            queue hold reference
18:         saved ← trySaveItem(item)
19:         if saved == false then
20:           unlinkQ(item)
21:           wasBusy ← wasBusy + 1
22:         end if
23:       end if
24:       item.refs ← item.refs − 1
25:       unlock(item)
26:     end if
27:   end if
28:   if wasBusy > 0 then
29:     usleep(1000)
30:   end if
31: end for
32: assignSlab(srcSlab, dstClass) // give slab to requesting
    class
```

executed. When the thread resumes execution, the execution begins at line 2 for the same item, checking all three cases again.

The vast majority of items fall into case 3. The current memcached design is to take a two-step approach in case 3. First, the item is unlinked from the queue. In the `unlinkQ` method, the item is removed from the LRU queue causing the reference count to drop to 1. Next, the item is deallocated from the slab causing the reference count to become 0 and placed on the slab's freelist. Therefore, after a successful call to `unlinkQ`, the item's reference count will be 0. Since prior to unlinking the item from the LRU queue the item *was busy* (reference count > 0), the base implementation will then proceed to line 29, where the thread waits for 1000 μs. When the thread resumes execution, the execution begins at line 2 and the item may now be removed from the slab's freelist.

While the reassignment algorithm sleeps, a recently deallocated item slot can be allocated by other memcached threads that are serving requests. In fact, we found that this case occurs over 3 million times while running the Two-Phase test workload under memcached's base (default) slab reassignment mechanism. Each time a recently freed item slot in a slab to be reassigned becomes allocated, the reassignment algorithm must now unlink the newly allocated item before it can reassign that slab. Every time the reassignment algorithm unlinks a recently allocated item, the algorithm will end up sleeping (line 29) for an extra round. This adds extra sleeping time and contributes significantly to the slow slab reassignment speed in the default memcached implementation.

3 Faster Slab Reassignment

Algorithm 2 Fast Slab Reassignment (replaces line 21 in Algorithm 1)

```
1: if item.refs == 0 then
2:   cutFromFreelist(srcSlab, item)
3: end if
```

An initial solution may involve removing the sleep statement in line 29. Or sleeping for a shorter time. First, we explain why sleeping is necessary: sleeping when a locked or busy item is encountered allows the reassignment thread to suspend execution while an item is being utilized by another thread. This reduces the CPU usage of the reassignment thread, allowing other processes/threads to be scheduled at that time.

A second solution may involve reducing the sleep time. The default in memcached is 1000μs. Sleeping for a shorter period of time does reduce the time it takes to reallocate slabs (and therefore reaches steady state miss ratio faster). But there is a tradeoff between the time it takes to reallocate slabs and the CPU usage of the reassignment thread. In the base implementation, a high sleep interval (1000 microseconds) will have lower CPU usage since it will be sleeping more in comparison to a low sleep interval. Since our goal is to move slabs as quickly as possible, we would like to avoid sleeping when it is not necessary. This will increase the CPU usage of the reassignment thread, but ultimately increase the amount of slabs moved over time.

Our implementation is based on the idea that *an item can be cut from the freelist immediately after it has been unlinked from the LRU queue*. However, this only is correct if there are no outstanding references to the item after unlinking it from the LRU queue (see case 3). If there are more than two references to the item, then our algorithm will follow the behavior of the base implementation (case 2). We must wait for the additional threads to release their references before freeing the item from the slab. Our results show that case 2

does not occur in any of our workloads, but it still needs to be accounted for since it is a possibility.

For example, a possible case 2 scenario is when the LRU crawler thread searches the queue for expired items. When the crawler thread encounters item A, it will increment item A's reference count. A context switch could occur at this moment, allowing the reassignment thread to run. If the reassignment thread encounters item A and acquires its lock, the reassignment thread will increment item A's reference count from 2 (LRU queue and crawler thread) to 3 (LRU queue, crawler thread, and reassignment thread). Therefore, in case 2, our algorithm will perform no worse than the current base implementation. Since case 2 is a very rare case, it does not affect the performance of our fast implementation.

