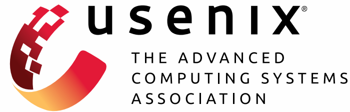

Memshare: a Dynamic Multi-tenant Key-value Cache

Asaf Cidon, *Stanford University;* Daniel Rushton, *University of Utah;* Stephen M. Rumble, *Google Inc.;* Ryan Stutsman, *University of Utah*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the
2017 USENIX Annual Technical Conference
is sponsored by USENIX.

Memshare: a Dynamic Multi-tenant Key-value Cache

Asaf Cidon*, Daniel Rushton†, Stephen M. Rumble‡, Ryan Stutsman†
*Stanford University, †University of Utah, ‡Google Inc.

Abstract

Web application performance heavily relies on the hit rate of DRAM key-value caches. Current DRAM caches statically partition memory across applications that share the cache. This results in under utilization and limits cache hit rates. We present Memshare, a DRAM key-value cache that dynamically manages memory across applications. Memshare provides a resource sharing model that guarantees reserved memory to different applications while dynamically pooling and sharing the remaining memory to optimize overall hit rate.

Key-value caches are typically memory capacity bound, which leaves cache server CPU and memory bandwidth idle. Memshare leverages these resources with a log-structured design that allows it to provide better hit rates than conventional caches by dynamically re-partitioning memory among applications. We implemented Memshare and ran it on a week-long trace from a commercial memcached provider. Memshare increases the combined hit rate of the applications in the trace from 84.7% to 90.8%, and it reduces the total number of misses by 39.7% without significantly affecting cache throughput or latency. Even for single-tenant applications, Memshare increases the average hit rate of the state-of-the-art key-value cache by an additional 2.7%.

1 Introduction

DRAM key-value caches are essential for reducing application latency and absorbing massive database request loads in web applications. For example, Facebook has dozens of applications that access hundreds of terabytes of data stored in memcached [24] in-memory caches [41]. Smaller companies use outsourced multi-tenant in-memory caches to cost-effectively boost SQL database performance.

High access rates and slow backend database performance mean reducing cache miss rates directly translates to end-to-end application performance. For example, one Facebook memcached pool achieves a 98.2% hit rate [9]. With an average cache latency of 100 µs and MySQL access times of 10 ms, increasing the hit rate by 1% reduces latency by 36% (from 278 µs to 179 µs) and reduces database read load by 2.3×.

Today, operators statically divide memory across applications. For example, Facebook, which manages its own data centers and cache clusters [9, 39], has an engineer that is tasked to manually partition machines into separate cache pools for isolation. Similarly, Memcachier [4, 18], a cache-as-a-service for hundreds of tenants, requires customers to purchase a fixed amount of memory.

Static partitioning is inefficient, especially under changing application loads; some applications habitually under utilize their memory while others are short of resources. Worse, it is difficult for cache operators to decide how much memory should be allocated to each application. This manual partitioning requires constant tuning over time. Ideally, a web cache should automatically learn and assign the optimal memory partitions for each application based on their changing working sets; if an application needs a short term boost in cache capacity, it should be able to borrow memory from one that needs it less, without any human intervention.

To this end, we designed *Memshare*, a multi-tenant DRAM cache that improves cache hit rates by automatically sharing pooled and idle memory resources while providing performance isolation guarantees. To facilitate dynamic partitioning of memory among applications, Memshare stores each application's items in a segmented in-memory log. Memshare uses an *arbiter* to dynamically decide which applications require more memory and which applications are over-provisioned, and it uses a *cleaner* to evict items based on their rank and to compact memory to eliminate fragmentation.

This paper makes two main contributions:

1. Memshare is the first multi-tenant web memory cache that optimally shares memory across applications to maximize hit rates, while providing isolation guarantees. Memshare does this with novel dynamic and automatic profiling and adaptive memory reallocation that boost overall hit rate.
2. Memshare uniquely enforces isolation through a log-structured design with application-aware cleaning that enables fungibility of memory among applications that have items of different sizes. Due to its memory-efficient design, Memshare achieves significantly higher hit rates than the state-of-the-art memory cache, both in multi-tenant environments and in single-tenant environments.

In Memshare, each application specifies a minimum amount of *reserved* memory; the remaining *pooled* memory is used flexibly to maximize hit rate. Inspired by

Cliffhanger [19], Memshare optimizes hit rates by estimating hit rate gradients; it extends this approach to track a gradient for each application, and it awards memory to the applications that can benefit the most from it. This enables cache providers to increase hit rates with fewer memory resources while insulating individual applications from slowdowns due to sharing. Even when all memory is reserved for specific applications, Memshare can increase overall system efficiency without affecting performance isolation by allowing idle memory to be reused between applications. Memshare also lets each application specify its own eviction policy (e.g., LRU, LFU, Segmented LRU) as a *ranking function* [11]. For example, to implement LRU, items are ranked based on the timestamp of their last access; to implement LFU, items are ranked based on their access frequency.

Existing memory caches cannot support these properties; they typically use a slab allocator [3, 18, 19], where items of different sizes are assigned to slab classes and eviction is done independently on a class-by-class basis. This limits their ability to reassign memory between different applications and between items of different sizes.

Memshare replaces slab allocation with a new log-structured allocator that makes memory fungible between items of different sizes and applications. The drawback of the log-structured allocator is that it continuously repacks memory contents to reassign memory, which increases CPU and memory bandwidth use. However, increasing hit rates in exchange for higher CPU and memory bandwidth use is attractive, since key-value caches are typically memory capacity bound and not CPU bound. In a week-long trace from Memcachier, cache inserts induce less than 0.0001% memory bandwidth utilization and similarly negligible CPU overhead. CPU and memory bandwidth should be viewed as under utilized resources that can be used to increase the cache efficiency, which motivates the log-structured approach for memory caches.

Nathan Bronson from the data infrastructure team at Facebook echoes this observation: "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 data centers. Increasing CPU to improve the hit rate would be a good trade off." [16]. Even under high CPU load, Memshare's cleaner can dynamically shed load by giving up eviction policy accuracy, but, in practice, it strongly enforces global eviction policies like LRU with minimal CPU load.

