Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

Asaf Cidon and Assaf Eisenman, *Stanford University;* Mohammad Alizadeh, *MIT CSAIL;*
Sachin Katti, *Stanford University*

Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

Asaf Cidon[1], Assaf Eisenman[1], Mohammad Alizadeh[2], and Sachin Katti[1]

[1]Stanford University

[2]MIT CSAIL

ABSTRACT

Web-scale applications are heavily reliant on memory cache systems such as Memcached to improve throughput and reduce user latency. Small performance improvements in these systems can result in large end-to-end gains. For example, a marginal increase in hit rate of 1% can reduce the application layer latency by over 35%. However, existing web cache resource allocation policies are workload oblivious and first-come-first-serve. By analyzing measurements from a widely used caching service, Memcachier, we demonstrate that existing cache allocation techniques leave significant room for improvement. We develop Cliffhanger, a lightweight iterative algorithm that runs on memory cache servers, which incrementally optimizes the resource allocations across and within applications based on dynamically changing workloads. It has been shown that cache allocation algorithms underperform when there are performance cliffs, in which minor changes in cache allocation cause large changes in the hit rate. We design a novel technique for dealing with performance cliffs incrementally and locally. We demonstrate that for the Memcachier applications, on average, Cliffhanger increases the overall hit rate 1.2%, reduces the total number of cache misses by 36.7% and achieves the same hit rate with 45% less memory capacity.

1. INTRODUCTION

Memory caches like Memcached [13], Redis [4] and Tao [8] have become a vital component of cloud infrastructure. Major web service providers such as Facebook, Twitter, Pinterest and Airbnb have large deployments of Memcached, while smaller providers utilize caching-as-a-service solutions like Amazon ElastiCache [1] and Memcachier [3]. These applications rely heavily on caching to reduce web request latency, reduce load on backend databases and lower operating costs.

Even modest improvements to the cache hit rate impact performance in web applications, because reading from a disk or Flash-based database (like MySQL) is much slower than a memory cache. For instance, the hit rate of one of Facebook's Memcached pools is 98.2% [5]. Assuming the average read latency from the cache and MySQL is $200\mu s$ and $10ms$, respectively, increasing the hit rate by just 1% would reduce the read latency by over 35% (from $376\mu s$ at 98.2% hit rate to $278\mu s$ at 99.2% hit rate). The end-to-end speedup is even greater for user queries, which often wait on hundreds of reads [26].

Web caching systems are generally simple: they have a key-value API, and are essentially an in-memory hash-table spread across multiple servers. The servers do not have any centralized control and coordination. In particular, memory caches are *oblivious* to application request patterns and requirements. Memory allocation across slab classes[1] and across different applications sharing a cache server is based on fixed policies like first-come-first-serve or static reservations. The eviction policy, Least-Recently-Used (LRU), is also fixed.

By analyzing a week-long trace of a popular caching service, Memcachier, we show that existing first-come-first-serve cache allocation techniques can be greatly improved by applying workload aware resource policies. We show that for certain applications, the number of misses can be reduced by 99%.

We propose Cliffhanger, a lightweight iterative algorithm that runs on memory cache servers. Cliffhanger runs across multiple eviction queues. For each eviction queue, it determines the *gradient* of the hit rate curve at the current working point of the queue. It then incrementally allocates memory to the queues that would benefit the most from increased memory, and removes memory from the queues that would benefit the least.

Cliffhanger determines the hit rate curve gradient of each queue by leveraging shadow queues. Shadow queues are an extension of the eviction queue that only contain the keys of the requests, without the values. We show that the rate of hits in the shadow queue approxi-

---

[1]To avoid memory fragmentation, Memcached divides its memory into several *slabs*. Each slab stores items with size in a specific range (e.g., $< 128B$, 128-256B, etc.) [2]

mates the hit rate curve gradient. Cliffhanger differs from previous cache allocation schemes in that it does not require an estimation of the entire hit rate curves, which is expensive to estimate accurately. We also prove that although Cliffhanger is incremental and relies only on local knowledge of the hit rate curve, it performs as well as a system has knowledge of the entire hit rate curve.

Prior work has shown that existing cache allocation algorithms underperform when there are performance cliffs [6, 29, 35]. Performance cliffs occur when a small increase in memory creates an unexpectedly large change in hit rate. In other words, performance cliffs are regions in the hit rate curve where increasing the amount of memory for the queue accelerates the increase in hit rate. Memcachier's traces demonstrate that performance cliffs are common in web applications.

We propose a novel technique that deals with performance cliffs without having to estimate the entire hit rate curve. The technique utilizes a pair of small shadow queues, which allow Cliffhanger to locally search for where the performance cliff begins and ends. With this knowledge, Cliffhanger can overcome the performance cliff and increase the hit rate of the queue, by splitting the eviction queue into two and dividing the traffic across the two smaller queues [6]. We demonstrate that this algorithm removes performance cliffs in real time.

Cliffhanger runs on each memory cache server and iteratively allocates memory to different queues and removes performance cliffs in parallel. We show that the performance increase resulting from both algorithms is cumulative. Cliffhanger supports any eviction policy, including LRU, LFU or hybrid policies such as ARC [25].

We have built a prototype implementation of Cliffhanger in C for Memcached. We demonstrate that Cliffhanger can achieve the same hit rate as Memcached's default cache allocation using on average only 55% of the memory. Our micro-benchmark evaluation based on measurements at Facebook [5] shows that Cliffhanger incurs a negligible overhead in terms of latency and throughput. Since Cliffhanger uses fixed-sized shadow queues, its memory overhead is minimal: less than $500KB$ for each application. Applications typically use $50MB$ or more on each server.

2. BACKGROUND

Multi-tenant web caches need to allocate memory across multiple applications and across requests within applications. Typically individual applications are statically assigned a fixed amount of memory across multiple servers. Within each application, each request occupies a position in the queue based on its order. To avoid memory fragmentation, Memcached divides its memory into several *slabs*. Each slab stores items with size in a specific range (e.g., $< 128B$, 128-256B, etc.) [2]. Each slab
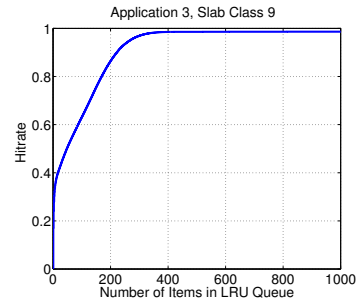

Figure 1: Hit rate curve for Application 3, Slab Class 9.

has its own LRU queue. Once the memory cache is full, when a new request arrives, Memcached evicts the last item from its corresponding slab's LRU queue.

There are several problems with this first-come-first-serve approach. First, slab memory is divided greedily, without taking into account the slab sizes. Therefore, in many applications, the large requests will take up too much space at the expense of smaller requests. Second, when applications change their request distribution, certain slab classes may not have enough memory and their queues will be too short. Some Memcached implementations have tried to solve the second problem by periodically evicting a page of a slab class, and reassigning it to the corresponding slab class of new incoming requests [26, 30]. However, even these improved slab class allocation may suffer from sub-optimal slab class allocation, and they still treat large and small items equally. Third, if one slab class suffers from a very high rate of compulsory misses (i.e., many items never get accessed more than once), the web cache operator may prefer to shift its resources to a different slab class that can achieve a higher hit rate with the same amount of memory.