Our change to the base algorithm is illustrated in Algorithm 2. First, we check that there are no outstanding references to this item from other threads (we are past the item lock, so no other thread can increment the item's reference count if it has not done so already). This also implies that the item was unlinked (the call to `unlinkQ(item)` was successful). The item will remain unlinked since linking would require the item lock, which is currently held by the reassignment thread. Now we can safely cut the item from the slab freelist.

By immediately removing items from the slab's freelist in line 2 of Algorithm 2, we are reduce the waiting due to case 3 (see Algorithm 1). This significantly reduces the amount of time that it takes to clear a slab's items and assign the slab to the destination class.

4 Results

We now present the impact of improved slab reassignment algorithm. We implement the fast algorithm in memcached 1.5.12 with the same miss ratio curve-based partitioning among classes as implemented in the LAMA system [10, 11]. We use the LAMA slab allocation policy over the default memcached Slab Automove policy because LAMA achieves near optimal miss ratios in its target slab allocations. LAMA uses miss ratio curves to relate the cache size to the average miss ratio for an LRU cache. Given a memory budget, we can search through a set of possible slab assignments in order to minimize the overall miss ratio of the system. Our fast implementation is independent of the slab allocation policy.

First, we show why our design is preferred over simply modifying the sleep intervals in the base implementation using a two phase workload. Second, we show the impact of the fast implementation on the slab allocation and miss ratio for the two phase workload. Third, we create realistic, multi-tenant, time-varying workload based rates measured at Facebook [1]. Finally, we run an another multi-tenant workload used in [4] that contains a diverse set of application access patterns and working set sizes. Our experiments run on a 48-core 2.2GHz Intel(R) Xeon(R) CPU E5-2650 v5 and 256GB of DDR4 DRAM at 2400 MHz. All experiments are compiled and run on Red Hat 6.3.1 with Linux kernel 4.11.12 and GNU C compiler (gcc 7.3.0, -O0 optimization).

Table 1. Comparison of sleep intervals and the resulting CPU usage and slab reassignment speed.

algorithm	sleep interval	cpu usage %	slabs/s
base	1	13.8	36.93
base	10	10.6	35.5
base	100	8.0	26.55
base	1000	3.1	4.12
fast	1000	97.0	252.31

4.1 Two-Phase Workload

The workload is divided into two phases: phase 1 is 100% items with size 200 bytes, phase 2 is split 33% items of size and 66% items of size 250 bytes. Phase 1 lasts 13 million requests and phase 2 lasts 87 million requests. Each size distribution belongs to its own class within memcached and has Zipfian access pattern $\alpha = 1.0$, consistent with measured access patterns in typical workloads [3]. Items sized 200 bytes are in class 1 and items sized 250 bytes are in class 2. Each item size distribution has 5 million unique items. All memory (384 MB) is initially allocated to class 1 (200 bytes). Once the new item size distribution is introduced, memcached begins assigning slabs from class 1 to class 2.

4.1.1 CPU Usage vs. Slab Reassignment Speed

Table 1 shows a comparison of CPU usage of the reassignment thread under different sleep intervals. We also report the average number of slabs moved per second of user space execution time of the reassignment thread.

The tradeoff that occurs is CPU usage vs. the number of slabs moved per second. Our fast design trades higher CPU usage during slab reassignment for a 6.8x increase over the base implementation in the number of slabs moved per second. This allows the reassignment thread to complete much faster, reducing the total execution time of the thread. In the base implementation, we see that a high sleep interval ($1000\mu s$) reduces CPU usage since the thread will spend more time idle compared to a low sleep interval ($1\ \mu s$). As a result, the base implementation with a high sleep interval will not move as many slabs per second when compared to a low sleep interval. When comparing the base implementation with a low sleep interval to our fast version, the base with low sleep interval has less CPU usage during slab reassignment. However, it takes longer to complete the slab reassignment because it spends time constantly sleeping on items that fall into case 3 category.