We implement Memshare and analyze its performance by running a week-long trace from Memcachier, a multi-tenant memcached service [18]. We show that Memshare adds 6.1% to the overall cache hit rate compared to memcached. We demonstrate that Memshare's added overheads do not affect client-observed performance for real workloads, since CPU and memory bandwidth are significantly under utilized. Our experiments show that Memshare achieves its superior hit rates and consumes less than 10 MB/s of memory bandwidth, even under aggressive settings. This represents only about 0.01% of the memory bandwidth of a single CPU socket. We demonstrate that in the case of a single-tenant application running in the cache, Memshare increases the number of hits by an extra 2.37% compared to Cliffhanger [19], the state-of-the-art single-tenant cache. To the best of our knowledge, Memshare achieves significantly higher average hit rates than any other memory cache both for multi-tenant and single-tenant workloads.

2 Motivation

DRAM key-value caches are an essential part of web application infrastructure. Facebook, Twitter, Dropbox, and Box maintain clusters of thousands of dedicated servers that run web caches like memcached [24] that serve a wide variety of real-time and batch applications. Smaller companies use caching-as-a-service providers such as ElastiCache [1], Redis Labs [5] and Memcachier [4]. These multi-tenant cache providers may split a single server's memory among dozens or hundreds of applications.

Today, cache providers partition memory statically across multiple applications. For example, Facebook, which manages its own cache clusters, partitions applications among a handful of pools [9,39]. Each pool is a cluster of memcached servers that cache items with similar QoS needs. Choosing which applications belong in each pool is done manually. Caching-as-a-service providers like Memcachier [4, 18] let customers purchase a certain amount of memory. Each application is statically allocated memory on several servers, and these servers maintain a separate eviction queue for each application.

2.1 Partitioned vs Pooled

We compare two different resource sharing schemes with memcached using simulation[1]: the static partitioning used by Memcachier, and a greedy pooled memory policy, both using memcached's slab allocator with LRU. In the static partitioning, we run applications just as they run in our commercial Memcachier trace; each is given isolated access to the same amount of memory it had in the trace. In the pooled policy, applications share all memory, and their items share eviction queues. An incoming item from any application evicts items from the tail of the shared per-class eviction queues (§2.2), which are oblivious to which application the items belong to. We use a motivating example of three different applications (3, 5 and 7) selected from a week-long trace of memcached traffic running on Memcachier. These applications suffer from bursts of requests, so they clearly demonstrate the trade offs between the partitioned and pooled memory policies.

[1] Source available at `http://github.com/utah-scs/lsm-sim/`

App	Hit Rate Partitioned	Pooled
3	97.6%	96.6%
5	98.8%	99.1%
7	30.1%	39.2%
Combined	87.8%	88.8%

Table 1: Average hit rate of Memcachier's partitioned and pooled policy over a week.

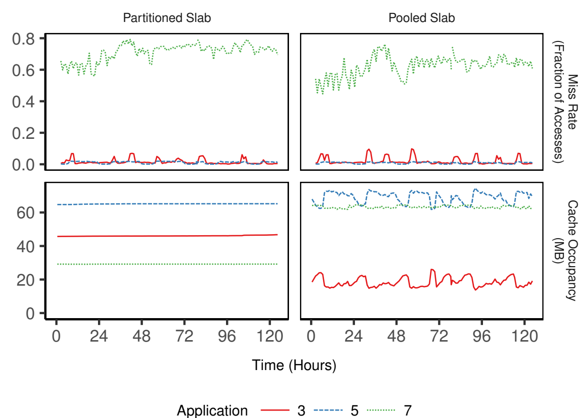

Figure 1: Miss rate and cache occupancy of Memcachier's partitioned and pooled policies over time.

Table 1 shows the average hit rates over a week of the three applications in both configurations. Figure 1 depicts the average miss rate and cache occupancy over the week. The pooled policy gives a superior overall hit rate, but application 3's hit rate drops 1%. This would result in 42% higher database load and increased latencies for that application. The figure also shows that the pooled scheme significantly changes the allocation between the applications; application 3 loses about half its memory, while application 7 doubles its share.

2.2 Slab Allocation Limits Multi-tenancy

Ideally, a multi-tenant eviction policy should combine the best of partitioned and pooled resource sharing. It should provide performance isolation; it should also allow applications to claim unused memory resources when appropriate, so that an application that has a burst of requests can temporarily acquire resources. This raises two requirements for the policy. First, it must be able to dynamically arbiter which applications can best benefit from additional memory and which applications will suffer the least when losing memory. Second, it needs to be able to dynamically reallocate memory across applications.

Unfortunately, allocators like memcached's slab allocator greatly limit the ability to move memory between applications, since items of different sizes are partitioned in their own slabs. The following example illustrates the problem. Imagine moving 4 KB of memory from application 1 to application 3. In the trace, the median item size for application 1 and 3 are 56 B and 576 B, respectively. In Memcachier, each 1 MB slab of memory is assigned a *size class*; the slab is divided into fixed sized chunks according to its class. Classes are in units of 64×2^i up to 1 MB (i.e. 64 B, 128 B, ..., 1 MB). Each item is stored in the smallest class that can contain the item. Therefore, items of 56 B are stored in a 1 MB slab of 64 B chunks, and 576 B are stored in a 1 MB slab of 1 KB chunks.

There are two problems with moving memory across applications in a slab allocator. First, even if only a small amount needs to be moved (4 KB), memory can only be moved in 1 MB units. So, application 1 would have to evict 1 MB full of small items, some of which may be hot; memcached tracks LRU rank via an explicit list, which doesn't relate to how items are physically grouped within slabs. Second, the newly reallocated 1 MB could only be used for a single item size. So, application 3 could only use it for items of size 256-512 B or 512-1024 B. If it needed memory for items of both sizes, it would need application 1 to evict a second slab. Ideally, the cache would only evict the bottom ranked items from application 1, based on application 1's eviction policy, which have a total size of 4 KB. This problem occurs even when assigning memory between different object sizes *within the same application*.

This motivates a new design for a multi-tenant cache memory allocator that can dynamically move variable amounts of memory among applications (and among different object sizes of the same application) while preserving applications' eviction policy and priorities.

3 Design

Memshare is a lookaside cache server that supports the memcached API. Unlike previous key-value caches, Memshare stores items of varying sizes and applications physically together in memory, and uses a cleaner running in the background to remove dead items. When the cache is full, it decides which items to evict based on the items' eviction priorities and how effectively each application uses its share of the cache.

Memshare is split into two key components. First, Memshare's arbiter must determine how much memory should be assigned to each application (its *targetMem*). Second, Memshare's cleaner implements these assignments by prioritizing eviction from applications that are using too much cache space.