These problem are not specific to slab-based memory allocation like Memcached. They also occur in systems like RAMCloud [28, 31] that assign memory contiguously in a log (i.e., in a global LRU queue). Regardless of the memory allocation approach, memory is assigned to requests greedily: when new requests reach the cache server they are allocated memory on a first-come-first-serve basis, without consideration for the request size or the requirements of the requesting application.

2.1 The Cache Allocation Problem

As a motivating example, we describe how to optimize resource allocation across slab classes for a single application in Memcached, where the goal is to maximize the overall hit rate. The same technique can also be applied to prioritize items of different sizes in a log-structured cache and to optimize memory across applications. The problem can be expressed as an optimization:

$$\begin{aligned} \underset{m}{\text{maximize}} \quad & \sum_{i=1}^{s} w_i f_i h_i(m_i) \\ \text{subject to} \quad & \sum_{i=1}^{s} m_i \le M \end{aligned} \quad (1)$$

Where s is the number of slab classes, f_i is the frequency of GETs for each slab class, $h_i(m_i)$ is the hit rate of each slab class as a function of the its available memory (m_i), and M is the amount of memory reserved by the application on the Memcached server. In case different queues have different priorities, we can assign weights (w_i) to the different queues. For simplicity's sake, throughout the paper we assume that the weights of all queues are always equal to 1.

To accurately solve this optimization problem, we need to compute $h_i(m_i)$, or the *hit rate curve* for each slab class. Stack distances [24] enable the computation of the hit rate curve beyond the allocated memory size. The stack distance of a requested item is its rank in the cache, counted from the top of the eviction queue. For example, if the requested item is at the top of the eviction queue, its stack distance is equal to 1. If an item has never been requested before, its stack distance would be infinite. Figure 1 depicts the stack distances for a particular slab class in Memcachier, where the X axis is the size of the LRU queue required to achieve a certain hit rate. In Dynacache [10] we demonstrated how to solve Equation 1 by estimating stack distances and using a simple convex solver. Equation 1 can be extended to maximize the hit rate across applications, or to assign different weights to different request sizes and applications.

Estimating stack distances for each application and running a convex solver is expensive and complex. Computing the stack distances directly is $O(N)$, where N is the number of requests. Instead, we estimated the stack distances using the bucket algorithm presented in Mimir [32]. This technique is $O(\frac{N}{B})$, where B is the number of buckets (we used 100 buckets). However, this technique is not accurate when estimating stack distance curves with tens of thousands of items or more. In addition, since application workloads change over time, the solver would need to run frequently, typically on a different node than the cache server. Furthermore, each server would need to keep track of the stack distances of multiple applications. This introduces significant complexity to the simple design of web caches like Memcached.

3. TRACE ANALYSIS

In this section, we analyze the performance of Memcached's default resource allocation. We show that there is great potential to improve the hit rate of Memcached by optimizing memory across different request sizes

App	Slab Class	% GETs	Original % Misses	Dynacache % Misses
4	0	9%	0%	7.4%
4	1	91%	100%	92.6%
6	0	1%	0.1%	2%
6	2	70%	92.6%	0%
6	5	29%	0.1%	0.2%

Table 1: Misses in two applications compared by slab class. Application 4's misses were reduced by 6.3% and Application 6's were reduced by 91.7%.

within each application. We then investigate and characterize performance cliffs.

Our analysis is based on a week-long trace of the top 20 applications (sorted by number of requests) running for a week on a server in Memcachier, a multi-tenant Memcached service. In Memcachier, each application reserves a certain amount of memory in advance, which is uniformly allocated across multiple Memcached servers comprising the Memcachier cache.

3.1 Resource Allocation in Memcachier

Figure 2 shows the hit rates and the number of misses that are reduced in Memcachier if we replace the default policy with the cache allocation using the Dynacache solver presented in Equation 1. We ran the solver on the week-long stack distances of each slab class. The results show that some applications benefit from a optimizing their memory across slab classes, and some do not. For example, the hit rate of Application 18 and 19 with full memory allocation is lower with the solver than the default algorithm. On the other hand, for some applications like Application 6, 14, 16 and 17, the number of misses is reduced by over 65%.

The solver's allocation is not optimal. This is due to several reasons. First, it assumes that all the hit rate curves are concave. We will explore this assumption later in the paper. Note that in Figure 2, the applications marked with asterisks are those that are not concave. Second, it requires that there be enough stack distance data points to accurately estimate the hit rate curve. If the requests for a slab class are too sparse, it will not be able to estimate its hit rate curve. Third, we ran the solver based on the trace of the entire week. If during that period the hit rate curves fluctuated considerably, the solver will not provide the ideal amount of memory for each slab class at any point in time. Due to these reasons, as we will show later, a heuristic dynamic cache allocation scheme can beat the solver.

3.2 Variance in Request Sizes

Table 1 demonstrates that the default scheme may assign too much memory to large slab classes, as evident
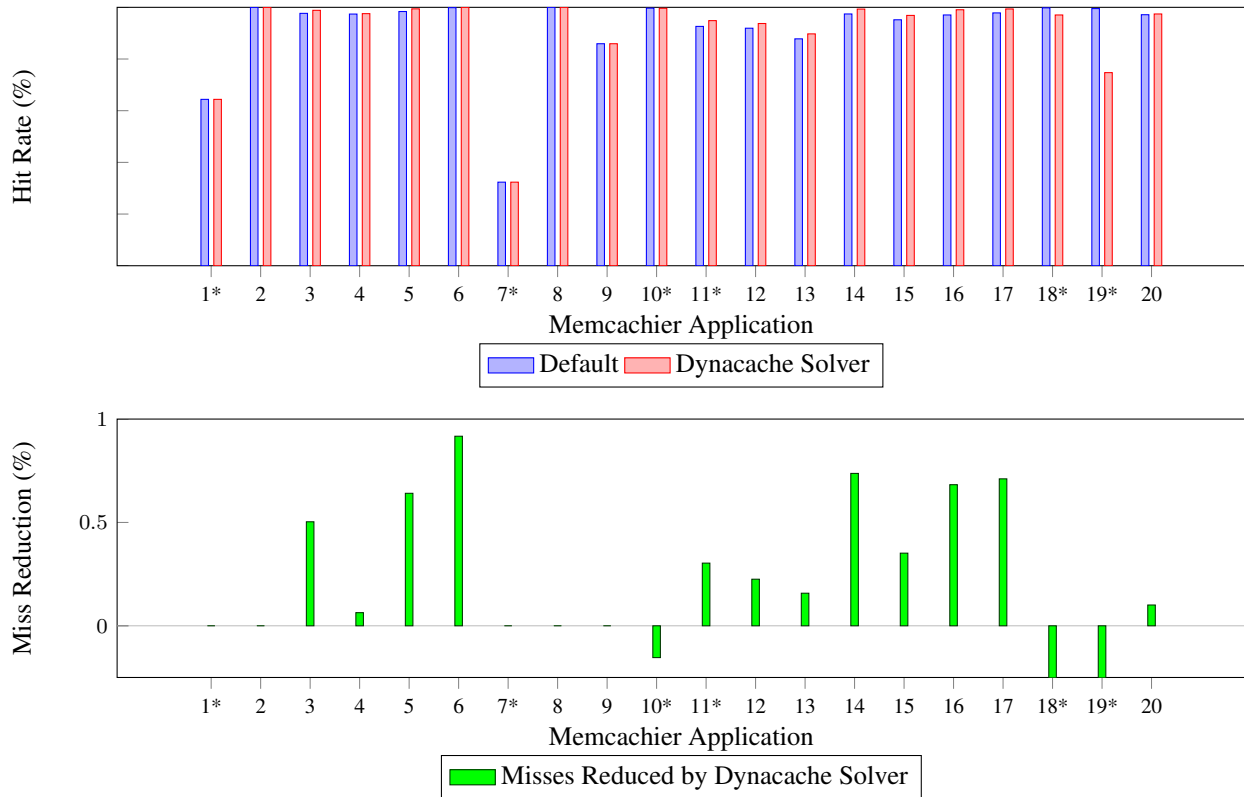
Figure 2: Hit rates and the number of misses reduced in Memcachier trace. Application 18 and 19's misses were increased by 13.6 and 51.5 times respectively. The applications that have an asterisk have performance cliffs.

