Adaptive Performance-Aware Distributed Memory Caching

Jinho Hwang and Timothy Wood
The George Washington University

Abstract

Distributed in-memory caching systems such as memcached have become crucial for improving the performance of web applications. However, memcached by itself does not control which node is responsible for each data object, and inefficient partitioning schemes can easily lead to load imbalances. Further, a statically sized memcached cluster can be insufficient or inefficient when demand rises and falls. In this paper we present an automated cache management system that both intelligently decides how to scale a distributed caching system and uses a new, adaptive partitioning algorithm that ensures that load is evenly distributed despite variations in object size and popularity. We have implemented an adaptive hashing system[1] as a proxy and node control framework for memcached, and evaluate it on EC2 using a set of realistic benchmarks including database dumps and traces from Wikipedia.

1 Introduction

Many enterprises use cloud infrastructures to deploy web applications that service customers on a wide range of devices around the world. Since these are generally customer-facing applications on the public internet, they feature unpredictable workloads, including daily fluctuations and the possibility of flash crowds. To meet the performance requirements of these applications, many businesses use in-memory distributed caches such as memcached to store their content. Memcached shifts the performance bottleneck away from databases by allowing small, but computationally expensive pieces of data to be cached in a simple way. This has become a key concept in many highly scalable websites; for example, Facebook is reported to use more than ten thousand memcached servers.

[1] Our system can be found in https://github.com/jinho10 as an open source project.

Large changes in workload volume can cause caches to become overloaded, impacting the performance goals of the application. While it remains common, overprovisoining the caching tier to ensure there is capacity for peak workloads is a poor solution since cache nodes are often expensive, high memory servers. Manual provisioning or simple utilization based management systems such as Amazon's AutoScale feature are sometimes employed [7], but these do not intelligently respond to demand fluctuations, particularly since black-box resource management systems often cannot infer memory utilization information.

A further challenge is that while memcached provides an easy to use distributed cache, it leaves the application designer responsible for evenly distributing load across servers. If this is done inefficiently, it can lead to cache hotspots where a single server is selected to host a large set of popular data while others are left lightly loaded. Companies such as Facebook have developed monitoring systems to help administrators observe and manage the load on their memcached servers [16, 18], but these approaches still rely on expert knowledge and manual intervention.

We have developed *adaptive hashing* that is a new adaptive cache partitioning and replica management system that allows an in-memory cache to autonomically adjust its behavior based on administrator specified goals. Compared to existing systems, our work provides the following benefits:

- A hash space allocation scheme that allows for targeted load shifting between unbalanced servers.
- Adaptive partitioning of the cache's hash space to automatically meet hit rate and server utilization goals.
- An automated replica management system that adds or removes cache replicas based on overall cache performance.

We have built a prototype system on top of the popular moxi + memcached platform, and have thoroughly evaluated its performance characteristics using real content and access logs from Wikipedia. Our results show that when system configurations are properly set, our system improves the average user reponse time by 38%, and hit rate by 31% compared to the current approaches.

2 Background and Motivation

Consistent hashing [10] has been widely used in distributed hash tables (DHT) to allow dynamically changing the number of storage nodes without having to reorganize all the data, which would be disastrous to application performance. Figure 1 illustrates basic operations of a consistent hashing scheme: node allocation, virtual nodes, and replication. Firstly, with an initial number of servers, consistent hashing calculates the hash values of each server using a hash function (such as md5 in the moxi proxy for memcached). Then, according to the predefined number of virtual nodes, the address is concatenated with "-X", X is the incremental number from 1 to number of virtual nodes. Virtual nodes are used to distribute the hash space over the number of servers. This way is particularly not efficient because the hash values of server addresses are not guaranteed to be evenly distributed over the hash space, which makes imbalances. This inefficiency is shown in Section 4.2.

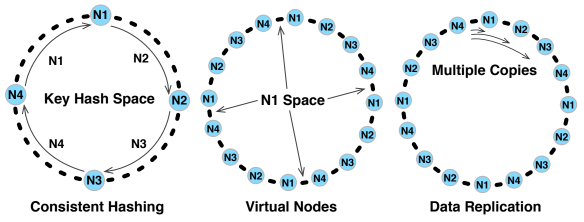


Figure 1: Consistent Hashing Operations; N_i is i^{th} cache node. Integer (32 bits) hash space consists of 2^{32} possible key hashes. Using virtual nodes somewhat helps to solve non-uniform key hash distribution, but it is not guaranteed; Also, data replication can help cache node faults.

Once the hash size for each server is fixed, it never changes even though they may have serious imbalances. Moreover, adding a new server may not significantly improve performance since node allocation is determined by hash values, which is a random allocation. Even worse, the consistent hashing scheme has no knowledge about the workload, which is a highly important variant [3].

As a motivating example, we randomly select 20,000 web pages among 1,106,534 pages in Wikipedia wikibooks database to profile key and value statistics. Figure 2(a) shows the number of objects when using 100 cache servers. Even though the hash function tends to provide uniformity, depending on the workloads the number of objects in each server can largely vary. The cache server that has the largest number of objects (659) has 15× more objects than the cache server with the smallest number of objects (42). This means that some cache servers use a lot more memory than others, which in turn worsens the performance. Figure 2(b) illustrates the object size has a large variation, potentially resulting in irregular hit rate to each server. Figure 2(c) describes the comparison between the number of objects and the size of objects in total. The two factors do not linearly increase so that it makes harder to manage the multiple number of servers. Figure 2(d) shows the average cache size per each object by dividing the total used cache size with the number of objects. From these statistics, we can easily conclude that consistent hashing needs to be improved with the knowledge of workloads.