3.1 The Cleaner and Arbiter

Memshare's in-memory cleaner fluidly reallocates memory among applications. The cleaner finds and evicts the least useful items for any application from anywhere in memory, and it coalesces the resulting free space for newly written items. This coalescing also provides fast allocation and high memory utilization.

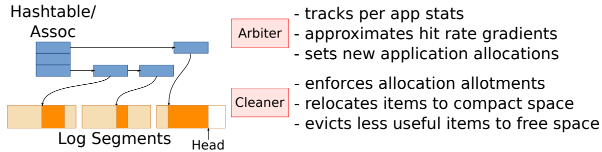

Figure 2: The Memshare design. Incoming items are allocated from the head of a segmented in-memory log. The hash table maps keys to their location in the log. The arbiter monitors operations and sets allocation policy. The cleaner evicts items according to the arbiter's policy and compacts free space.

All items in Memshare are stored in a segmented in-memory log (Figure 2). New items are allocated contiguously from the same active *head* segment, which starts empty and fills front-to-back. Once an item has been appended to the log, the hash table entry for its key is pointed to its new location in the log. Unlike slab allocator systems like memcached, Memshare's segments store items of all sizes from all applications; they are all freely intermixed. By default, segments are 1 MB; when the head segment is full, an empty "free" segment is chosen as head. This accommodates the largest items accepted by memcached and limits internal fragmentation.

When the system is running low on free segments ($<1\%$ of total DRAM), it begins to run the cleaner in the background, in parallel with handling normal requests. The cleaner frees space in two steps. First, it evicts items that belong to an application that is using too much cache memory. Second, it compacts free space together into whole free segments by moving items in memory. Keeping a small pool of free segments allows the system to tolerate bursts of writes without blocking on cleaning.

Memshare relies on its arbiter to choose which items the cleaner should prefer for eviction. To this end we define the *need* of each application as its need for memory:

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

Where *actualMem* is the actual number of bytes currently storing items belonging to the application, and *targetMem* is the number of bytes that the application is supposed to be allocated. In the case of partitioned resource allocation *targetMem* is constant. If the need of an application is above 1, it means it needs to be allocated more memory. Similarly, if the need is below 1, it is consuming more memory than it should. The arbiter ranks applications by their *need* for memory; the cleaner prefers to clean from segments that contain more data from applications that have the lowest need. Items in a segment being cleaned are considered one-by-one; some are saved and others are evicted.

Cleaning works in "passes". Each pass takes n distinct segments and outputs at most $n-1$ new segments, freeing up at least one empty segment. This is done by writing back the most essential items into the $n-1$ output segments. The writing is contiguous so free space, caused by obsolete items that were overwritten, is also eliminated. n is a system parameter that is discussed in Section 6. Note that multiple passes can run in parallel.

Algorithm 1 Memory relocation

```
1:  function CLEANMEMORY(segments, n)
2:      relocated = 0
3:      residual = (n - 1) · segmentSize
4:      while segments not empty do
5:          app = arbiter.maxNeed()
6:          item = maxRank(segments, app)
7:          segments.remove(item)
8:          if item.size ≤ residual - relocated then
9:              relocate(item)
10:             relocated = relocated + item.size
11:             app.actualMem = app.actualMem + item.size
12:         else
13:             break
14:         end if
15:     end while
16: end function
```

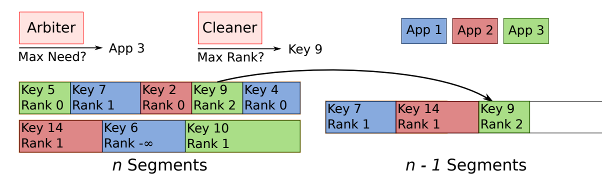

Figure 3: Memshare relocates items from n segments to $n-1$ segments. The arbiter first chooses the application with the highest need, and the cleaner relocates the item with the highest rank among the items of that application.

In each pass, Memshare selects a fraction of the segments for cleaning randomly and a fraction based on which segments have the most data from applications with the lowest need. Random selection helps to avoid pathologies. For example, if segments were only chosen based on application need, some applications might be able to remain over provisioned indefinitely so long as there are worse offenders. Based on experience with the Memcachier traces, choosing half of the segments randomly avoided pathologies while tightly enforcing arbiter policies.

Once a set of segments is selected for cleaning, the cleaner sorts the items in the segments by rank to determine which items should be preserved. Figure 3 and Algorithm 1 show how this is done in a single cleaning pass. *segments* is a list of all the items from the segments being cleaned in the pass. In order to choose which item to relocate next, the cleaner first determines the application that has the highest need (*maxNeed*). Among the items in the segments that belong to that application, the cleaner then chooses the item with the highest rank (*maxRank*, e.g. LRU-rank). It relocates the item by copying it and updating its entry in the hash table. After the item is relocated, the need for that application is recalculated. The

process is repeated until the $n-1$ segments are full or all items are relocated. The remaining items are evicted by dropping them from the hash table, and the need for the applications' whose items were evicted is adjusted.

Memshare can use any generic ranking function on items to prioritize them for eviction; in fact, it can be determined by the application. Memshare supports any ranking function *rank*(t, f), that is based on the timestamp t of the last access of each item and f the number of times it has been accessed. For example, to implement LRU, the ranking function is *rank*$(t) = t$; that is, it is the item's last access timestamp. LFU is just the number of accesses to an item: *rank*$(f) = f$. Segmented LRU can be implemented as a combination of the timestamp of the last access of the item and the number of times it has been accessed. Throughout the paper, when evaluating the hit rate of different caches, we use LRU as the default eviction policy.

A key idea behind Memshare is that memory partitioning is enforced by the decision of which items to clean, while any application can write at any time to the cache. Consider the case where Memshare is configured for a static partitioning among applications, and one application continuously writes new items to the cache while other applications do not. Allocations are static, so *targetMem* will remain constant. As the first application inserts new items, its *actualMem* will increase until its need drops below the need of the other applications. When the memory fills and cleaning starts, the arbiter will choose to clean data from the application that has the lowest need and will begin to evict its data. If there are other active applications competing for memory, this application's *actualMem* will drop, and its need will increase.

3.2 Balancing Eviction Accuracy and Cleaning

The cost of running Memshare is determined by a trade off between the accuracy of the eviction policy, determined by the parameter n and the rate of updates to the cache. The higher the rate of updates, the faster the cleaner must free up memory to keep up. Section 6.1 evaluates this cost and finds for the trace the cleaning cost is less than 0.01% utilization for a single CPU socket. Even so, the cleaner can be made faster and cheaper by decreasing n; decreasing n reduces the amount of the data the cleaner will rewrite to reclaim a segment worth of free space. This also results in the eviction of items that are ranked higher by their respective applications, so the accuracy of the eviction policy decreases. In our design, n can be dynamically adjusted based on the rate of updates to the cache. Web cache workloads typically have a low update rate (less than 3%) [39].