Application	Original Hitrate	Log-structured Hitrate	Dynacache Solver Hitrate
3	97.7%	99.5%	98.8%
4	97.4%	97.8%	97.6%
5	98.4%	98.6%	99.4%

Table 2: Hit rates under log-structured memory and Dynacache solver.

in the case of applications 4 and 6. In application 6 this problem is much more severe, which is why the number of misses were reduced much more significantly by the Dynacache solver. The applications in the trace that did not see a significant hit rate improvement did not have much variance in terms of request size or were over-allocated memory.

The greedy nature of first-come-first-serve is not specific to Memcached's slab allocation scheme, it also occurs in contiguous Log Structured Memory (LSM). LSM [31] stores memory in a continuous log, rather than splitting it into slab classes, and it requires a log cleaner to run continuously to clear out stale items from the log and compact it. In memory caches, the benefit of LSM compared to slab classes would be the ability to run a global LRU queue, rather than having an LRU queue per slab class. To demonstrate this, we ran applications in three modes: with the default allocation of Memcached, the Dynacache solver allocation, and with a global LRU queue that simulates LSM. The global LRU simulates LSM with 100% memory utilization (such a scheme does not exist in practice). The results are displayed in Table 2. The results show that while LSM outperforms the Memcached slab allocator, in the case of application 5, even LSM running at 100% memory utilization may not perform as well as the optimized hit rate running on slab classes. The reason is that even in a global LRU queue large items may take space at the expense of small items.

3.3 Cross-application Performance

The Dynacache solver can be applied across applications running on the same memory cache server. To demonstrate the potential benefits of cross-application memory optimization, we applied the Dynacache solver to the top 5 applications in the trace. The results are displayed in Table 3. Note that in this example, we assigned each application the same weight. System operators can also assign different applications different weights in the optimization, for example, if certain applications belong

App	Original Memory Allocation %	Dynacache Solver Memory Allocation %	Original Hitrate	Dynacache Solver Hitrate
1	81%	69%	67.7%	67%
2	4%	13%	27.5%	38.6%
3	1%	1%	97.6%	97.6%
4	6%	8%	97.6%	98.1%
5	8%	9%	98.4%	98.5%

Table 3: Hit rates of the top 5 applications in the trace after we optimize their memory to maximize overall hit rate.

Figure 3: Examples of performance cliff in Memcachier traces.

to production systems and others do not.

3.4 Climbing Concave Hills

In order to solve Equation 1 we relied on estimating the entire hit rate curve. This requires a large number of stack distance data points and does not adapt to changing workloads. Instead, the same problem can be solved incrementally. If the gradient of each hit rate curve is known, we can incrementally add memory to the queue with the highest gradient (the one with the steepest hit rate curve), and remove memory from the queue with the lowest gradient. This process can be continued until shifting resources from one curve to the other will not result in overall hit rate improvement. This approach is called *hill climbing*.

The potential benefits of local hill climbing are that it can be conducted locally on each web-cache server, and that the algorithm is responsive to workload changes. If the hit rate curves change over time, the hill climbing algorithm incrementally responds. This leads us to ask: how can we measure the hit rate curve gradient locally without estimating the entire hit rate curve?

Our main insight is that the local hit rate curve can be measured using *shadow queues*. A shadow queue is an extension of an eviction queue that does not store the values of the items, only the keys. Items are evicted from the eviction queue into the shadow queue. For example, with an eviction queue of 1000 objects and a shadow queue of 1000 objects, when a request misses the eviction queue but hits the shadow queue, it means that if the eviction queue was allocated space for another 1000 objects, the request would have been a hit. The rate of hits in the shadow queue provides an approximation of the queue's local hit rate gradient.

3.5 Performance Cliffs

Hill climbing works well as long as the hit rate curves behave concavely and do not experience performance cliffs. This is true for some, but not for many web applications. Performance cliffs occur frequently in web applications: 6 out of the 20 top applications in our traces
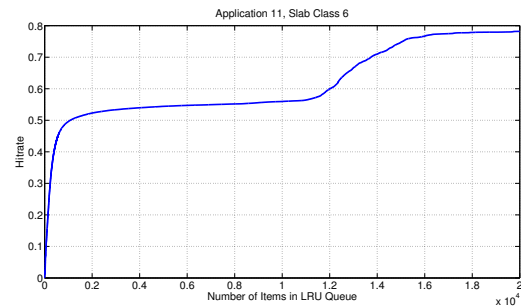
have performance cliffs. The applications in Figure 2 that have an asterisk are the ones with performance cliffs.

Figure 3 depicts the hit rate curve of a queue from the trace. Performance cliffs are thresholds where the hit rate suddenly changes as data fits in the cache. Cliffs occur, for example, with sequential accesses under LRU. Consider a web application that sequentially scans a 10 MB database. With less than 10 MB of cache, LRU will always evict items before they hit. However, with 10 MB of cache, the array suddenly fits and every access will be a hit. Therefore, increasing the cache size from 9 MB to 10 MB will increase the hit rate from 0% to 100%. Performance cliffs may hurt the hill climbing algorithm, because the algorithm will underestimate the gradient before a cliff, since it does not have knowledge of the entire hit rate curve. In the example of the queue in Figure 3, the algorithm can get stuck when the LRU queue has 10000 items.

