Load Balancing of Heterogeneous Workloads in Memcached Clusters

Wei Zhang*, Timothy Wood and H. Howie Huang
The George Washington University
**Beihang University*

Jinho Hwang
IBM T. J. Watson Research Center

K.K. Ramakrishnan
Rutgers University

Abstract

Web services, large and small, use in-memory caches like memcached to lower database loads and quickly respond to user requests. These cache clusters are typically provisioned to support peak load, both in terms of request processing capabilities and cache storage size. This kind of worst-case provisioning can be very expensive (e.g., Facebook reportedly uses more than 10,000 servers for its cache cluster) and does not take advantage of the dynamic resource allocation and virtual machine provisioning capabilities found in modern public and private clouds. Further, there can be great diversity in both the workloads running on a cache cluster and the types of nodes that compose the cluster, making manual management difficult. This paper identifies the challenges in designing large-scale self-managing caches. Rather than requiring all cache clients to know the key to server mapping, we propose an automated load balancer that can perform line-rate request redirection in a far more dynamic manner. We describe how stream analytic techniques can be used to efficiently detect key hotspots. A controller then guides the load balancer's key mapping and replication level to prevent overload, and automatically starts additional servers when needed.

1 Introduction

In-memory caching has become a popular technique for enabling highly scalable web applications. These caches typically store frequently accessed or expensive to compute query results to lower the load on database servers that are typically difficult to scale up. Memcached has become the standard caching server for a wide range of applications.

Large web companies like Facebook and Twitter provision their caching infrastructure to support the peak load and to hold a majority of data in cache [1]. Given the dynamic nature of Internet workloads, this kind of static provisioning is generally very expensive. For example, the workload handled by the Facebook cache was shown to have a peak workload about two times higher than the minimum seen over a 24 hour period [2], and this may be even more dramatic for websites with a less global reach. Thus provisioning a cache infrastructure for peak load can be highly wasteful in terms of hardware and energy costs.

At the same time, public-facing applications need to be wary of flash crowds that direct a large workload to a small portion of the application's total content. In-memory caches are crucial for handling this type of load, but even they may become overloaded if popular data is not efficiently replicated or the system is unable to scale up the number of cache nodes in time. Further, one caching cluster is often multiplexed for several different applications, each of which may have distinct workload characteristics. The cache must be able to balance the competing needs of these applications despite differences in get/set rates, data churn, and the cost of a cache miss.

Heterogeneity can occur not only within keys and workloads, but also among the servers that make up the caching cluster. A wide range of key-value store architectures (many supporting the same memcached protocol) have been proposed, ranging from energy efficient FPGA designs [3] to high-powered data stores capable of saturating multiple 10 gigabit NICs [4, 5]. These approaches provide different trade-offs in the energy efficiency, throughput, latency, and data volatility of the cache, suggesting that a heterogeneous deployment of different server and cache types may offer the best overall performance.

While many resource management systems have been proposed for web applications, the caching tier is often ignored, leaving it statically partitioned and sized for worst case workloads. In part, this is because caches are typically accessed by a distributed set of clients (usually web servers), so dynamically adjusting the cache setup

requires coordination across a large number of nodes. To get around this problem, we eschew the traditional approach where clients know precise key-server mappings, and instead propose a middlebox-based load balancer capable of making dynamic adaptations within the caching infrastructure. Having a centralized load balancer is made possible by recent advances in high performance network cards and multi-core processors that allow network functions to be run on commodity servers [6–8]. Our system will be built upon the following components:

- A high speed memcached load balancer that can forward millions of requests per second.
- A hot spot detection algorithm that uses stream data mining techniques to efficiently determine the hottest keys.
- A two-level key mapping system that combines consistent hashing with a lookup table to flexibly control the placement and replication level of hot keys.
- An automated server management system that takes inputs from the load balancers and overall application performance levels to determine the number and types of servers in the caching cluster.

In this paper we describe our preliminary work on designing this dynamically scalable caching infrastructure. Our architecture provides greater flexibility than existing approaches that place complexity and intelligence in either the clients or memcached servers. By removing the reliance on manual, administrator specified policies, our self-managing cache cluster can automatically tune key placement and server settings to provide high performance at low cost.

2 Background

In this paper we focus on the memcached in-memory key value store. Memcached provides a simple put/get/delete interface and is primarily used to store small (e.g., $<$ 1KB) data values [2]. Clients, such as a PHP web application, generally follow the pattern of first requesting data from a cache node, but querying a database and loading the relevant entry into the cache if it was not found.

A memcached client can directly connect to a memcached server, or a proxy can be used to help manage the mappings of keys to servers. Individual memcached servers are designed so that they are unaware of each other. Many distributed key-value stores employ consistent hashing [9] to determine how keys are mapped to different servers [1, 10, 11]. This provides both an even key distribution and simplifies the addition and removal of servers into the key space.

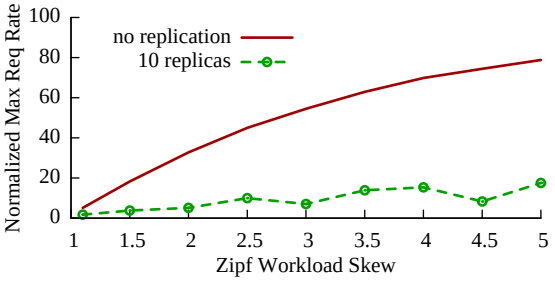

Figure 1: As workloads become more skewed (larger θ), the imbalance across nodes rises significantly relative to the load under a uniform workload.