3 System Design

The main argument against consistent hashing is that it can become very inefficient if the hash space does not represent the access patterns and cannot change over time to adapt to the current workload. The main idea of system design is that we adaptively schedule the hash space size for each memory cache server so that the overall performance over time improves. This is essential because currently once the size of hash space for each memory cache server is set, it never changes the configuration unless a new cache server is added or the existing server is deleted. However, adding/deleting a server does not have much impact since the assigned location is chosen randomly ignoring workload characteristics. Our system has three important phases: initial hash space assignment using virtual nodes, space partitioning, and memory cache server addition/removal. We first explain the memory cache architecture and assumptions used in the system design.

3.1 System Operation and Assumptions

There exist three extreme ways to construct a memory caching tier depending on the location of load-balancers: centralized architecture, distributed architecture, and hierarchically distributed architecture as shown in Figure 3. The centralized architecture handles all the requests from applications so that it can control hash space in one place which means object distribution can be controlled easily, whereas the load-balancers in the distributed architectures can have different configurations so that managing object distribution is hard. Since the centralized architecture is widely used structure in real memory caching deployments, we use this architecture in this paper. As load-balancers are implemented in a very efficient way minimizing the processing time, we

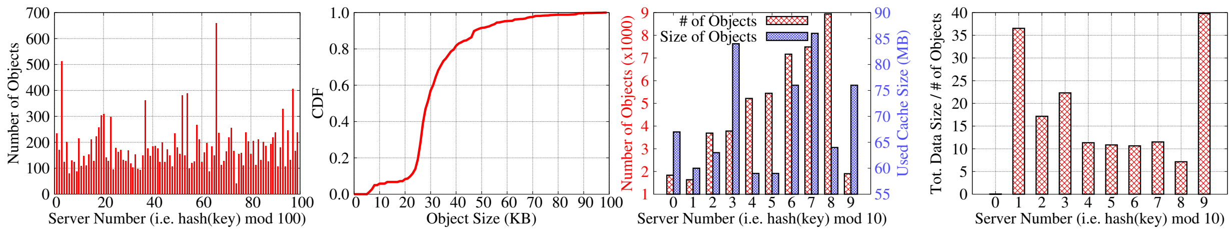

(a) Number of Objects (b) Object Size Distribution (c) # of Objects vs. Used Cache Size (d) Avg. Cache Size per Object

Figure 2: Wikibooks object statistics shows the number of objects in each server and used cache size are not uniform so that cache server performance is not optimized.

assume that the load-balancer does not become the bottleneck.

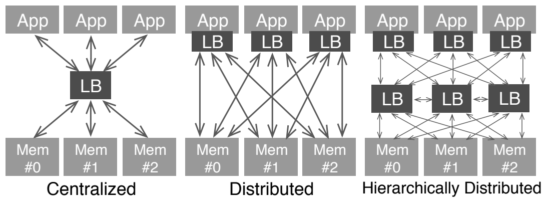


Figure 3: Memory Cache System Architecture; LB is load-balancer or proxy.

When a user requests a page from a web server application, the application sends one or more cache requests to a load-balancer – applications do not know there is a load-balancer since the implementation of the load-balancer is transparent. The load-balancer hashes the key to find the location where the corresponding data is stored, and sends the request to one of the memory cache servers (*get* operation). If there is data already cached, the data is delivered to the application and then to the user. Otherwise, the memory cache server notifies the application that there is no data stored yet. Then, the application queries the source medium such as database or file system to read the data, then sends it to the user and stores in the cache memory (*set* operation). Next time another user wants to read the same web site, the data is read from the memory cache server, resulting in a faster response time.

3.2 Initial Assignment

Consistent hashing mechanism can use "virtual nodes" in order to balance out the hash space over multiple memory cache servers so that different small chunks of the hash space can be assigned to each cache server. The number of virtual nodes is an administrative decision based on engineering experience, but it has no guarantee on the key distribution. Since our goal is to dynamically schedule the size of each cache server, we make a minimum bound on how many virtual nodes we need for schedulability.

Let $S = \{s_1, \ldots, s_{n_0}\}$ be a set of memory cache servers (the terms, memory cache server and node, are exchangeably used), where n_0 is the initial number of nodes. We denote v as the number of virtual nodes that each node has in the hash space H, and v_i as a virtual node i. That is, a node i can have $|s_i| = \frac{|H|}{n_0}$ objects, and a virtual node i can have $|v_i| = \frac{|H|}{n_0 \times v}$ objects, where $|H|$ is the total number of possible hash value. One virtual node can affect the other cache server in a clockwise direction as shown in Figure 1.

The key insight in our system is that in order to enable efficient repartitioning of the hash space, it is essential to ensure that each node has some region of the total hash space that is adjacent to every other node in the system. This guarantees that, for example, the most overloaded node has some portion of its hash space that is adjacent to the least loaded node, allowing a simple transfer of load between them by adjusting just one boundary. In order to allow every pair to influence each other, we need to make at least

$$v \geq \frac{{}^{n_0}\mathrm{P}_2}{n_0} = n_0 - 1, \qquad (1)$$

virtual nodes, where P is a permutation operation. Equation (1) guarantees that every node pair appears twice in a reverse order. So each physical node becomes $(n_0 - 1)$ virtual nodes, and the total number of overall virtual nodes becomes $n_0 \times (n_0 - 1)$. Also, we can increase the total number of virtual nodes by multiplying a constant to the total number. Figure 4 depicts an example assignment when there are five nodes. In a clockwise direction, every node influences all the other nodes.

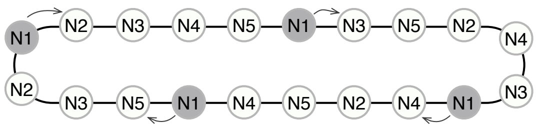


Figure 4: Assignment of Five Memory Cache Servers in Ring; As the example shows, N1 can influence all the other nodes N2, N3, N4, and N5. This applies to all the nodes.