Performance cliffs do not just hurt local-search based algorithms like hill climbing. They cause even bigger problems for solving optimization problems like the one described in Equation 1, since solvers assume that the hit rate curves do not have performance cliffs. For example, in application 19, due to the performance cliff, the solver approximates the hit rate curve to be lower than it is. This is why it fails to find the correct allocation for application 19, and significantly reduces its hit rate from 99.5% to 74.7%. In fact, optimal allocation is NP-complete without concave hit rate curves [6, 29, 35].

Resource allocation algorithms like Talus [6] and LookAhead [29] provide techniques to overcome performance cliffs for a single hit rate curve, but they require estimating the entire hit rate curve with stack distances. Since estimating stack distances introduces significant complexity and cost to web based storage systems, this leads us to ask: how can we overcome performance cliffs without estimating the entire hit rate curve?

4. DESIGN

In this section, we present the design of Cliffhanger, a hill-climbing resource allocation algorithm that runs on

Algorithm 1 Hill Climbing Algorithm
___
1: **if** request $\in$ shadowQueue(i) **then**
2:      queue(i).size = queue(i).size + credit
3:      chosenQueue = pickRandom({queues} - {queue(i)})
4:      chosenQueue.size = chosenQueue.size - credit
5: **end if**
___

each memory cache server and can scale performance cliffs. First, we describe the design of the hill climbing algorithm and show that it approximates the solution to the memory optimization problem. Second, we show how to overcome performance cliffs. Finally, we show how Cliffhanger climbs concave hit rate curves while navigating performance cliffs in parallel.

4.1 Hill Climbing Using Shadow Queues

The key mechanism we leverage to design our hill-climbing algorithm is shadow queues. Algorithm 1 depicts the hill-climbing algorithm, where $request$ is a new request coming into the cache server, $ShadowQueue(i)$ is the shadow queue of a particular queue (it can be the queue of a slab or a queue of an entire application), $pickRandom$ is a function that randomly picks one of the queues out the list of queues that we are optimizing. The algorithm is simple. Queue sizes are initialized so their capacity sums to the total available memory. When one of the shadow queues gets a hit, we increase its size by a constant credit, and randomly pick a different queue and decrease its size by the same constant credit. Once a queue reaches a certain amount of credits, it is allocated additional memory at the expense of another queue.

The intuition behind this algorithm is that the frequency of hits in the shadow queue is a function of the gradient of the local hit rate curve. In fact, we can prove that Algorithm 1 approximates the optimization described in Equation 1. Like before, we use the example of optimized memory across slab classes.

$$\text{maximize}_m \quad \sum_{i=1}^{s} f_i h_i(m_i)$$
$$\text{subject to} \quad \sum_{i=1}^{s} m_i \leq M$$

As a reminder, s is the number of slab classes, f_i is the frequency of GETs, $h_i(m_i)$ is the hit rate as a function of the slab's memory, and M is the application's total amount of memory. Assume that $h_i(m_i)$ are increasing and concave. The Lagrangian for this problem is:

$$\mathcal{L}(m, \gamma) = \sum_{i=1}^{s} f_i h_i(m_i) - \gamma(\sum_{i=1}^{s} m_i - M)$$

The optimality condition is:

$$f_i h_i'(m_i) = \gamma \text{ for } 1 \leq i \leq s$$
$$\sum_{i=1}^{s} m_i = M \quad (2)$$

Therefore, $f_i h_i'(m_i)$ is a constant (for any i), at the optimal solution. We show that with this algorithm, $f_i h_i'(m_i)$ is roughly constant in equilibrium for any i. To see why, consider the rate at which the credits for slab class i increase with a hit in its shadow queue:

$$f_i(h_i(m_i + \delta) - h_i(m_i)) \cdot \epsilon \approx f_i h_i'(m_i) \cdot \delta \cdot \epsilon$$

Here δ is the shadow queue size, and ϵ is the amount of credits we add to each queue when there is a hit in the shadow queue. The reason this approximation holds is that $f_i(h_i(m_i + \delta) - h_i(m_i))$ is the rate of hits that fall in the shadow queue. At the same time, the rate of credit decreases for class i is:

$$\frac{\sum_{j=1}^{s} f_j(h_j(m_j + \delta) - h_j(m_j)) \cdot \epsilon}{s} \approx$$
$$\frac{\sum_{j=1}^{s} f_j h_j'(m_j) \cdot \delta \cdot \epsilon}{s}$$

The reason this approximation holds is because when there is a hit in the shadow queue of any slab class j, we remove credits slab class i with probability $\frac{1}{s}$. In equilibrium the rate of credit increase and decrease have to be the same for every slab class. Therefore:

$$f_i h_i'(m_i) = \frac{\sum_{j=1}^{s} f_j h_j'(m_j) \cdot \delta \cdot \epsilon}{s} = \gamma$$

Since the right-hand side of the above equation does not depend on i, in equilibrium the gradients of each queue (normalized by the number of requests) are equal. This guarantees optimality (assuming concave hit rate curves) as shown in Equation 2.

4.2 Scaling Cliffs Using Shadow Queues

The hill climbing algorithm can get stuck in a sub-optimal allocation if the hit rates exhibit performance cliffs. We present for incrementally scaling performance cliffs. Our algorithm is inspired by Talus [6]. Talus introduced a queue partitioning scheme that scales performance cliffs, as long as the shape of the hit rate curve is known. Talus partitions a given queue to two smaller queues. By carefully choosing the ratio of requests it assigns to each queue and the size of the queues, Talus achieves the concave hull of the hit rate curve.
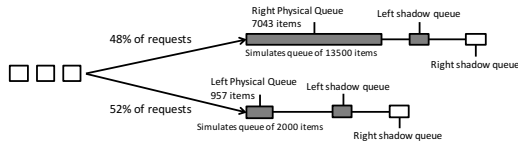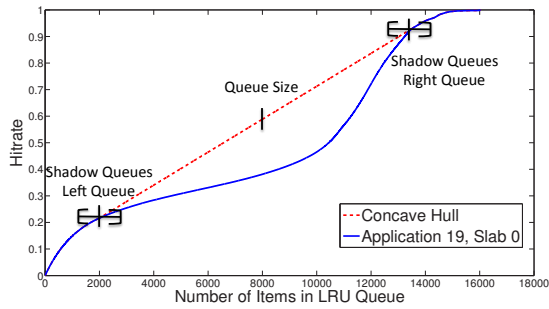
Figure 5: Illustration of Cliffhanger implementation. The darkly-colored sections represent the physical queue, which stores both the keys and the values of items, and the light sections are shadow queues.

Figure 4: Visualization of shadow queues on Application 19, Slab Class 0.