The last of the $n-1$ segments produced by the cleaning pass may be less than full when there are many dead items in the original n segments. The new $n-1$ segments are sorted based on need and rank, so one optimization is to evict the items in last segment if its utilization is low ($< 50\%$) since it contains low rank and need items.

4 Memshare's Sharing Model

Memshare allows the operator to fix a reserved amount of memory for each application. The rest of the cache's memory is pooled and dynamically assigned to the applications whose hit rates would benefit the most from it. Each application's reserved memory we call *reservedMem*; the remaining memory on the server is *pooledMem*, shared among the different applications. At each point in time, Memshare has a target amount of memory it is trying to allocate to each application, *targetMem*. In the case of statically partitioned memory, *pooledMem* is zero, and *targetMem* is always equal to *reservedMem* for each application.

targetMem defines an application's fair share. The resource allocation policy needs to ensure that each application's *targetMem* does not drop below its *reservedMem*, and that the remaining *pooledMem* is distributed among each application in a way that maximizes some performance goal such as the maximum overall hit rate.

To maximize the overall hit rate among the applications, each application's hit rate curve can be estimated; this curve indicates the hit rate the application would achieve for a given amount of memory. Given applications' hit rate curves, memory can be reallocated to applications whose hit rate would benefit the most. However, estimating hit rate curves for each application in a web cache can be expensive and inaccurate [18, 19].

Instead, Memshare estimates local hit rate curve gradients with *shadow queues*. A shadow queue is an extension of the cache that only stores item keys and not item values. Each application has its own shadow queue. Items are evicted from the cache into the shadow queue. For example, imagine an application has 10,000 items stored in the cache, and it has a shadow queue that stores the keys of 1,000 more items. If a request misses the cache and hits in the application's shadow queue, it means that if the application had been allocated space for another 1,000 items, the request would have been a hit. The shadow queue hit rate gives a local approximation of an application's hit rate curve gradient [19]. The application with the highest rate of hits in its shadow queue would provide the highest number of hits if its memory was incrementally increased.

Algorithm 2 shows how *targetMem* is set. Each application is initially given a portion of *pooledMem*. For each cache request that is a miss, the application's shadow queue is checked. If key is present in the shadow queue, that application is assigned a *credit* representing the right to use to a small chunk (e.g., 64 KB) of the pooled memory. Each assigned credit is taken from another application at random (*pickRandom* above). The cleaner uses

Algorithm 2 Pooled memory: set target memory

```
1: function SETTARGET(request, application)
2:     if request ∉ cache AND
           request ∈ application.shadowQueue then
3:         candidateApps = {}
4:         for app ∈ appList do
5:             if app.pooledMem ≥ credit then
6:                 candidateApps = candidateApps ∪ {app}
7:             end if
8:         end for
9:         pick = pickRandom(candidateApps)
10:        application.pooledMem =
            application.pooledMem + credit
11:        pick.pooledMem = pick.pooledMem - credit
12:    end if
13:    for app ∈ appList do
14:        app.targetMem =
            app.reservedMem + app.pooledMem
15:    end for
16: end function
```

App	Hit Rate Partitioned	Hit Rate Memshare 50%
3	97.6%	99.4%
5	98.8%	98.8%
7	30.1%	34.5%
Combined	87.8%	89.2%

Table 2: Average hit rate of Memshare with 50% reserved memory compared to the partitioned policy.

Reserved Memory	Total Hit Rate
0%	89.4%
25%	89.4%
50%	89.2%
75%	89.0%
100%	88.8%

Table 3: Comparison of Memshare's total hit rate with different amounts of reserved memory for applications 3, 5, and 7.

targetMem to choose which applications to evict items from. *appList* is a list of all applications in the cache and *cache* is a list of all items in the cache.

Table 2 compares Memshare with the statically partitioned Memcachier scheme. For Memshare, each application is configured to use 50% of the memory that was allocated to it in the original trace as reserved memory with the rest as pooled memory. Memshare delivers equal or better hit rates both application-by-application and overall. Even with 50% of memory reserved, Memshare also achieves a higher overall hit rate (89.2%) than the greedy pooled memory scheme (88.8%, Table 1).

Table 3 and Figure 4 further explore the trade off between overall hit rate and per-application hit rates as we vary the percentage of memory that is held reserved. The figure shows that with more memory held reserved, reallocation between applications dampens. In addition, the figure shows Memshare's cleaner enforces the reserved memory allocation for each application: applications never fall below their reservations. The figure also shows how Memshare's memory allocation reacts to the rate of shadow queue hits. In the far left graphs, when the cache has no reserved memory, Memshare allocates pooled memory to the applications that have a high shadow queue hit rate. As Memshare allocates more memory to the bursty application, its shadow queue hit rate tempers. In the far right graphs, when the cache is fully reserved, Memshare cannot allocate any additional memory to the bursty applications; therefore, the shadow queue hit rate remains high.

App	Credit Size	Hit Rate	Credit Size	Hit Rate
3	64 KB	99.4%	64 KB	99.5%
5	128 KB	98.5%	64 KB	98.6%
7	192 KB	33.4%	64 KB	32.3%

Table 4: Assigning different credit sizes to each application allows cache operators to prioritize among applications.

Finally, Table 2 and 3 break down how much of Memshare's hit rate improvements come from its allocator and how much come from its sharing model. With 100% reserved memory, Memshare is equivalent to static partitioning, but it achieves a 88.8% hit rate compared to 87.8% for memcached: a 1% gain strictly due to the allocator. Going from 100% reserved memory to 0% shows a 0.6% gain. This shows that about 38% of Memshare's gains are from memory sharing. Note that effective sharing also requires log-structured allocation.

4.1 Allocation Priority

Cache providers may want to guarantee that when certain applications have bursts of requests, they would get a higher priority than other applications. In order to accommodate this requirement, Memshare enables cache operators to assign different shadow queue credit sizes to different applications. This guarantees that if a certain application has a higher credit size than other applications, when it requires a larger amount of memory due to a burst of activity, it will be able to expand its memory footprint faster than other applications.