Our node assignment algorithm is as follows. Let each node s_i have an array $s_{i,j} = \{(x,y) \mid 1 \le x \le n_0 \text{ and } y \in \{0,1\}\}$, where $1 \le i \le n_0$ and $1 \le j \le (n_0 - 1)$. Let $s^x_{i,j}$ and $s^y_{i,j}$ be x and y values of $s_{i,j}$, respectively. $s^x_{i,j}$ is defined as

$$s^x_{i,j} = \begin{cases} j & \text{if } j < i \\ j+1 & \text{if } j \ge i, \end{cases}$$

and all $s^y_{i,j}$ are initialized to 0. We pick two arbitrary numbers w_1 and w_2, where $1 \le w_1 \le n_0$ and $1 \le w_2 \le n_0 - 1$, assign w_1 in the ring, and label it as virtual node v_* in sequence ($*$ increases from 1 to $n_0 \times (n_0 - 1)$). Set $s^y_{w_1,w_2} = 1$, and $w_3 = s^x_{w_1,w_2}$. We denote $w_4 = (w_2 + k) \bmod n_0$, where $1 \le k \le n_0 - 1$. We then increment k from 1 to $n_0 - 1$, and check entries satisfying $s^y_{w_3,w_4} = 0$, and assign w_3 to w_1, w_4 to w_2, and $s^x_{w_3,w_4}$ to w_3. Repeat this routine until the number of nodes reaches $n_0 \times (n_0 - 1)$. For performance analysis, the time complexity of the assignment algorithm is $O(n_0^3)$ because we have to find $s^y_{w_3,w_4} = 0$ to obtain one entry each time, and there are $n_0 \times (n_0 - 1)$ virtual nodes. Therefore, the total time is $n_0(n_0 - 1)^2$. Note that this cost only needs to be paid once at system setup.

3.3 Hash Space Scheduling

As seen in Figure 2, key hash and object size are not uniformly distributed so that the number of objects and the size of used memory are significantly different, which in turn gives different performance for each memory cache server. The goal to use memory cache servers is to speed up response time to users by using a faster medium than the original source storage. Therefore, the performance of memory cache servers can be represented by the hit rate with the assumption that response time for all the cache servers are the same. However, usage ratio of each server should also be considered because the infrequent use of a cache server usually means the memory space is not fully utilized.

We define t_0 as the unit time slot for memory cache scheduling, which means the load-balancer repartitions the cache every t_0 time units. t_0 is an administrative preference that can be determined based on workload traffic patterns. Typically, only a relatively small portion of the hash space controllable by a second system parameter is rearranged during each scheduling event. If workloads are expected to change on an hourly basis, setting t_0 on the order of minutes will typically suffice. For slower changing workloads t_0 can be set to an hour.

In the load-balancer which distributes cache requests to memory cache servers, we can infer the cache hit rate based on the standard operations: *set* and *get*. A hit rate of a node s_i is

$$h_i = 1 - \frac{set(i)}{get(i) - set(i)},$$

where if $h_i > 1$, $h_i = 1$, and if $h_i < 0$, $h_i = 0$. “Hit rate” is a composite metric to represent both object sizes and key distribution, and this also applies when servers have different cache size. A simplified weighted moving average (WMA) with the scheduling time t_0 is used to estimate the hit rate smoothly over the scheduling times. Therefore, $h_i(t) = h_i(t-1)/t_0 + (1 - set(i)/get(i))$, where t is the current scheduling time and $t-1$ is the previous scheduling time. In each scheduling, $set(i)$ and $get(i)$ are reset to 0. We can also measure the usage ratio meaning how many requests are served in a certain period of time. The usage of a node s_i is $u_i = set(i) + get(i)$, and the usage ratio is $r_i = u_i / \max_{1 \le j \le n}\{u_j\}$, where n is the current number of memory cache servers, The usage ratio also uses a simplified WMA so that $r_i(t) = r_i(t)/t_0 + u_i/\max_{a \le j \le n}\{u_j\}$. In order to build up a scheduling objective with the hit rate and the usage ratio, we define a composite cost from hit rate and usage rate as $c = \alpha \cdot \overline{h} + (1-\alpha) \cdot r$, where $\alpha \in [0,1]$ is the impact factor to control which feature is more important and $\overline{h} = 1 - h$ is a miss rate, and state the scheduling objective as follows:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} (\alpha \cdot \overline{h_i} + (1-\alpha) \cdot r_i) \\ \text{subject to} & \overline{h_i} \in [0,1] \text{ and } r_i \in [0,1], \quad 1 \le i \le n \\ & \alpha \in [0,1] \end{array}$$

where n is the current number of the memory cache servers, and the objective is the sum of cost, and the conditions bind the normalized terms. This remains in a linear programming because we do not expand this to an infinite time span, which means the current scheduling state information propagates to the next scheduling only through hit rate WMA and usage ratio WMA. That is, we do not target to optimize all future schedulings, but the current workload pattern with small impact from past workload patterns. To satisfy the objective, we define the simple heuristic that finds the most cost disparity node pair with

$$s^*_{i,j} = max_{1 \le i,j \le n}\{c_i - c_j\}. \tag{2}$$

For performance analysis, since c is always non-negative, this problem becomes the problem finding a maximum cost and a minimum cost. Therefore, we can find proper pairs in $O(n)$ because only neighbor nodes are considered to be compared. Equation (2) outputs a node pair where $c_i > c_j$, so a part of the hash space in c_i needs to move to c_j for balancing out. The load-balancer can either just change the hash space or migrate objects from c_i to c_j. Changing just hash space would provide more performance degradation than data migration because the old cache space in c_i should be filled in c_j again by reading slow data source medium. The amount of hash space is determined by the ratio of two nodes as c_j/c_i to balance the space. Also, we define $\beta \in (0,1]$ to control the amount of hash space moved from one node to the other node. Therefore, we move data from node s_i in a counter clockwise direction (i.e., decreasing direction) of the consistent hash ring for the amount of

$$\beta \cdot (1 - \frac{c_j}{c_i}) \times |s_i|. \tag{3}$$

For example, if we start with five inital memory cache servers, and at the first scheduling point with $c_i = 1$, $c_j = 0.5$ and $\beta = 0.01$ (1%), we have to move c_i with the amount of $0.01 \cdot (1 - \frac{0.5}{1}) \times \frac{2^{32}}{20} = 1,073,741$. This means 0.5% of the hash space from s_i moves to s_j. With traditional consistent hashing, there is no guarantee that s_i has hash space adjacent to s_j, but our initial hash assignment does guarantee all pairs of nodes have one adjacent region, allowing this shift to be performed easily without further dividing the hash space.