We demonstrate how Talus works with the hit rate curve of the smallest slab class of Application 19. Figure 4 depicts the concave hull of the hit rate curve. Suppose the slab class is currently allocated 8000 items. Talus allows us to achieve a hit rate that is a linear interpolation between any two points in the hit rate curve, by simulating two queues of different sizes. For example, if we select the points that correspond to 2000 and 13500 items, Talus can trace the linear curve between these two points on the hit rate curve graph. It does so, by simulating a queue of size 2000 and a queue of size 13500. The key insight is that since each of the smaller queues get a fraction of the requests, it causes them to behave as if they were larger queues. In this example, if we split an 8000 item queue into a queue of 957 items and a queue of 7043 items, and split the requests between the two queues using a ratio of 0.48% and 0.52% respectively, the first queue will simulate a queue of 2000 items (957 items seeing a ratio of 0.48% of the requests), and the second queue will be of a simulated size of 13500 items (7043 items with a ratio of 52% of the requests). By partitioning the requests into these two queues, the application can achieve the hit rate of the concave hull. For more information, see Talus [6].

However, in our setting we do not know the entire hit rate curve. Therefore, to apply the Talus partitioning, we need to dynamically determine whether the current operating point is on a performance cliff of the hit rate curve, or in other words, whether it is in a convex section of the curve. We also need to find the two points in the curve that will provide anchors for the concave hull.

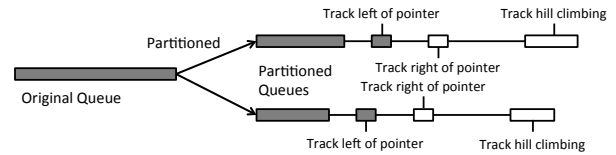The key insight to determining whether a certain queue is in a convex section of the graph, is to approx-

imate its second derivative. If the second derivative is positive, then the queue is currently in a convex area, a performance cliff. Similar to the hill-climbing algorithm, which locally approximates the first gradient with a shadow queue, to approximate the second derivative we use two shadow queues.

Each queue is split into two: a left and right physical queue. As long as the second derivative is negative (the function is concave), the left and right physical queues are the same size, and each receive half of the requests. Two evenly split queues behave exactly the same as one longer queue, since the frequency of the requests is halved for each queue, and the average stack distances of each request are also halved. Each one of the physical queues has its own shadow queue. The goal of these shadow queues is to find the points in the graph where the convex region ends. In the example of Application 19, these shadow queues are trying to locate points 2000 and 13500. In order to find the convexity region, each one of these shadow queues is in turn also split in half (right half and left half). The algorithm tracks whether each of the shadow queues receive hits in its right half or left half.

Algorithm 2 describes the cliff scaling algorithm. We initialize the algorithm by splitting the queue into half: a left and right physical queue, each with its own shadow queue, split in turn into two parts (the right and left half). The algorithm uses two pointers (right and left), which track the size of the simulated queues. The goal of the algorithm is to move the left and right pointers to the place in the hit rate curve where it starts and stops being convex. We initialize both of these pointers to the current size of the physical queue.

If the graph is in a convex point, the right pointer needs to be moved to the right, and the left pointer to the left, until it stops being convex. If each of the pointers are in a convex region of the graph, the rate of hits to the right of the pointer will be greater than to its left. When a request arrives to the server, we check if it hits the right or left shadow queue. If it hits the right half of the right shadow queue, we move the right pointer to the right. If it hits the left half, we move it to the left. Therefore, if the right pointer is in a convex region, the right pointer will move towards the top of the cliff. The left pointer moves in

Algorithm 2 Update Pointers

```
 1: function INIT
 2:     ratio = 1/2
 3:     rightPointer = leftPointer = queue.size
 4:     UPDATEPHYSICALQUEUES(ratio)
 5: end function
 1: function UPDATEPOINTERS(request)
 2:     if request ∈ rightShadowQueue then
 3:         if request ∈ rightShadowQueue.rightHalf then
 4:             rightPointer = rightPointer + credit
 5:         end if
 6:         if request ∈ rightShadowQueue.leftHalf AND
            rightShadowQueue.size > queue.size then
 7:             rightPointer = rightPointer - credit
 8:         end if
 9:     end if
10:     if request ∈ leftShadowQueue then
11:         if request ∈ leftShadowQueue.rightHalf then
12:             leftPointer = leftPointer - credit
13:         end if
14:         if request ∈ leftShadowQueue.leftHalf AND
            leftShadowQueue.size < queue.size then
15:             leftPointer = leftPointer + credit
16:         end if
17:         ratio = COMPUTERATIO(queue.size,
            rightPointer, leftPointer)
18:         UPDATEPHYSICALQUEUES(ratio)
19:     end if
20: end function
```

Algorithm 3 Compute Ratios

```
 1: function COMPUTERATIO(queue.size, rightPointer, leftPointer)
 2:     distanceRight = rightPointer - queue.size
 3:     distanceLeft = queue.size - leftPointer
 4:     if distanceRight > 0 AND distanceLeft > 0 then
 5:         requestRatio = distanceRight /
            (distanceRight + distanceLeft)
 6:     else
 7:         requestRatio = 0.5
 8:     end if
 9: end function
 1: function UPDATEPHYSICALQUEUES(ratio)
 2:     rightPhysicalQueue.size = rightPointer · (1 - ratio)
 3:     leftPhysicalQueue.size = leftPointer · ratio
 4: end function
```

the opposite direction. If it gets a hit to the right shadow queue, it moves left, and vice versa.

After adjusting the left and right pointers, Algorithm 3 updates the ratio of requests going to each physical queue, and their size. If the ratio is more than 0.5, more requests will be diverted to the left queue, and if the ratio is smaller than 0.5, more requests will be diverted to the right queue. As shown in Talus [6], the ratio is inverse to the distance of the position of the left and right shadow queues from the current operating point on the hit rate graph. In addition, the sizes of the right and left queues will also be updated based on this ratio. Their sum adds up to the current operating point (the queue size).

If Algorithm 2 does not see a performance cliff (i.e., a fully concave hit rate curve), the right and left pointers will not move from their starting points, because

there will be more hits on the left halves of both queues than the right halves, since the hit rate curve is concave. Therefore, the physical queue will be split in half and each half will receive half of the requests, which will result in the same hit rate of the original queue.

Note that Algorithm 2 can only efficiently scale a single cliff. If there are multiple cliffs in the hit rate curve graph, the algorithm will either scale only one of them, or the right and left shadow queue positions will scale multiple cliffs. In any case, even if multiple cliffs are scaled, the resulting concave hull will be at a higher hit rate than at the original hit rate curve, since the cliffs are convex. In the Memcachier traces, we did not find an example of multiple performance cliffs in any of the hit rate curves.

4.3 Combining the Algorithms

So far, we've introduced two algorithms: first, the gradient approximation (hill climbing) algorithm allows us to iteratively optimize the resource allocation across multiple queues, and works well as long as they do not have performance cliffs. Second, the second derivative approximation (cliff scaling) algorithm allows us to "flatten" performance cliffs to their concave hulls.

Cliffhanger combines both algorithms by utilizing two shadow queues: a small shadow queue to approximate the second derivative and detect and mitigate performance cliffs, and a longer shadow queue appended to the shorter shadow queue to approximate the first gradient and perform hill-climbing. When we get a hit in the larger shadow queue, we assign credits to the entire slab class. When we get a hit at the end of the physical queue or in the small shadow queue, we adjust the two pointers and update the ratio and the split between the left and right queue. Cliffhanger runs on each memory cache server and does not require any coordination between different servers. In addition, it can support any eviction policy, including LRU, LFU and other hybrid schemes.