Table 4 demonstrates how assigning different weights to different applications affects their overall hit rate. In this example, application 7 achieves a higher relative hit rate, since it receives larger credits in the case of a shadow queue hit.

4.2 Increasing Efficiency for Reserved Memory

Pooled memory works for environments like Facebook's where multiple cooperative applications use a shared caching layer, and the operator wants to provide the best overall performance while providing minimum guarantees to applications. However, in some environments, applications are inherently selfish and would like to maximize their reserved memory, but the cache operator still

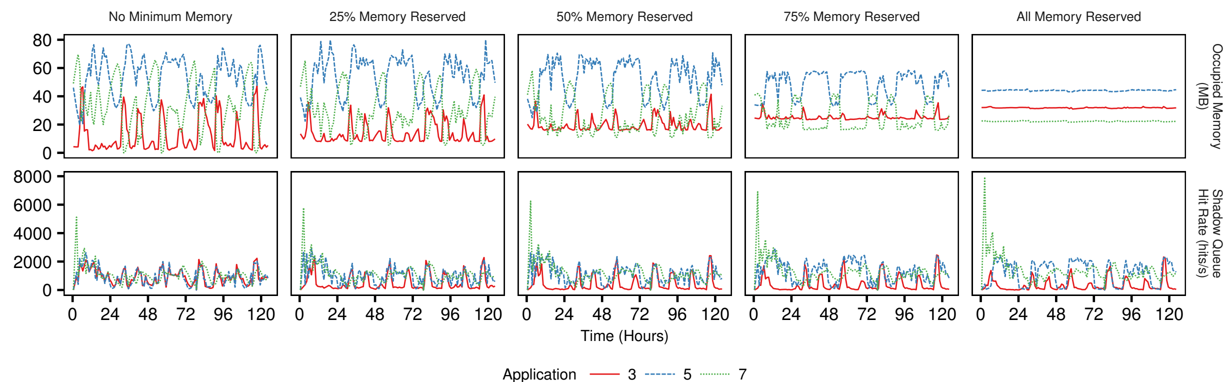

Figure 4: Comparison of Memshare's memory consumption and the rate of shadow queue hits with different amounts of memory reserved for applications 3, 5 and 7. Memshare assigns more pooled memory to applications with a high shadow queue hit rate.

Algorithm 3 Idle tax: set target memory

1: **function** SETTARGET(app, taxRate, idleTime)
2: idleMem = 0
3: **for** item $\in$ app **do**
4: **if** item.timestamp $<$ currentTime - idleTime **then**
5: idleMem $+=$ item.size
6: **end if**
7: **end for**
8: activeFraction = $1 - \frac{\text{idleMem}}{\text{app.actualMem}}$
9: $\tau = \frac{1 - \text{activeFraction} \cdot \text{taxRate}}{1 - \text{taxRate}}$
10: app.targetMem = $\frac{\text{app.reservedMem}}{\tau}$
11: **end function**

has an incentive to optimize for effective memory utilization. If applications are "sitting on" their underutilized reserved memory, their resources can be reassigned without negatively impacting their performance.

To help with this, Memshare also supports an idle memory tax that allows memory that has not been accessed for a period to be reassigned. Memshare implements the tax with one small change in how the arbiter sets each application's *targetMem*. Algorithm 3 shows how the arbiter computes *targetMem* for each application when the tax is enabled; *taxRate* $\in [0,1]$ determines what fraction of an application's memory can be reassigned if it is idle. If *taxRate* is 1, all of the application's idle memory can be reassigned (and its *targetMem* will be 0). If *taxRate* is 0, the idle tax cache policy is identical to partitioned allocation. Idle memory is any memory that has not been accessed more recently than *idleTime* ago. The arbiter tracks what fraction of each application's memory is idle, and it sets *targetMem* based on the tax rate and the idle fraction for the application.

In this algorithm, *targetMem* cannot be greater than *reservedMem*. If multiple applications have no idle memory and are competing for additional memory, it will be allocated to them in proportion to their *reservedMem*. For example, if two applications with a *targetMem* of 5 MB and 10 MB respectively are contending for 10 MB, the 10 MB will be split in a 1:2 ratio (3.3 MB and 6.7 MB).

App	Hit Rate Memcachier Partitioned	Idle Tax
3	97.6%	99.4%
5	98.8%	98.6%
7	30.1%	31.3%
Combined	87.8%	88.8%

Table 5: Average hit rate of Memshare's idle tax policy.

Table 5 depicts the hit rate Memshare's idle tax algorithm using a tax rate of 50% and a 5 hour idle time. In the three application example, the overall hit rate is increased, because the idle tax cache policy favors items that have been accessed recently. Application 5's hit rate decreases slightly because some of its idle items were accessed after more than 5 hours.

5 Implementation

Memshare consists of three major modules written in C++ on top of memcached 1.4.24: the log, the arbiter and the cleaner. Memshare reuses most of memcached's units without change including its hash table, basic transport, dispatch, and request processing.

5.1 The Log

The log replaces memcached's slab allocator. It provides a basic `alloc` and `free` interface. On allocation, it returns a pointer to the requested number of bytes from the current "head" segment. If the request is too big to fit in the head segment, the log selects an empty segment as the new head and allocates from it.

Allocation of space for new items and the change of a head segment are protected by a spin lock. Contention is not a concern since both operations are inexpensive:

allocation increments an offset in the head segment and changing a head segment requires popping a new segment from a free list. If there were no free segments, threads would block waiting for the cleaner to add new segments to the free list. In practice, the free list is never empty (we describe the reason below).

5.2 The Arbiter

The arbiter tracks two key attributes for each application: the amount of space it occupies and its shadow LRU queue of recently evicted items. The SET request handler forwards each successful SET to the arbiter so the per-application bytes-in-use count can be increased. On evictions during cleaning passes, the arbiter decreases the per-application bytes-in-use count and inserts the evicted items' into the application's shadow queue. In practice, the shadow queue only stores the 64-bit hash of each key and the length of each item that it contains, which makes it small and efficient. Hash collisions are almost non-existent and do no harm; they simply result in slight over-counting of shadow queue hits.

5.3 The Cleaner

The cleaner always tries to keep some free memory available. By default, when less than 1% of memory is free the cleaner begins cleaning. It stops when at least 1% is free again. If the cleaner falls behind the rate at which service threads perform inserts, then it starts new threads and cleans in parallel. The cleaner can clean more aggressively, by reducing the number of segments for cleaning (n) or freeing up more segments in each cleaning pass. This trades eviction policy accuracy for reduced CPU load and memory bandwidth.