3.4 Node Addition/Removal

Most current memory cache deployments are fairly static except for periodic node failures and replacements. We believe that these cache deployments can be made more efficient by automatically scaling them along with workloads. Current cloud platforms allow virtual machines to be easily launched based on a variety of critiera, for example by using EC2's *as-create-launch-config* command along with its CloudWatch monitoring infrastructure [5].

The main goal of adding a new server is to balance out the requests across replicas that overall performance improves. Existing solutions based on consistent hashing rely on randomness to balance the hash space. However, this can not guarantee that a new server will take over the portion of the hash space that is currently overloaded. Instead, our system tries to more actively assign a balanced hash space to the new server. The base idea is that when servers are overloaded – the loads cross upward the threshold line defined in advance based on service level agreement (SLA) and sustain the overloaded states for a predefined period of time – we find the most overloaded k servers with $s_i^* = max^k_{1 \le i \le n}\{c_i\}$ and support them with new servers, where an operator max^k denotes finding top k values. So, n_0 number of virtual nodes are added as neighbors of s_i^*'s virtual nodes in the counter clockwise direction. The new server takes over exactly half of the hash space from s_i^*, which is $\frac{|s_i|}{2}$. The left part of Figure 5 illustrates that s_j is overloaded and s_k is added. s_k takes over a half of the hash space s_j has.

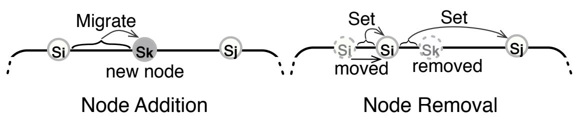


Figure 5: Object Affiliation in Ring After Node Addition and Removal

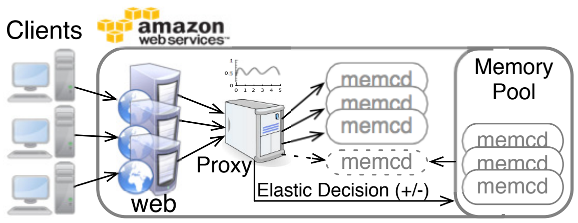
When a server is removed, the load-balancer knows about the removal by losing a connection to the server or missing keep-alive messages. Existing systems deal with node removal by shifting all the objects belonging to the removed node to the next node in a clockwise direction. However, this operation may make the next node overloaded and also misses a chance to balance the data over all the cache servers. When a node is removed in our system due to failure or managed removal – as with the adding criteria, the loads cross downward the threshold line and sustain the states – the two immediately adjacent virtual nodes will divide the hash space of the removed node. As shown in the right part of Figure 5, when there are three nodes s_i, s_k, and s_j in a clockwise sequence, and s_k is suddenly removed due to some reasons, the load-balancer decides how much hash space s_i moves based on the current costs c_i and c_j. s_i needs to move $\frac{c_j}{c_i+c_j} \times |s_j|$ amount of the hash space in a clockwise direction.

Of course, after a node is added or removed, the hash space scheduling algorithm will continue to periodically repartition hash space to keep the servers balanced.

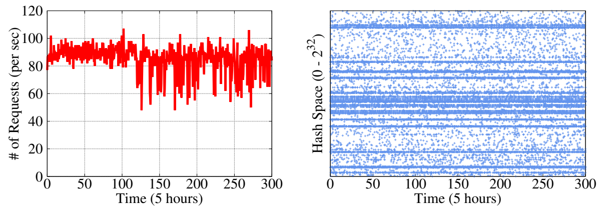
3.5 Implementation Considerations

To end our discussion of the system design, it is worth highlighting some of the practical issues involved in implementing the system in a cloud infrastructure. The scheduling algorithm is simple, and so is reasonable for implementation; however there exist two crucial aspects that must be addressed to deploy the system in the real infrastructure.

Data migration: When the scheduling algorithm schedules the hash space, it inevitably has to migrate some data from one server to another. Even though data are not migrated, the corresponding data are naturally filled in the moved hash space. However, since a response time between an original data source and a memory cache server are significantly different, users may feel slow response time [9]. The best way is to migrate the affected data behind the scene when the scheduling decision is made. The load-balancer can control the data migration by getting the data from the previous server and setting the data to the new server. The implementation should only involve the load-balancer since memory cache applications like memcached are already used in

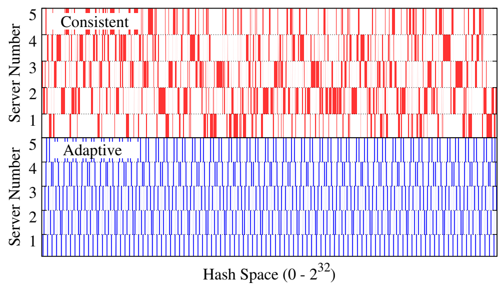


(a) Amazon EC2 Deployment

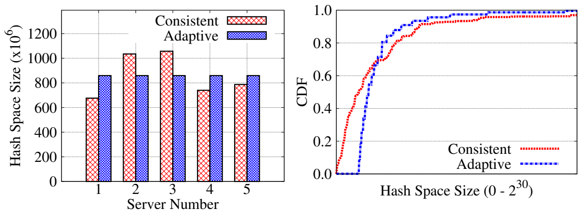


(b) Wikipedia Workload Characteristics: (1) # of Web Requests Per Second (left); (2) Key Distribution

Figure 6: Experimental Setup



(a) Initial Assignment Hash Map (5 servers)



(b) Hash Space Size (5 servers) (c) Hash Space Dist. (20 servers)

Figure 7: Initial hash space assignment with 5 - 20 memory cache servers.

many production applications. Also, Couchbase [6], an open source project, currently uses a data migration so that it is already publicly available.