5. EVALUATION

In this section we present the implementation and evaluation of Cliffhanger on the Memcachier traces and a set of micro benchmarks.

5.1 Implementation

We implemented Cliffhanger on Memcached in C. The shadow queues were implemented on a separate hash and queue data structures. In order to measure the end-to-end performance improvement across the Memcachier applications, we re-ran the Memcachier traces and simulated the hit rate achieved by Cliffhanger. In order to stress the implementation, we ran our micro benchmarks on an Intel Xeon E5-2670 system with 32 GB of RAM and an SSD drive, using a micro bench mark workload gen-
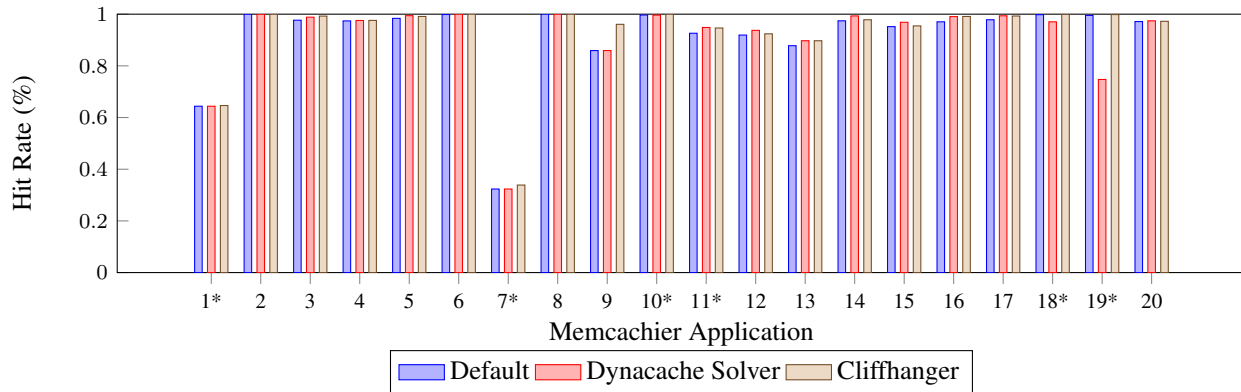
Figure 6: Hit rates of top 20 applications in Memcachier trace with Cliffhanger, compared to the Dynacache solver.
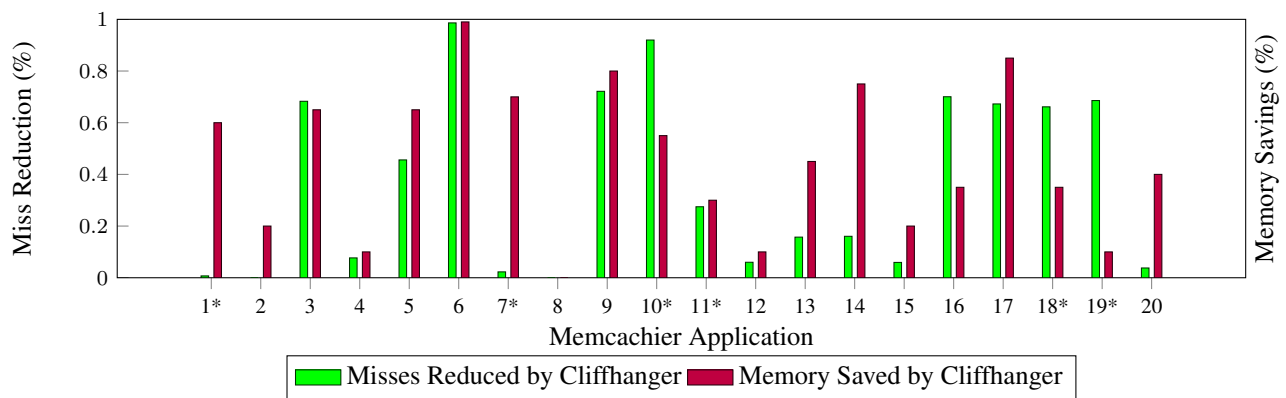

Figure 7: Miss reduction and amount of memory that can be saved to achieve the default hit rate of top 20 applications in Memcachier trace with Cliffhanger.

erated by Mutilate [19], a load generator that simulates traffic from the 2012 Facebook study [5].

Figure 5 is an illustration of the structure of the queues in Cliffhanger's implementation. Each queue is partitioned into two smaller queues (left and right queue). Each of the smaller queues needs to track whether items hit to the right or left of the pointers of the cliff scaling algorithm. In order to determine if an item hit to the left of the pointer, we do not need an extra shadow queue, since the section of the hit rate curve that is to the left of the pointer is already contained in the physical queue. To this end, our implementation tracks whether it sees hits in the last part of the queue (the last 128 items). In order to track hits to the right of the pointer, a 128 item shadow queue is appended after the physical queue. Finally, another shadow queue is appended to the end of each queue, to track hits for the hill climbing algorithm, since it requires a longer shadow queue.

The implementation only runs the cliff scaling algorithm when the queue is relatively large (over 1000 items), since it needs a large queue with a steep cliff to be accurate. It always runs the hill climbing algorithm,

including on short queues. When it runs both of the algorithms in parallel, the 1 MB shadow queue used for hill climbing is partitioned into two shadow queues in proportion to the partition sizes (e.g., if the smaller partition is 0.4 MB and the larger one is 1.6 MB, the shadow queue will be split into 0.2 MB and 0.8 MB). To avoid thrashing, the cliff scaling algorithm resizes the two queues only when there is a miss (i.e., when a miss occurs we insert the new item into the queue that is larger).

5.2 Miss Reduction and Memory Savings

Figure 6 presents the hit rate of Cliffhanger, and Figure 7 presents the miss reduction of Cliffhanger compared to the default scheme, and the amount of memory that Cliffhanger requires to produce the same hit rate as the default scheme. Cliffhanger on average increases the hit rate by 1.2% and reduces the number of misses by 36.7%, and requires 55% of the memory to achieve the same hit rate as the default scheme.

For some of the applications, the reduction in misses is negligible (less than 10%). In these applications there is not much room for optimizing the memory alloca-
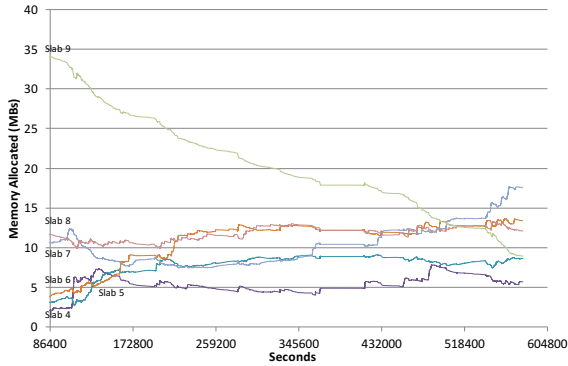
Figure 8: Memory allocated to slabs over time in Application 5, using Cliffhanger with shadow queues of 1 MB and 4 KB credits.