Cleaning passes must synchronize with each other and with normal request processing. A spin lock protects the list of full segments and the list of empty segments. They are both manipulated briefly at the start and end of each cleaning pass to choose segments to clean and to acquire or release free segments. In addition, the cleaner uses Memcached's fine-grained bucket locks to synchronize hash table access. The cleaner accesses the hash table to determine item liveness, to evict items, and to update item locations when they are relocated.

The arbiter's per-app bytes-in-use counts and shadow queues are protected by a spin lock, since they must be changed in response to evictions. Each cleaner pass aggregates the total number of bytes evicted from each application and it installs the change with a single lock acquisition to avoid the overhead of acquiring and releasing locks for every evicted item. The shadow queue is more challenging, since every evicted key needs to be installed in some application's shadow queue. Normally, any GET that results in a miss should check the application's shadow queue. So, blocking operations for the whole cleaning pass or even just for the whole duration needed to populate it with evicted keys would be prohibitive. Instead, the shadow queue is protected with a spin lock while it is being filled, but GET misses use a `tryLock` operation. If the `tryLock` fails, the shadow queue access is ignored.

The last point of synchronization is between GET operations and the cleaner. The cleaner never modifies the items that it moves. Therefore, GET operations do not acquire the lock to the segment list and continue to access records during the cleaning pass. This could result in a GET operation finding a reference in the hash table to a place in a segment that is cleaned before it is actually accessed. Memshare uses an epoch mechanism to make this safe. Each request/response cycle is tagged at its start with an epoch copied from a global epoch number. After a cleaning pass has removed all of the references from the hash table, it tags the segments with the global epoch number and then increments it. A segment is only reused when all requests in the system are from epochs later than the one it is tagged with.

5.4 Modularity

Memshare maintains separation between the cleaner and the arbiter. To do this, after a cleaning pass chooses segments, it notifies the arbiter which items are being cleaned. The arbiter ranks them and then calls back to the cleaner for each item that it wants to keep. If the relocation is successful, the arbiter updates the item's location in the hash table. Once the empty segments have been filled with relocated items, the arbiter removes the remaining entries from the hash table and updates per-application statistics and shadow queues. In this way, the cleaner is oblivious to applications, ranking, eviction policy, and the hash table. Its only task is efficient and parallel item relocation.

6 Evaluation

To understand Memshare's benefits, we ran two sets of tests. First, we ran a week-long multi-tenant Memcachier trace with Memshare to measure hit rate gains and end-to-end client-observed speedup. Second, we also benchmarked the implementation with the YCSB [20] workload to understand the overheads introduced by Memshare's online profiling and log cleaning.

Our experiments run on 4-core 3.4 GHz Intel Xeon E3-1230 v5 (with 8 total hardware threads) and 32 GB of DDR4 DRAM at 2133 MHz. All experiments are compiled and run using the stock kernel, compiler, and libraries on Debian 8.4 AMD64.

6.1 Performance

We tested the performance of Memshare using all the major applications from the Memcachier trace with the pooled memory and idle tax policies. Figure 5 presents

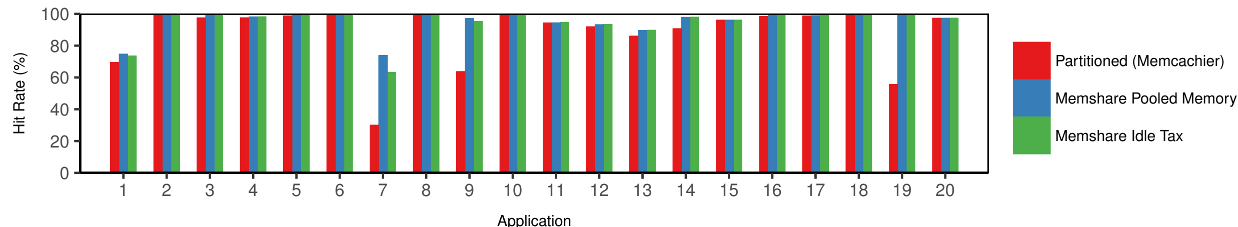

Figure 5: Memshare's pooled memory and idle tax algorithms' hit rates for top Memcachier applications compared to memcached.

Policy	Combined Hit Rate	Miss Reduction
memcached	84.66%	0.00%
Cliffhanger	87.73%	20.00%
Memshare Tax	89.92%	34.28%
Memshare Pooled	90.75%	39.69%

Table 6: Combined hit rate of Memshare's idle tax (50% tax) and pooled memory policy (75% reserved) compared with Cliffhanger, which is the state-of-the-art slab-based cache and Memcached. The miss reduction column compares the miss rate of the different policies to memcached.

the hit rate results, and Table 6 presents the summary. The pooled cache policy provides a higher overall combined hit rate increase, since it tries to maximize for overall hit rates. On average, Memshare reduces the number of misses by 39.7%. With an average cache latency of 100 µs and database latency of 10 ms, this would result in an average application-observed speedup of 1.59× (average access time of 1,016 µs versus 1,619 µs). In some cases, such as applications 7, 9, and 19, Memshare provides more than a 20% hit rate improvement.

Our evaluation uses 1 MB segments and 100 candidate segments for cleaning, the same as memcached's default slab and maximum item size. The number of candidate segments was chosen experimentally (see Table 7); it provides the best hit rate and results in less than 0.01% memory bandwidth use. The pooled policy used 75% of each application's original Memcachier memory as reserved with the rest pooled. Shadow queues were configured to represent 10 MB of items. Idle tax policy was set to a 50% tax rate with all memory reserved for each application. For the pooled policy, we experimented with different credit sizes. When credit sizes are too small, pooled memory isn't moved fast enough to maximize hit rates; when they are too high, memory allocation can oscillate, causing excessive evictions. We found a credit size of 64 KB provides a good balance.

Table 7 presents the combined hit rate and cleaner memory bandwidth consumption of Memshare's pooled memory policy when varying n, the number of segments that participate in each cleaning pass. The table shows that for the Memcachier traces, there is a diminishing increase in hit rate beyond n=40. While memory bandwidth use increases as the number of candidate segments is higher,

Segments (n)	Hit Rate	Memory Bandwidth (MB/s)
1	89.20%	0.04
10	90.47%	2.14
20	90.58%	2.86
40	90.74%	4.61
60	90.74%	6.17
80	90.75%	7.65
100	90.75%	9.17

Table 7: Combined hit rate and memory bandwidth use of top 20 applications in Memcachier trace using Memshare with the pooled memory policy with 75% reserved memory and varying the number of segments in each cleaning pass.