Scheduling cost estimation: In the scheduling algorithm, the cost function uses the hit rate and the usage ratio because applications or load-balancers do not know any information (memory size, number of CPUs, and so on) about the attached memory cache servers. Estimating the exact performance of each cache server is challenging, especially under the current memory cache system. However, using the hit rate and the usage ratio makes sense because these two factors can represent the current cache server performance. Therefore, we implement the system as practical as possible to be deployed without any modifications to the existing systems.

4 Experimental Evaluation

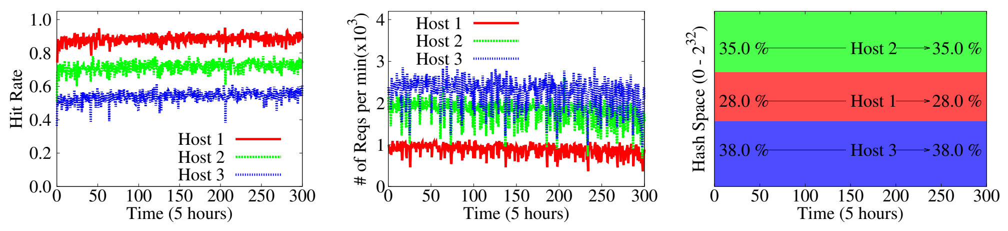
Our goal is to perform experiments in a laboratory environment to find out the scheduler behavior, and in a real cloud infrastructure to see the application performance. We use the Amazon EC2 infrastructure to deploy our system.

4.1 Experimental Setup

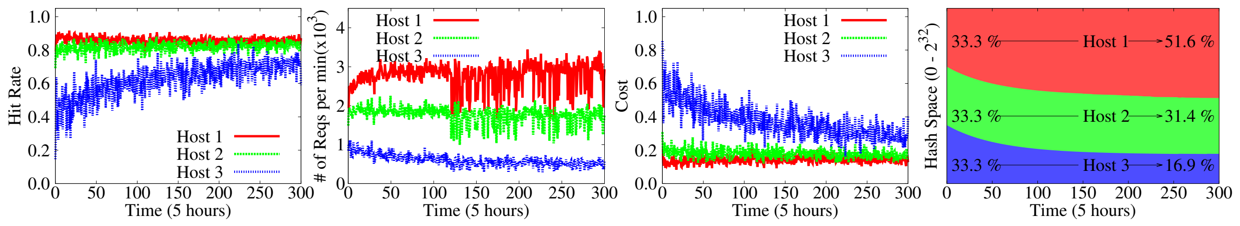
Laboratory System Setup: Five experimental servers, each of which has 4× Intel Xeon X3450 2.67GHz processor, 16GB memory, and a 500GB 7200RPM hard drive. Dom-0 is deployed with Xen 4.1.2 and Linux kernel 3.5.0-17-generic, and the VMs use Linux kernel 3.3.1. A Wikipedia workload generator, a web server, a proxy server, and memory cache servers are deployed in a virtualized environment. We use MediaWiki 1.14.1 [13], moxi 1.8.1 [15], and memcached 1.4.15 [14]. MediaWiki has global variables to specify whether it needs to use memory cache: wgMainCacheType, wgParserCacheType, wgMessageCacheType, wgMemCachedServers, wgSessionsInMemcached, and wgRevisionCacheExpiry. In order to cache all texts, we need to set wgRevisionCacheExpiry with expiration time, otherwise MediaWiki always retrieves text data from database.

Amazon EC2 System Setup: As shown in Figure 6(a), web servers, proxy, and memory cache servers are deployed in Amazon EC2 with m1.medium – 2 ECUs, 1 core, and 3.7 GB memory. All virtual machines are in us-east-*. Wikipedia clients are reused from our laboratory servers.

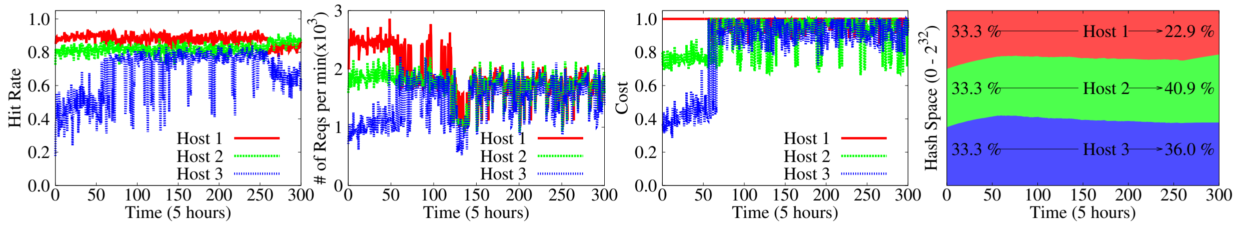
Wikipedia Trace Characteristics: Wikipedia database dumps and request traces have been released to support research activities [21]. January 2008 database dump and request traces are used in this paper. Figure 6(b) shows the trace characteristics of Wikipedia after we have scaled down the logs through sampling. Figure 6(b)(1) illustrates the number of requests per second from a client side to a Wikipedia web server. Requests are sent to a web server, which creates the requests sent to a proxy server and to individual memory cache servers depending on the hash key. Figure 6(b)(2) depicts the key distribution over the hash space 2^{32} range – most keys are the URL without a root address (e.g., ~~http://en.wikipedia.org/~~wiki/3D_Movie).