Slab Class	Original Hitrate	Cliff Scaling Hitrate	Hill Climbing Hitrate	Combined Algorithm Hitrate
0	38.1%	44.8%	95.3%	98.3%
1	37.3%	45.6%	67.4%	69.1%
Total Hitrate	37.3%	45.5%	70.3%	72.1%

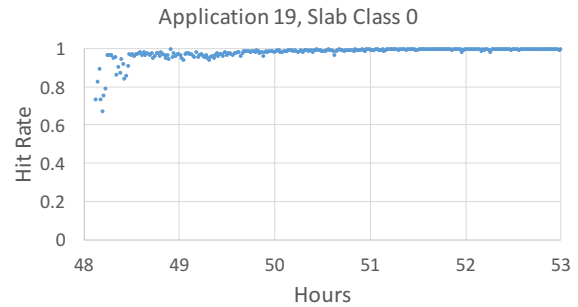**Table 4:** Comparing the hit rates with the default scheme, with the hill climbing and cliff scaling algorithms.

Figure 9: Hit rate of Application 19's Slab Class 0 under Cliffhanger, when the queue is in a region with a convex cliff.

tion based on request sizes. For some applications, such as Application 5, 13 and 16, the hit rate with Cliffhanger is very similar to the hit rate of the Dynacache solver. In some applications, like applications 9, 18 and 19, Cliffhanger significantly outperforms the Dynacache solver. The reason that Cliffhanger outperforms the Dynacache solver in these applications is that it is an incremental algorithm, and therefore it can deal with hit rate curve changes even in queues that are relatively small. For the Dynacache solver to work well, it needs to profile a larger amount of data on the performance of the queue, otherwise it will not estimate the concave shapes of the curves accurately (for more information, see Dynacache [10]). In addition, application 19 has steep performance cliffs, which hurt the performance of the Dynacache solver.

5.3 Constants and Queue Sizes

Both the hill climbing algorithm and cliff scaling algorithms require the storage designer to determine the size of the shadow queues and the amount of credits we award each queue when it gets a hit to its shadow queue. For example, the behavior of the hill climbing algorithm is demonstrated in Figure 8. The figure depicts the memory allocation across slabs, when we use the hill climbing algorithm with shadow queues of 1 MB and 4 KB credits, and shows that it takes several days for the algorithm to shift memory across the different slabs. The reason it takes several days is that the request rate on each Memcachier server is relatively low: about 10,000 requests per second, across 490 different total applications.

We found little variance in the behavior of the hill climbing algorithm when we use shadow queues over 1 MB. The cliff scaling algorithm is more sensitive to the size of the shadow queues, since it measures the second gradient. We found that we achieve the highest hit rate when we use shadow queues of 128 items for the cliff scaling algorithm. We experimented with using different credit sizes for both algorithms, and found that 1-4 KB

provide the highest hit rates when we run the algorithms across an entire week. If we use significantly larger credits sizes, the algorithms may oscillate their memory allocation, which could cause thrashing across the queues.

5.4 Combined Algorithm Behavior

To better understand the affect of the hill climbing and cliff scaling algorithms, we focus on Application 19 that has steep performance cliffs in both slab classes. Table 4 depicts the results of running Cliffhanger on Application 19 when we use queues of 8000 items so that the hillclimbing algorithm gets stuck in the performance cliffs in both of its slab classes. We compare it to the default first-come-first-serve resource allocation, and to running only Algorithms 1 and 2 separately.

This demonstrates the algorithms have a cumulative hit rate benefit. The hill climbing algorithm's benefit is due to a long period where the application sends requests belonging to Slab Class 0, and then sends a burst of requests belonging to Slab Class 1. The reason the cliff scaling algorithm improves the hit rate, is that both slab class 0 and 1 are stuck in a performance cliff.

The behavior of the combined algorithms is demonstrated in Figure 9. The queue starts at a hit rate of about 70%. It takes about 30 minutes to stabilize until it reaches a hit rate of about 99.7%.

5.5 Comparison with LFU Schemes

Much of the previous work on improving cache hit rates focuses on allocating memory between LRU

Application	Original Hitrate	Facebook Hitrate	Cliffhanger + LRU Hitrate	Cliffhanger + Facebook Hitrate
3	97.7%	97.8%	99.3%	99.3%
4	97.4%	97.6%	97.6%	97.6%
5	98.4%	98.5%	99.1%	99.1%

Table 5: Hit rates with the Facebook eviction scheme.

Algorithm	Operation	Cache Hit	Cache Miss
Hill Climbing	GET	0%	1.4%
Hill Climbing	SET	0%	4.7%
Cliffhanger	GET	0.8%	1.4%
Cliffhanger	SET	0.8%	4.8%

Table 6: Average latency overhead when the cache is full.

% GETs	% SETs	Throughput Slowdown
96.7%	3.3%	1.5%
50%	50%	3%
10%	90%	3.7%

Table 7: Throughput overhead when the cache is full and CPU bounded. The first row represents the GET/SET ratio in Facebook.

and LFU queues. We compared the performance of Cliffhanger with two such schemes: the first is ARC [25], which splits each queue to an LRU and LFU queue and uses shadow queues to dynamically resize them based similar to our algorithm. The second scheme is used by Facebook, in which the first time a request hits it is inserted at the middle of the queue. When it hits a second time, it is inserted to the top of the queue.

We found that ARC did not provide any hit rate improvement in any of the applications of the Memcachier trace. We found that most recently used items that are ranked high in the LFU queue, are also ranked high by LRU. Therefore, the LFU queue never gets any hits in its shadow queue and does not get more memory.

The Facebook scheme performed better than LRU, but did not perform as well as Cliffhanger with LRU. Table 5 presents the results with Applications 3, 4 and 5.

5.6 Micro Benchmarks

We observed negligible latency and throughput overhead with high hit rates, such as the hit rate of most applications in Memcachier, since in the case of a hit, the shadow queue does not add any latency. To analyze the overhead of Cliffhanger under a worst-case scenario, we used synthetic trace where all keys are unique and all queries miss the cache. In this scenario the cache is always full, and every single request causes shadow queue allocations and evictions and all the GETs perform lookups in the shadow queue. We warm up the caches for 100 seconds to fill the eviction queues and shadow queues, and only then start measuring the latency and throughput. The experiment utilizes the same key and value distribution described by Facebook [5].

Table 6 shows that the average latency in this worst-case scenario was between 1.4%-4.8%, when the request missed. When the request hit there was no latency over-

head with the hill climbing algorithm, since a hit does not require a lookup in the shadow queue, and 0.8% of latency with Cliffhanger, because we need to route the request between two physical queues. Table 7 presents the throughput overhead when the cache is full, which are identical when we are running the hill climbing algorithm alone and Cliffhanger. In any case, both Memcachier and Facebook are not CPU bound, but rather memory bound. Therefore, in both of these cases, increasing the average hit rate for applications at the expense of slightly decreased throughput at maximum CPU utilization is a favorable trade-off.