Policy	Average Single Tenant Hit Rate
memcached	88.3%
Cliffhanger	93.1%
Memshare 100% Reserved	95.5%

Table 8: Average hit rate of the top 20 applications in the trace run as a single tenant with Memshare with 100% reserved memory compared with Cliffhanger and memcached.

near peak hit rates can be achieved for this trace while consuming less than 0.01% of the memory bandwidth of a single modern CPU socket. Even at 100 candidate segments, the memory bandwidth of Memshare is less than 10 MB/s for the top 20 applications in the trace.

6.1.1 Single Tenant Hit Rate

In addition to providing multi-tenant guarantees, Memshare's log structured design significantly improves hit rates on average for individual applications on a cache which uses a slab allocator. Table 8 compares the average hit rates between Memshare and two systems that utilize slab allocators: memcached and Cliffhanger [19]. Within a single tenant application, Cliffhanger optimizes the amount of memory allocated to each slab to optimize for its overall hit rate. However, Memshare's log structured design provides superior hit rates compared to Cliffhanger, because it allows memory to be allocated fluidly for items of different sizes. In contrast, each time Cliffhanger moves memory from one slab class to another, it must evict an entire 1 MB of items, including items that may be hot. On average, Memshare with 100% reserved memory increases the hit rate by 7.13% compared to memcached and by 2.37% compared to Cliffhanger.

	GET Hit	Latency GET Miss	SET
memcached	21.44 µs	21.8 µs	29.48 µs
Memshare	22.04 µs	23.0 µs	23.62 µs

Table 9: Memshare and memcached access latency under an artificial workload that causes high CPU load. Shadow queue lookups increases latency in the case of GET cache misses.

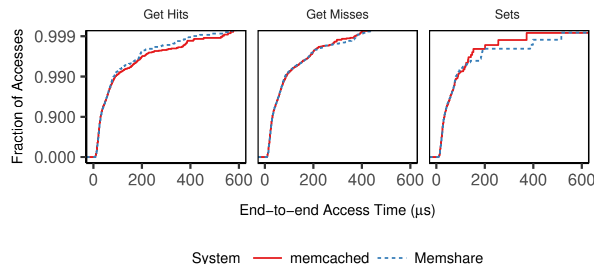

Figure 6: Tail latency distribution for Memshare/memcached.

6.2 Microbenchmarks

The Memcachier traces result in a low CPU utilization, so we also ran microbenchmarks using the YCSB framework [20] to stress CPU and memory bandwidth utilization. All of the microbenchmarks use 25 B items with 23 B keys over 100 million operations. Measurements always include the full cost of cleaning.

6.2.1 Latency

Table 9 shows the average response latency of Memshare with a full cache and a running cleaner compared to memcached. The clients and cache server are running on one machine, so the measurements represent a worst case. Access times are dominated by the network software stack and round trip delay [42]. Memshare's GET hit latency is 2.8% slower than memcached, and GET misses are 5.5% slower due to the check for the key in the shadow queue. Shadow queues are naïve LRU queues, so this could be mitigated. The additional latency on a miss is hidden, since the application must access the database which takes tens to hundreds of milliseconds.

Large-scale applications that exploit caches have high request fan-out and are known to be sensitive to tail latency [21, 39]. Figure 6 compares the tail latency of Memshare with memcached. Despite Memshare's slower average latency, it improves 99th and 99.9^{th} percentile get hit response times from 91 to 84 µs and 533 to 406 µs, respectively. Get miss tail latency is nearly identical between the systems; despite the extra cost of maintaining the shadow queue, 99th and 99.9^{th} percentile Memshare response times are 4 µs faster and 9 µs slower than memcached, respectively. 99th and 99.9^{th} percentile set times show the impact of the cleaner with Memshare showing times 8 µs faster and 143 µs slower, respectively; most allocation is faster, but occasionally allocation is delayed by

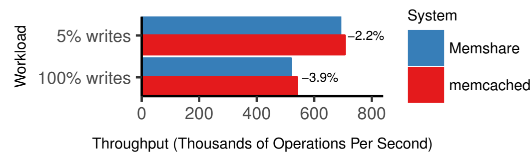

Figure 7: Average throughput of Memshare compared to memcached under a YCSB workload with 5% writes and 95% reads and under a worst case workload with 100% writes.

cleaning. Tail latency is often a concern for systems that perform garbage collection, like flash solid-state drives; Memshare is more robust against outliers since its critical sections are small and it never holds shared resources like serial channels to flash packages. Cleaning is fully parallel and effectively non-blocking.

6.2.2 CPU and Throughput

Table 7 compares Memshare throughput with memcached under a YCSB workload with 95%/5% reads/writes and one with 100% writes. Memshare is 2.2% slower for the first workload and 3.9% slower with all writes.

Most of the throughput loss is due to Memshare's cleaner. To breakdown the loss, we measured the CPU time spent on different tasks. In the 5% write workload, 5.1% of the process's CPU time is spent on cleaning, and 1.1% is spent testing shadow queues on GET misses. Note that the 100% write workload is unrealistic (such a workload does not need a cache). With a 100% write workload 12.8% of the process's CPU time is spent on cleaning.

The small decrease in Memshare's throughput is well justified. In-memory caches are typically capacity-bound not throughput-bound, and operate under low loads [16, 18]. The Memcachier trace loads are two orders of magnitude less than the microbenchmark throughput. Cache contents are often compressed; the gains from Memshare's efficient allocation are orthogonal, and the benefits can be combined since cleaning little CPU.

6.2.3 Memory Overhead and Utilization

Memshare has a small memory overhead. By default, shadow queues represent 10 MB of items; the overhead of the queues depends on the size of the items. Assuming small items on average (128 B), one queue stores 81,920 keys. Queues only keep 8 B key hashes, so key length isn't a factor. The default overhead is 81,920 · 8 B = 640 KB per application. The other structures used by Memshare have a negligible memory overhead.

Memshare's cleaner wastes some space by keeping some segments pre-cleaned; however, this space only represents about 1% of the total cache in our implementation. Even with some idle memory, Memshare is still better than memcached's slab allocator, since it eliminates the internal fragmentation that slab allocators suffer from. For example, in the trace, memcached's fragmentation restricts memory utilization to 70%-90%.