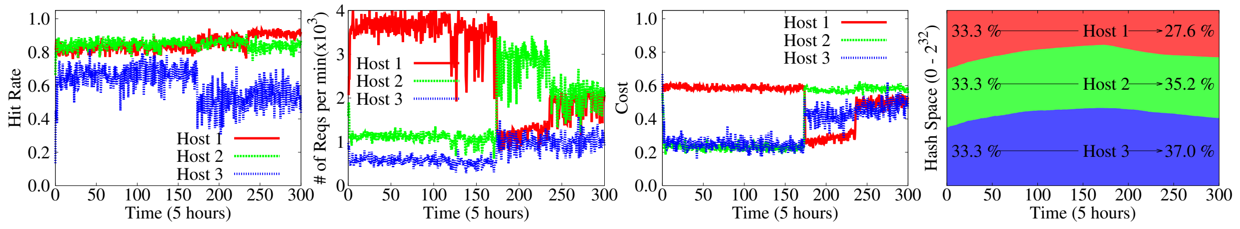
(a) Consistent Hashing



(b) Adaptive Hashing ($\alpha = 1.0$ and $\beta = 0.01$)



(c) Adaptive Hashing ($\alpha = 0.0$ and $\beta = 0.01$)



(d) Adaptive Hashing ($\alpha = 0.5$ and $\beta = 0.01$)

Figure 8: Hash Space Scheduling with Different Scheduling Impact Values α and β.

4.2 Initial Assignment

As we explain in Section 2 and 3.2, the initial hash space scheduling is important. Firstly, we compare the hash space allocation with the current system, ketama [11] – an implementation of consistent hashing. Figure 7 illustrates the initial hash space assignment. Figure 7(a) shows the difference between the consistent hashing allocation and adaptive hasing allocation when there are five memory cache servers. The number of virtual nodes is 100 (system default) for the consistent hashing scheme so that the total number of virtual nodes is 5×100. Our system uses the same number of virtual nodes by increasing the number of virtual nodes per physical node by a factor of $\frac{100}{(n_0-1)}$. With $n_0 = 5$, the total number of virtual nodes in our system is $5 \times 4 \times \frac{100}{4} = 500$. Consistent hashing has an uneven allocation without knowledge of workloads, which is bad. Adaptive hasing starts with the same size of hash space to all the servers, which is fair. Figure 7(b) compares the size of hash space allocated per node with each technique. In consistent hashing, the largest gap between the biggest hash size and the smallest hash size is 381,114,554. This gap can make a huge performance difference between memory servers. Figure 7(c) shows the hash size distribution across 20 servers—our approach has a less variability in the hash space assigned per node. Even worse, the consistent hashing allocation fixes the assignment based on a server's address, and does not adapt if the servers are not utilized well. We can easily see that without knowledge of workloads, it is hard to manage this allocation to make all the servers perform well in a balanced manner.

4.3 α Behavior

As described in Section 3.3, we have two parameters α and β to control the behaviors of the hash space scheduler. α gauges the importance of hit rate or usage rate. $\alpha = 1$ means that we only consider the hit rate as a metric of scheduling cost. $\alpha = 0$ means that we only consider the usage rate as a metric of scheduling cost. β is the ratio of the hash space size moved from one memory server to another. Since β changes the total scheduling time and determines fluctuation of the results, we fix β as a 0.01 (1%) based on our experience running many times. In this experiment, we want to see the impact of α parameter. Particularly, we check how α changes hit rate, usage rate, and hash space size. Our default scheduling frequency is 1 min.

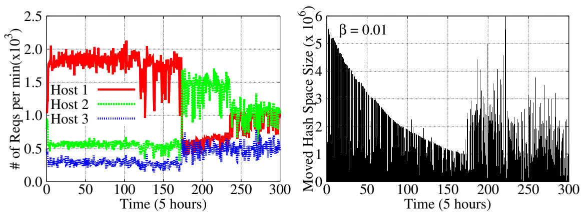
As a reference, Figure 8(a) illustrates how the current consistent hashing system works under the Wikipedia workload. The default hash partitioning leaves the three servers unbalanced, causing significant differences in the hit rate and utilization of each server.

Figure 8(b) shows the performance changes when $\alpha = 1.0$, which means we only consider balancing hit rates among servers. As Host 3 starts with a lower hit rate than other two hosts, the hash scheduler takes a hash space from Host 3 in order to increase its hit rate. The usage rate of host 3 decreases as its hash allocation decreases.

Figure 8(c) depicts the results when $\alpha = 0.0$, which means we only seek to balance usage rates among servers. The system begins with an equal hash space allocation across each host, but due to variation in object popularity, each host receives a different workload intensity. Since Host 1 initially has about 2.5 times as many requests per second arriving to it, the scheduler tries to rebalance the load towards Hosts 2 and 3. The system gradually shifts data to these servers, eventually balancing the load so that the request rate standard deviation across servers drops from 0.77 to 0.09 over the course of the experiment. This can be seen from the last (fourth) figure in Figure 8(c).

To balance these extremes, we next consider how α value (0.5) affects the performance. Figure 8(d) shows hit rate and usage rate of each server with $\alpha = 0.5$. Since the cost of each server is calculated out of hit rate and usage rate, the scheduler tries to balance both of them. As shown in the third graph in Figure 8(d), the costs balance among three servers which also means balancing both hit rate and usage rate.

Since workloads have different characteristics, the parameters α and β should be adjusted accordingly. We show further aspects of this adjustment while experimenting the system in the Amazon EC2 infrastructure.