The tradeoff for increased CPU usage over the reassignment period is justified in memcached servers. Since the

Table 2. Comparison of the number of items encountered for each case in the slab reassignment algorithm under the Two-Phase workload.

	case 1	case 2	case 3
base	15	0	2,658,153
fast	20	0	0

Table 3. Comparison of time (GET requests) required to reassign a single slab from class 1 to class 2 under the Two-Phase workload.

	mean reassign time	min	max
base	314,608	181,903	469,813
fast	47.95	10	100

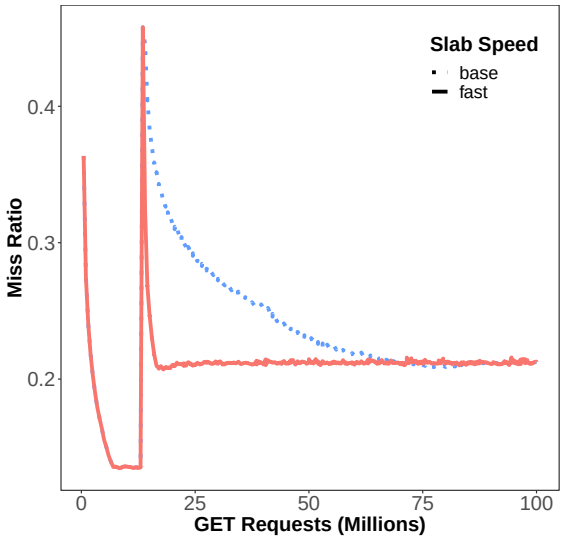

Figure 4. Miss ratio of base system vs. fast implementation for Two-Phase workload. Fast reaches steady state miss ratio over 75% faster than the base system.

performance of memcached is typically bound by memory capacity, CPU is an underutilized resource [8]. By increasing CPU usage in order to improve the performance of memcached, we follow the observations made by the data infrastructure team at Facebook: "Memcached shares a RAM-heavy server configuration with other services that have more demanding CPU requirements, so in practice memcached is never CPU-bound in our datacenters. Increasing the CPU to improve hit rate would be a good trade off." [8].

4.1.2 Miss Ratio and Slab Allocation

The improvement in the miss ratio is illustrated in Figure 4. Figure 4 shows the miss ratio over the number of GET

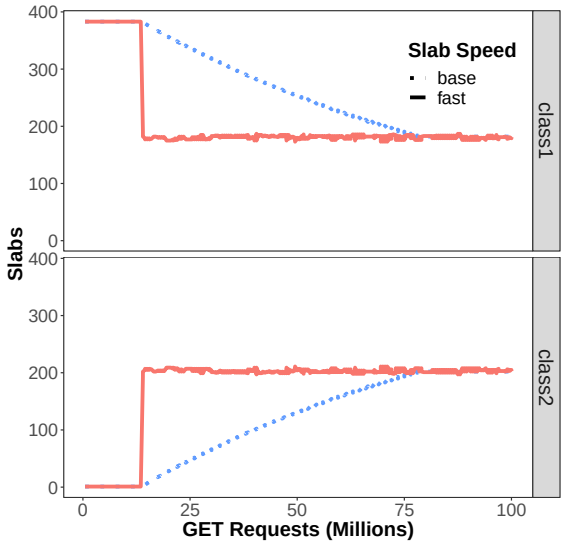

Figure 5. Slab allocation of base system vs. fast implementation for Two-Phase workload. Fast reaches steady state slab allocation over 78% faster than the base system.

requests comparing the base slab reassignment with our fast implementation. The fast algorithm reaches the steady state miss ratio, 21%, at 16.5 million requests. In the base implementation, the steady state is not reached until 67 million requests. The fast algorithm thus results in an over 75% improvement in time to reach steady state.