7 Related Work

Memshare builds on work in memory allocation and caching. Cliffhanger [19] estimated local hit rate curve gradients to rebalance slabs of items of different sizes. Memshare estimates local gradients to divide memory among applications. Memshare's log-structured allocator achieves significantly higher hit rates than Cliffhanger and flexibly moves memory across applications.

ESX Server [53] introduced idle memory taxation and min-funding revocation [52] in the context of a virtual machine hypervisor. Ranking functions to determine cache priorities were introduced by Beckmann et al [11] in the context of CPU caches. Memshare is the first application of both of these ideas to DRAM caches.

RAMCloud [45] and MICA [36] apply techniques from log-structured file systems [15, 37, 44, 47, 48] to DRAM-based storage. Log-structured caches have appeared in other contexts, such as a CDN photo cache [51] and mobile device caches [6]. Unlike these systems, Memshare addresses multi-tenancy. Also, MICA relies on FIFO eviction which suffers from inferior hit rates. Memshare enables application developers to apply any eviction policy using their own ranking functions.

MemC3 [23] and work from Intel [35] improve memcached multicore throughput by removing concurrency bottlenecks. These systems significantly improve performance, but they do not improve hit rates. In the case of Facebook and Memcachier, memcached is memory capacity bound, not CPU or throughput bound [16, 18].

Some caches minimize write amplification (WA) on flash [22, 51]. As presented, Memshare would suffer high WA on flash: low-need segments must be cleaned first, resulting in near-random 1 MB overwrites, which are detrimental for flash. Newer non-volatile media [2] may work better for Memshare.

Resource Allocation and Sharing. FairRide [43] gives a general framework for cache memory allocation and fairness when applications share data. Data sharing among competing applications is not common in key-value web caches. For both Facebook and Memcachier, applications each have their own unique key space; they never access common keys. For applications that do not share data, FairRide implements a memory partitioning policy in a distributed setup. Memshare, unlike FairRide, can efficiently use non-reserved and allocated idle memory to optimize the hit rate of applications and provide them with a memory boost in case of a burst of requests.

Mimir [46] and Dynacache [18] approximate stack distance curves of web caches for provisioning and slab class provisioning, respectively. They do not provide a mechanism for allocating memory among different applications sharing the same cache.

Efforts on cloud resource allocation, such as Moirai [50], Pisces [49], DRF [25] and Choosy [26] focus on performance isolation in terms of requests per second (throughput), not hit rate which is key in determining speedup in data center memory caches [16]. Similarly, there have been several projects analyzing cache fairness and sharing in the context of multicore processors [27, 30, 31]. In the context of multicore, fairness is viewed as a function of total system performance. Unlike CPU caches, DRAM-based web caches are typically separate from the compute and storage layer, so the end-to-end performance impact is unknown to the cache.

Ginseng [8] and RaaS [7, 13] are frameworks for memory pricing and auctioning for outsourced clouds; they only focus on pricing memory for applications that have dedicated memory cache servers running on VMs. In contrast, Memshare enables applications to share the same memory cache server, without the need for VM isolation. This is the preferred deployment model for most web application providers (e.g., Facebook, Dropbox, Box).

Eviction Policies. Many eviction schemes can be used in conjunction with Memshare. For example, Greedy-Dual-Size-Frequency [17] and AdaptSize [14] take into account request sizes to replace LRU as a cache eviction algorithm for web proxy caches. Greedy-Dual-Wheel [34] takes into account how long it takes to process a request in the database to improve cache eviction. EVA [10, 12] computes the opportunity cost per byte for each item stored in a cache. ARC [38], LRU-K [40], 2Q [29], LIRS [28] and LRFU [32, 33], offer a combination of LRU and LFU.

8 Conclusion

This paper demonstrates there is a large opportunity to improve key-value hit rates in multi-tenant environments, by utilizing dynamic and fungible memory allocation across applications. Memshare serves as a foundation for promising future research of memory sharing in distributed cache environments. For example, the techniques introduced in this paper are implemented within a single server running multiple applications. Similar techniques can be applied across servers, to determine the appropriate dynamic resources allocated to each application. Finally, key-value caches are being extended to other storage mediums beyond memory, such as flash and non-volatile memory. Running multiple applications on a heterogeneous caching environment creates novel future research challenges.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amazon Elasticache. `aws.amazon.com/elasticache/`.

[2] Intel® Optane™ Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`.

[3] Memcached. `code.google.com/p/memcached/`.

[4] Memcachier. `www.memcachier.com`.

[5] Redis Labs. `www.redislabs.com`.

[6] A. Aghayev and P. Desnoyers. Log-structured cache: Trading hit-rate for storage performance (and winning) in mobile devices. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM.

[7] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The rise of raas: the resource-as-a-service cloud. *Communications of the ACM*, 57(7):76–84, 2014.

[8] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. *SIGPLAN Not.*, 49(7):41–52, Mar. 2014.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[10] N. Beckmann and D. Sanchez. Bridging theory and practice in cache replacement. 2015.

[11] N. Beckmann and D. Sanchez. Modeling cache performance beyond LRU. *HPCA-22*, 2016.

[12] N. Beckmann and D. Sanchez. Maximizing cache performance under uncertainty. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 109–120. IEEE, 2017.

[13] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Resource-as-a-Service (RaaS) cloud. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA. USENIX.

[14] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[15] T. Blackwell, J. Harris, and M. I. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *USENIX*, pages 277–288, 1995.

[16] N. Bronson. Personal Communication, 2016.

[17] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[18] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[19] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[21] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[22] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a key-value cache that minimizes writes to flash. *CoRR*, abs/1702.02588, 2017.

[23] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[24] B. Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[25] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[26] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM.

[27] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[28] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[29] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450, 1994.

[30] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM.

[31] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[32] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 134–143. ACM, 1999.

[33] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, (12):1352–1361, 2001.

[34] C. Li and A. L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 5. ACM, 2015.

[35] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488. ACM, 2015.

[36] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[37] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 238–251, New York, NY, USA, 1997. ACM.

[38] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[40] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[41] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[42] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, Aug. 2015.

[43] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, Mar. 2016. USENIX Association.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[45] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *FAST*, pages 1–16, 2014.

[46] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 28:1–28:14, New York, NY, USA, 2014. ACM.

[47] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3. USENIX Association, 1993.

[48] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance compari-

son. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, pages 21–21. USENIX Association, 1995.

[49] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, Hollywood, CA, 2012. USENIX.

[50] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[51] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, Feb. 2015. USENIX Association.

[52] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical report, Cambridge, MA, USA, 1995.

[53] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.