(a) # Requests of Each Host (b) Moved Hash Size in Each Scheduling Time

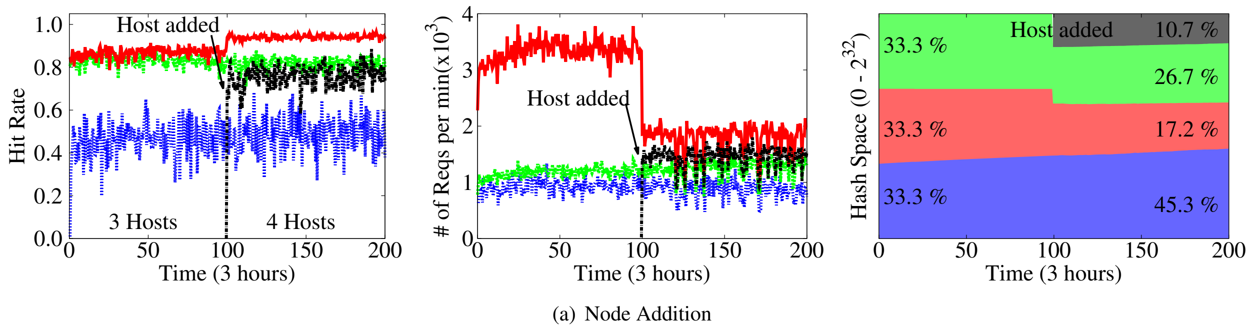
Figure 10: Hash Space Scheduling Analysis

4.4 β Behavior

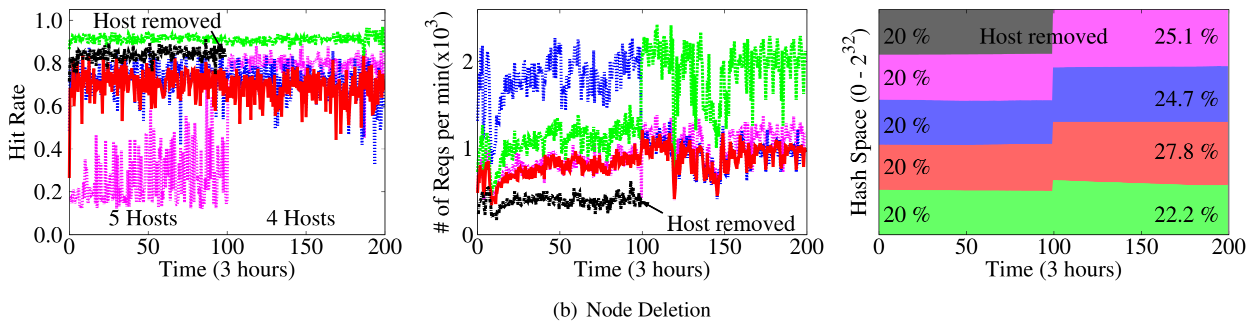
β value is the ratio of the amount of hash size moved in each scheduling time. We can show the behavior of β by illustrating the number of requests from each server (Figure 10(a)) and the amount of hash size per scheduling time (Figure 10(b)). As β value 0.01 yields approximately 1% of hash space from Equation (3), the moved hash size is decreasing as the hash space of an overloaded server decreases. Figure 10(b) shows the amount of hash space moved each interval. This continues to fall as Host 1's hash space decreases, but has relatively little effect on the request rate. However, after 180 minutes, a small, but very popular region of the hash space is shifted to Host 2. The system responds by trying to move a larger amount of data between hosts in order to rebalance them. This illustrates how the system can automatically manage cache partitioning, despite highly variable workloads.

4.5 Scaling Up and Down

As we explained in Section 3.4, adaptive hasing can autonomously add or delete memory cache servers based on the current performance. Since cloud infrastructure hosting companies provide a function to control the resources elastically, this is a very useful feature to prevent performance issues due to traffic bursts situation. Figure 9 shows the performance impact when adding a new server or deleting a server from the memory cache tier. Figure 9(a) starts with three memory cache servers, and a new server is added at 100 minutes due to an overloaded server. When a new server is added, the overloaded server gives 30% of its traffic to the new server so that overall usage rates of all servers are balanced. Conversely, Figure 9(b) assumes that one server out of five servers crashes at 100 minutes. As our initial hash allocation assigns the servers adjacent to one another, this gives a good benefit by distributing hash space to all other servers. This can be seen from the third graph in Figure 9(b).



(a) Node Addition



(b) Node Deletion

Figure 9: Memory Cache Node Addition / Deletion ($\alpha = 0.5$ and $\beta = 0.01$).

4.6 User Performance Improvement

The previous experiments have demonstrated how the parameters affect the adaptive hash scheduling system; next we evaluate the overall performance and efficiency improvements it can provide. We use Amazon EC2 to run up to twenty total virtual machines — three web servers, one proxy server, one database, and between 3 and 15 memory cache servers. We use five servers in our own lab to act as clients, and have them connect to the web servers using the Wikipedia workload.

We compare two caching approaches: a fixed size cluster of fifteen caches partitioned using Ketama's consistent hashing algorithm and our adaptive approach. The workload starts at 30K req/min, rises to 140K req/min, and then falls back to the original load over the course of five hours, as shown in Figure 12. We configure Ketama for a "best case scenario"—it is well provisioned and receives an average response time of 105 ms, and a hit rate of 70%. We measure our system with α values between 0 and 1, and initially allocate only 3 servers to the memcached cluster. Our system monitors when the request rate of a cache server surpasses a threshold for more than 30 seconds to decide whether to add or remove a node from the memory server pool. For this experiment, we found that more than 6K requests/sec caused a significant increase for the response time, so we use this as the threshold.

Figure 11 shows (a) average hit rate; (b) average response time from clients; (c) average standard deviation on number of requests to the EC2 cache servers; (d) number of used servers including dynamically added ones. Horizontal lines show the average performance of the current consistent hashing system used by moxi, and bars represent our system with different α values.

As expected, increasing α causes the hit rate to improve, providing as much as a 31% increase over Ketama. The higher hit rate can lower response time by up to 38% (figure b), but this is also because a larger α value tends to result in more servers being used (figure d). Since a large α ignores balance between servers (figure c), there is a greater likelihood of a server becoming overloaded when the workload shifts. As a result, using a high α does improve performance, but it will come at increased monetary cost for using more cloud resources. We find that for this workload, the system administrators may want to assign $\alpha = 0.5$, which achieves a reasonable average response time while requiring only a small number servers compared to Ketama.