Figure 5 shows the slab allocation over the number of requests for each system. In each system when the LAMA partitioning algorithm determines the number of slabs for each class, it sends commands to the slab reassignment mechanism (either the base implementation or our fast implementation). Each system has the same target number of slabs for each class. The fast algorithm can assign the target number of slabs to each class at 16.5 million requests, while it takes the base system until 76.5 million requests to the target. This indicates an over 78% improvement in the time needed to allocate slabs to classes by the fast algorithm.

Table 2 shows the number of items encountered that fall into cases 1, 2 and 3 under the fast slab reassignment algorithm. Table 3 also shows the average time it takes to assign a slab from class 1 to 2. The mean time it takes to reassign a slab is reduced from 314,608 to just 47.95 GET requests.

We can explain the decrease in reassignment time from the massive reduction in busy items and no longer having recently freed items become reallocated. We encounter 2,658,153 busy items and 3,095,891 unlinked then re-allocated slots in the slab to reassign. Each of those costs about 1000 microseconds to sleep. In the Two-Phase workload, we reallocate 380 slabs totally and measure a throughput of 32K requests/second. Therefore: (2,658,153 busy items + 3,095,891 unlinked then reallocated) * 1000 μs / (1,000,000 μs/sec) / 380

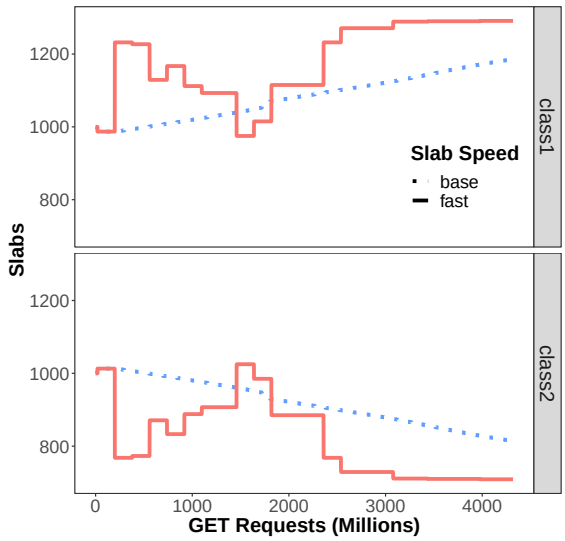

Figure 6. Slab allocation of base system vs. fast implementation for Time-Varying workload. Fast reaches steady state for each hour, while the base implementation fails to reach a target assignment to due excessive waiting.

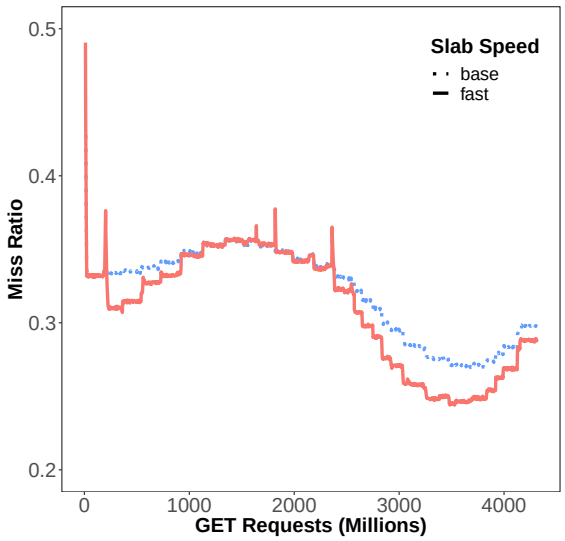

Figure 7. Miss ratio of base system vs. fast implementation for Time-Varying workload. Fast achieves 3.42% improvement in miss ratio over the base algorithm.

Table 4. Comparison of the number of items encountered for each case in the slab reassignment algorithm under the Time-Varying workload.

	case 1	case 2	case 3
base	4,819	0	146,856,662
fast	169	0	0

Table 5. Comparison of time (GET requests) required to reassign a single slab from class 1 to class 2 under the Two-Phase workload.