5.7 Memory Overhead

The memory overhead of Cliffhanger is minimal. The hill climbing algorithm uses shadow queues that represent 1 MB of requests. For example, with a 64 byte slab class the shadow queue will store 16384 keys, and with a $1KB$ slab class the shadow queue will store 1024 keys. The average key size is about 14 bytes. The cliff scaling algorithm uses a constant of 4 shadow queues (two left and right queues) of 128 items for each queue. Therefore, with the smallest slab class (64 bytes) the overhead will be $16384 + 128 \cdot 4 = 16896$ keys of 14 bytes for the smallest queue, which is about $200KB$ of extra memory for each queue. In Memcachier applications have 15 slab classes at most, and the overhead in the worst case will be $0.5MB$ of memory for each application.

6. RELATED WORK

There are two main bodies of related work. The first is previous work on improving the performance of web-based caches. The second is resource allocation techniques applied in other areas of caching and memory management, such as multi-core caches.

6.1 Web-based Memory Caches

Several systems improved the performance of memory cache servers by modifying their cache allocation and eviction policies. GD-Wheel [20] (GDW) uses the cost of recomputing the request in the database when prioritizing items in the cache eviction queue. This approach assumes the cache knows the recomputation cost in the database. Such information is not available to memory caches like Memcachier and Facebook, and would re-

quire changes to the memory cache clients. Regardless, Cliffhanger can be used with GDW as a replacement for LRU. Similar to Cliffhanger, GD-Size-Frequency (GDSF) [9] takes into account value size and frequency to replace LRU as a cache eviction algorithm for web proxy caches. GDSF relies on a global LRU queue, which is not available in Memcached, and on knowing the frequencies of each request. Unlike Cliffhanger, GDW and GDSF suffer from performance cliffs.

Mica [23], MemC3 [11] and work from Intel Labs [21] focus on improving the throughput of memory caches on multi-cores, by increasing concurrency and removing lock bottlenecks. While these systems offer significant throughput improvements over stock Memcached, they do not improve hit rates as a function of memory capacity. In the case of Facebook and Memcachier, Memcached is memory bound and not CPU bound.

Dynacache [10], Moirai [33, 34], Mimir [32] and Blaze [7] estimate stack distances and optimize resource allocation based on knowing the entire hit rate curve. Similar to Cliffhanger, Mimir approximates the hit rate curves using multiple buckets that contain only the keys and not the value sizes. Similarly, Wires et. al. profile hit rate curves using Counter Stacks [36] in order to better provision Flash based storage resources. All of these systems rely on estimation of the entire hit rate curve, which is generally more expensive and complex that local-search based methods like Cliffhanger, and do not deal with performance cliffs.

A recent study on the Facebook photo cache demonstrates that modifying LRU can improve web cache performance [14]. Twitter [30] and Facebook [26] have tried to improve Memcached slab class allocation to better adjust for varying item sizes, by periodically shifting pages from slabs with a high hitrate to those with a low hitrate. Unlike Cliffhanger, both of these schemes do not take into account the hit rate curve gradients and therefore would allocate too much memory to large requests.

Facebook [22], Zhang et al [37] and Fan et al [12] propose client-side proxies that provide better load-balacing and application isolation for Memcached clusters by choosing which servers to route requests to. While client-side proxies improve load-balancing, they do not control resource allocation within memory cache servers.

6.2 Cache Resource Allocation

Our work relies on results from multi-core cache partitioning. Talus [6] laid the groundwork for dealing with performance cliffs in memory caches, by providing a simple cache partitioning scheme that allows caches to trace the hit rate curve's concave hull, given knowledge of the shape of the convex portions of the hit rate graph. Talus relies on hardware utility monitors (UMONs) [29] to estimate stack distances and construct the hit rate

curves. In contrast, Cliffhanger does not rely on profiling stack distance curves to trace the concave hull, and is therefore more lightweight and can incrementally adapt to changes in the hit rate curve profile of applications. LookAhead [29] is another algorithm that deals with performance cliffs. Instead of tracing the concave hull, it simply looks ahead in the hit rate curve graph, and allocates memory to applications after taking into account the affect of performance cliffs. Like Talus, it also relies on having knowledge of the entire hit rate curve.

There is an extensive body of working on workload aware eviction policies for multi-core systems that utilize shadow queues. A prominent example is ARC [25], which leverages shadow queues to dynamically allocate memory between LRU and LFU queues. Cliffhanger also leverages shadow queues in order to locate performance cliffs and dynamically shift memory across slab classes and applications. There are many other systems that try to improve on variants of LRU and LFU, including LRU-K [27], 2Q [16], LIRS [15] and LRFU [17, 18]. In addition, Facebook has implemented a hybrid scheme, where the first time a request is inserted into the eviction queue, it is not inserted at the top of the queue but in the middle. We have found that for the Memcachier traces, Facebook's hybrid scheme does provide hit rate improvements over LRU, and that ARC does not.

7. CONCLUSION

By analyzing a week-long trace from a multi-tenant Memcached cluster, we demonstrated that the standard hit rate of a data center memory cache can be improved significantly by using workload aware cache allocation. We presented Cliffhanger, a lightweight iterative algorithm, that locally optimizes memory allocation within and across applications. Cliffhanger uses a hill climbing approach, which allocates more memory to the queues with the highest hit rate curve gradient. In parallel, it utilizes a lightweight local algorithm to overcome performance cliffs, which have been shown to hurt cache allocation algorithms. We implemented Cliffhanger and evaluated its performance on the Memcachier traces and micro benchmarks. The algorithms introduced in this paper can be applied to other cache and storage systems that need to dynamically handle different request sizes and varying workloads without having to estimate global hit rate curves.

8. ACKNOWLEDGMENTS

References

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

[2] Memcached. code.google.com/p/memcached/wiki/NewUserInternals.

[3] Memcachier. www.memcachier.com.

[4] Redis. redis.io.

[5] 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.

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

[7] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

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

[10] 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.

[11] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent Memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[12] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 23. ACM, 2011.

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

[14] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181. ACM, 2013.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. Leverich. Mutilate. github.com/leverich/mutilate/.

[20] 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.

[21] 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.

[22] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani. Introducing McRouter: A memcached protocol router for scaling Memcached deployments, 2014. https://code.facebook.com/posts/296442737213493.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[24] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

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

[26] 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.

[27] 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.

[28] 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.

[29] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[30] M. Rajashekar and Y. Yue. Twemcache. blog.twitter.com/2012/caching-with-twemcache.

[31] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured memory for DRAM-based storage. In *FAST*, volume 1, page 16, 2014.

[32] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[33] 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.

[34] 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. Technical Report CSRG-626, Department of Computer Science, University of Toronto, 2015.

[35] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[36] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[37] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang. Load balancing of heterogeneous workloads in Memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14), Philadelphia, PA*, 2014.