Figure 12 shows how the workload and number of active servers changes over time for $\alpha = 1$. As the workload rises, the system adapts by adding up to five additional servers. While EC2 charges in full hour increments, our system currently aggressively removes servers when they are no longer needed; this behavior could easily be changed to have the system only remove servers at the end of each instance hour.

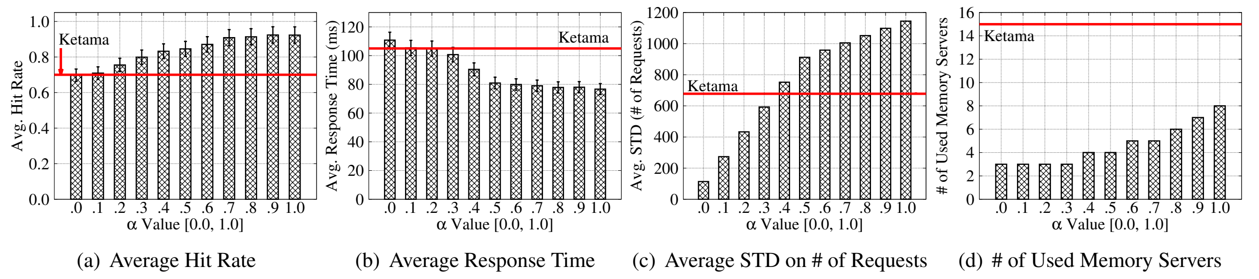


Figure 11: Amazon EC2 Deployment: Five Workload Generators, Three Web Servers, One Database, and Total 15 Memory Cache Servers in Memory Cache Pool; Three memory cache servers are used initially.

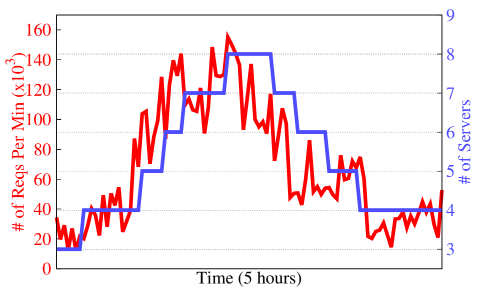


Figure 12: Number of cache servers adapting dynamically based on the workload intensity.

5 Related Work

Peer-to-peer applications gave rise to the need for distributed lookup systems to allow users to find content across a broad range of nodes. The Chord system used consistent hashing algorithms to build a distributed hash table that allowed fast lookup and efficient node removal and addition [20]. This idea has since been used in a wide range of distributed key-value stores such as memcached [14], couchbase [6], FAWN [2], and SILT [12]. Rather than proposing a new key-value store architecture, our work seeks to enhance memcached with adaptive partitioning and automated replica management. Previously, memcached has been optimized for large scale deployments by Facebook [16, 18], however their focus is on reducing overheads in the network path, rather than on load balancing. Zhu et. al. [22] demonstrate how scaling down the number of cache servers during low load can provide substantial cost savings, which motivates our efforts to build a cache management system that is more adaptable to membership changes. Christopher et. al. [19] proposes a prediction model to meet the strict service level objectives by scaling out using replication.

There are many other approaches for improving the performance of key-value stores. Systems built upon a wide range of hardware platforms have studied, including low-power servers [2], many-core processors [4], having front-end cache [8], and as combined memory and SSD caches [17]. While our prototype is built around memcached, which stores volatile data in RAM, we believe that our partitioning and replica management algorithms could be applied to a wide range of key-value stores on diverse hardware platforms.

Centrifuge [1] proposes a leasing and partioning model to provide the benefits of fine-grained leases to in-memory server pools without their associated scalability costs. However, the main goal of Centrifuge is to provide a simplicity to general developers who can use the provided libraries to model leasing and partioning resources. This work can be applied to managing the memory cache system, but Centrifuge does not support dynamic adaptation to workloads.

6 Conclusion

Many web applications can improve their performance by using distributed in-memory caches like memcached. However, existing services do not provide autonomous adjustment based on the performance of each cache server, often causing some servers to see unbalanced workloads. In this paper we present how the hash space can be dynamically re-partitioned depending on the performance. By carefully distributing the hash space across each server, we can more effectively balance the system by directly shifting load from the most to least loaded servers. Our adaptive hash space scheduler balances both the hit rate and usage rate of each cache server, and the controller can decide automatically how many memory cache servers are required to meet the predefined performance. The partitioning algorithm uses these parameters to dynamically adjust the hash space so that we can balance the loads across multiple cache servers. We implement our system by extending memcached and an open source proxy server, and test both in the lab and in Amazon EC2. Our future works include an automatic α value adjustment according to the workloads and a micro management of hot objects without impacting application performance.

Acknowledgments: We are grateful to our reviewers and shepherd for their comments. This work was supported in part by NSF grant CNS-1253575.

References

[1] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Amazon CloudWatch. `http://aws.amazon.com/cloudwatch`.

[6] Couchbase. vbuckets: The core enabling mechanism for couchbase server data distribution. *Technical Report*, 2013.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[8] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[9] Adam Wolfe Gordon and Paul Lu. Low-latency caching for cloud-based web applications. *NetDB*, 2011.

[10] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comput. Netw.*, 31(11-16):1203–1213, May 1999.

[11] Ketama. `http://www.audioscrobbler.net/development/ketama/`. 2013.

[12] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[13] MediaWiki. `http://www.mediawiki.org/wiki/MediaWiki`.

[14] Memcached. http://memcached.org.

[15] Moxi. http://code.google.com/p/moxi.

[16] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcached at facebook. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[17] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleswar K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. *2012 41st International Conference on Parallel Processing*, 0:470–479, 2012.

[18] Paul Saab. Scaling memcached at facebook, `http://www.facebook.com/note.php?note\_id=39391378919`.

[19] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. *Internation Conference on Autonomic Computing*, 2013.

[20] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.

[21] Erik-Jan van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master Thesis*, 2009.

[22] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.