	mean reassign time	min	max
base	19,454,304	17	26,333,587
fast	77.62	4	220

slabs * (32,000 * 0.774 GETS) = 375042.53 expected number of GET requests to reassign a single slab, not far from our measured mean of 314608 GET requests. In the fast implementation we only need to wait on 20 items due to case 1 and we avoid the case of item slots being reallocated since the slot is cut immediately from the freelist.

As a result of reaching the steady state miss ratio faster, we also lower the total number of misses recorded on the system. The overall mean miss ratio for the base system is 23.3%. Under fast slab reassignment, the overall mean miss ratio is reduced to 20.9%. This results in an 11.5% improvement in the mean miss ratio. A lower number of misses reduces the number of queries to the backend systems and results in significant end-to-end latency improvement in a typical datacenter environment.

4.2 Time-Varying Workload

In some environments, multiple applications share a single memcached instance. This is referred to as multi-tenancy [4, 8]. It allows applications to share the same cache resources as their demand for cache space changes over time. Our time-varying workload simulates two applications (ETC1 and ETC2) over an entire 24 hour period following the request rate distribution at Facebook described in Atikoglu et al. [1]. Both applications have the same Zipfian access pattern ($\alpha = 0.25$) and draw from 7 million unique items. Each application has 2.16 billion requests in total over the 24 hour period. We initially allocated 1 GB to each application and reassign slabs every hour after an initial training period (18 million requests), each hour is 180 million requests totally.

Figure 6 shows the slab allocation for each application over the number of requests. We can see that it takes a significant amount of time for the base mechanism to reassign slabs to the target amount. In fact, the base implementation does not even reach the proper target within the hour. This is due to the unnecessary waiting for many busy items. Table 4 shows how many many items we wait on and Table 5 shows the mean time it takes to reassign a slab. We observe a 99.99% decrease in the mean time required to reassign a slab.

The impact on the miss ratio is illustrated in Figure 7. The fast implementation results in a 31% average miss ratio over the entire workload, while the base implementation

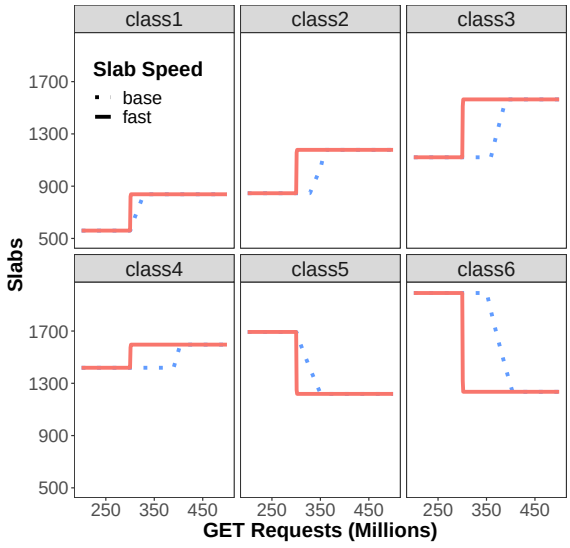

Figure 8. Slab allocation of base system vs. fast implementation for multi-tenant workload. Slab reassignment completes in 40,713 GET requests for the fast implementation. The base requires over 100 million requests to complete.

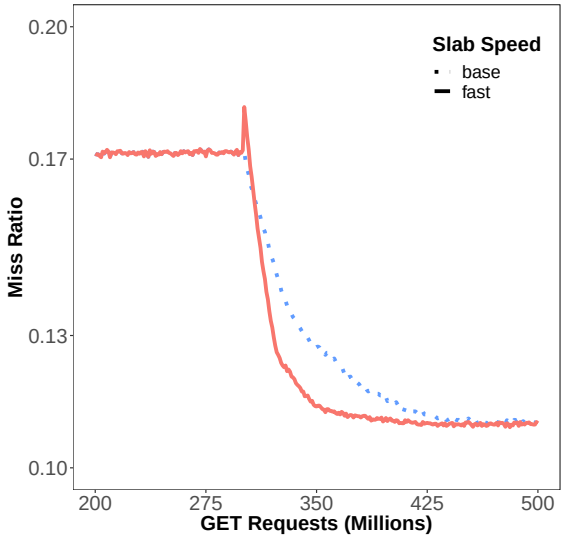

Figure 9. Miss ratio of base system vs. fast implementation for multi-tenant workload. Fast reaches steady state miss ratio at 360 million requests compared to 430 million requests in the base system.

Table 6. Workload parameters used. Number of objects is expressed in millions.

Workload	Unique objects	WSS	Zipf - α
etc	7	6.97	0.25
etc.small	3.5	3.47	0.25
psa	7	6.12	0.85
psa.small	3.5	3.38	0.85
ycsb	10	5.28	1.0
ycsb.small	5	3.63	1.0

Table 7. Comparison of the number of items encountered for each case in the slab reassignment algorithm under the multi-tenant workload.

	case 1	case 2	case 3
base	143	0	3,217,421
fast	71	0	0

Table 8. Comparison of time (GET requests) required to reassign a single slab under the multi-tenant workload.

	mean reassign time	min	max
base	83,808	59,581	125,601
fast	33.1	0	117

results in a 32.1% average miss ratio. This results in a 3.42% improvement. The small spikes in the miss ratio are a result of clearing many items from the source application. One way to address this is to adjust the percentage of slabs that are reassigned at every interval. A typical limit is 10-20% depending on the application's cache behavior [4]. This also requires increasing the number of reassignment intervals in order to reach the best allocation for the workload.

4.3 Multi-Tenant Workload

The multi-tenant workload contains 6 different applications with the following reuse patterns based on Zipfian-like distribution with varying α values to model web access patterns [3] and varying number of objects (ETC, PSA, YCSB) from [5, 9, 10]. Table 6 gives the parameters for each workload. Each workload has the same total number of requests, 100 million. We define the working set size (WSS) as the number of objects in the cache such that any additional objects do not increase the cache hit ratio.

In our experiment, each application is initially given an equal number of objects allocated, 3 million. We emulate different applications by using a different slab class for each application. That is, *etc*'s data resides in slab class 1 (objects sized 192 bytes), *etc.small*'s data resides in slab class 2 (objects sized 296 bytes), and so on. We reassign the memory among the applications halfway through the trace at 300 million requests.

Figure 8 shows the slab allocation for each application over the number of requests for the multi-tenant workload.

Figure 8 illustrates the serial behavior of the slab reassignment thread: only a single slab may be moved at a time from the source class to the destination class. In order to move a slab from class C to D, we must wait until the prior reassignment completes. At the repartition time (300 million requests), slabs are added to class 1 and removed from class 5. Once all of the slabs from class 5 are reassigned to class 1 then we may proceed to reassign slabs from class 6 to class 2. And again, from class 6 to class 3. And finally, from class 6 to class 4. Under the base implementation the entire process takes over 100 million requests to complete. In our test environment (32K requests per second),the process can take over 50 minutes for the system to complete reassignment.

Table 7 shows how many many items we wait on and Table 8 shows the mean time it takes to reassign a slab. We observe a 99.9% decrease in the mean time required to reassign a slab, similar to the other workloads.

The impact on the average miss ratio of the system is shown in Figure 9. We focus on the miss ratio during the slab reassignment time, starting at 300 million requests. The fast implementation requires 60 million requests to reach steady state miss ratio, or about 31 minutes in our test environment. In comparison, the base system requires 130 million requests (about 67 minutes) to reach the same steady state miss ratio. Our fast implementation in this workload achieves a 53% speedup in the time required to reach the steady state miss ratio. As a result, the average miss ratio during the slab reassignment period (300 million to 460 million requests) decreases from 12.4% in the base system to 11.8% in the base system, a 4.8% improvement.

5 Related Work

One of the earliest works to identify the issues caused by static slab assignments was Periodic Slab Allocation (PSA) by Carra et al. [5]. PSA showed that by reassigning slabs between classes dynamically, memcached can avoid slabs being underutilized when other classes would benefit from additional slabs. Since then, works such as Cliffhanger [7] and Dynacache [6] have utilized victim caches in order to decide how many slabs a class should receive based on their current hit rate in a class's respective victim cache. The higher the class's respective victim cache hit rate, the steeper the gradient of the miss ratio curve. Therefore, slabs should be assigned to classes with high victim cache hit rates.

Our work on memcached is orthogonal to the prior works that reallocate slabs between applications and/or classes. Works such as LAMA [10], mPart [4], and Robinhood [2] focus on *how many* slabs should each class (or application) receive in order to maximize the hit ratio or impact on the 99th percentile latency. Both LAMA and mPart use miss ratio curves in order to achieve near optimal memory assignments. Our work uses LAMA's slab partitioning algorithm to dynamically calculate how many slabs to assign to each class. mPart studied the multi-tenant environment showing that miss ratio curves can outperform other sharing models in a typical multi-tenant environment. The Robinhood system aims to reduce the 99th percentile tail latency in memory cache systems by reassigning slabs from applications with extra memory to applications that could benefit from additional cache space.

6 Conclusion

Apart from the initial introduction of the slab reassignment mechanism to memcached, there has not been any related work on improving the process of slab reassignment. In the face of dynamic workloads and multi-tenant environments, slab reallocation policies are increasingly relying on reassigning large amounts of memory among classes. This motivates the need for an efficient slab reassignment mechanism as we have shown the current implementation is inefficient. This paper shows two significant performance improvements due to the faster slab reassignment mechanism. First, improvement in the miss ratio is seen as a consequence of reaching the steady state faster. Second, our slab reassignment mechanism reduced CPU utilization compared to other possible implementations.

7 Availability

Our modified version of memcached, which includes the fast implementation can be found here:

`https://github.com/mpart-mtu/memcached`

We plan to submit this as a patch to the current memcached source code, enabling faster slab reassignment for the user base.

Acknowledgments

We would like to thank Cheng Pan of Peking University for his correspondence and sharing with us the initial version of memcached with miss ratio curve-based partitioning. This research is supported in part by the National Science Foundation under Grant No. CSR1618384, the National Science Foundation of China under Grant No. 61232008, 61472008, 61672053 and U1611461, and Shenzhen Key Research Project under Grant No. JCYJ20170412150946024. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64.

[2] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association, Carlsbad, CA, 195–212. https://www.usenix.org/conference/osdi18/presentation/berger

[3] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings (INFOCOM '99)*, Vol. 1. 126–134 vol.1.

[4] 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 (ISMM 2018)*. ACM, New York, NY, USA, 84–95. https://doi.org/10.1145/3210563.3210571

[5] D. Carra and P. Michiardi. 2014. Memory partitioning in Memcached: An experimental performance analysis. In *2014 IEEE International Conference on Communications (ICC '14)*. 1154–1159.

[6] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching. In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'15)*. USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=2827719.2827738

[7] 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)*. USENIX Association, Santa Clara, CA, 379–392.

[8] 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.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154.

[10] 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)*. USENIX Association, Santa Clara, CA, 57–69.

[11] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2017. Optimizing Locality-Aware Memory Management of Key-Value Caches. *IEEE Trans. Comput.* 66, 5 (May 2017), 862–875. https://doi.org/10.1109/TC.2016.2618920

[12] memcached. 2018. memcached. Retrieved May 10, 2018 from https://memcached.org

[13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[14] Manju Rajashekhar and Yao Yue. 2012. Twemcache: Twitter memcached.