To prevent a centralized proxy from becoming a bottleneck, each client machine typically runs its own local proxy instance that maintains the mapping of all key ranges to servers [1, 11]. While this reduces latency, coordinating their consistency can be a problem if there are a very large number of clients. As a result, the key to server mapping in these clusters is typically kept relatively static, *limiting the flexibility with which the cluster can be managed*.

2.1 Workload Heterogeneity

Workload characteristics can have a large impact on the performance of a memcached cluster since requests often follow a heavy tailed, Zipfian distribution. Figure 1 shows how the amount of skew in a Zipfian request distribution affects the number of key requests that occur on the most loaded server in a simulated cluster of 100 machines (normalized relative to the number of requests under a uniform workload). As the workload becomes more focused on a smaller number of keys, the imbalance across servers can rise significantly, but if the hot keys can be replicated to even a small number of servers, the balance is significantly improved.

In addition to varied key popularity, analysis of the Facebook memcached workload [1, 2] shows that different applications can have different read/write rates, churn rates, costs for cache misses, quality-of-service demands, etc. To handle these heterogeneous workloads, Facebook breaks their memcached cluster into groups, each of which services a different application or set of applications [1]. However, from their descriptions it appears that this partitioning is done in a manual fashion. This leaves it susceptible to inefficient allocations under dynamic conditions and may not be feasible for companies with less expertise in memcached cluster management.

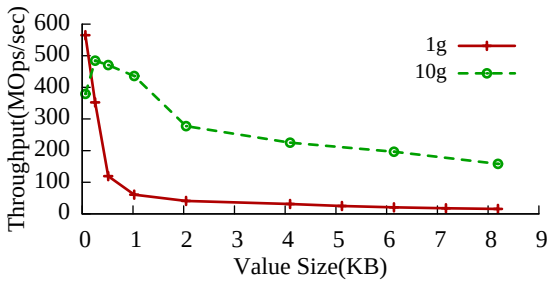

Figure 2: On our test system, a Gigabit NIC achieves higher performance for small value sizes, but the network quickly becomes saturated.

2.2 Server Heterogeneity

The hardware and software that make up a cache cluster can also be diverse. Not only are there several research [4, 10, 12] and commercial [13, 14] key-value store software solutions, many of them support the standard memcached protocol (even recent versions of MySQL). These alternatives are optimized for different use cases: e.g., couchbase [13] supports key replication for high availability and MICA [4] provides extremely high throughput, but only supports small value sizes. While many alternatives exist, memcached continues to be the most popular in-memory cache.

Figure 2 illustrates the performance of memcached using either an Intel 82599EB ten gigabit NIC or a Broadcom 5720 gigabit NIC on the same server with dual Intel Xeon X5650 CPUs. The ten gigabit NIC appears to suffer from inefficient processing of small packets, reducing its maximum throughput for very small sized objects. However, the gigabit network card quickly reaches its bandwidth limit when using larger value sizes. Since many web applications store primarily small objects [2], the added energy and hardware cost of ten gigabit adapters is not always necessary, suggesting that an intelligently scheduled mixed deployment could provide the best performance per dollar spent. Other hardware options such as low-power Atom CPUs have also been shown to provide valuable trade-offs when designing a memcached cluster [15].

3 Memcached Load Balancer Design

This section describes our preliminary design for a scalable, in-network dynamic load balancer for memcached clusters. Our overall architecture is shown in Figure 3. The load balancer is composed of a Lossy Counter used to detect key hot spots, a two level Key Director that stores where a key should be routed, and Key and Server managers that run the control algorithms to decide how the system should respond to workload changes. We describe these components in the following subsections.

3.1 Middlebox Platform

Recent advances in network interface cards (NICs) and user-level packet processing libraries [6, 7, 16] have enabled high speed packet processing on commodity servers. We are using these techniques to build an efficient load-balancing platform that can direct traffic to a large number of back-end memcached servers. Further, the load balancer's placement within the network path allows it to observe important statistics about the servers and their workloads.

Our prior work has demonstrated that even when a load balancer such as this is run inside a virtual machine, it is possible to achieve full 10 Gbps line rates [8]. A typical memcached request packet is approximately 96 bytes[1], so the maximum rate that can be handled by a single 10 Gbps NIC port is 13 million requests per second. Our current prototype can handle approximately 10 million 64-byte requests per second when using a single core to run a simplified version of the key redirection system described below.

As shown in Figure 3, client requests are sent as UDP packets to the IP of the load balancer, but replies are returned directly from the memcached servers back to the client. This significantly lowers the processing requirements of the load balancer since memcached responses are often much larger (and thus more expensive to process) than requests. The load balancer acts as a "bump in the wire", so it does not need to maintain any connection state, unlike existing proxies such as Twemproxy which establishes separate socket connections with each client and each server [11].

3.2 Hot Spot Detection

The load balancer must determine how to efficiently forward requests to memcached servers, while preventing some of them from becoming overloaded. Currently, our focus is on handling skewed workloads that cause a small number of servers to become overloaded. To prevent this, the load balancer must be able to detect which keys are causing the greatest load imbalance.

Since memcached does not store much data per key (e.g., true frequency over time) to limit overheads, we cannot simply rely on the servers sending this info. We propose a frequency counting mechanism to build a table of hot items in the request stream. Once hot keys are detected, requests for them can be either directed to more powerful servers or to replicas spread across several machines. The Lossy Counting algorithm [17] is a one-pass

[1] This is calculated based on 52 bytes for the MAC, IP, and UDP headers, 8 bytes for memcached application level header, plus the median memcached key size for Facebook's ETC pool [2]

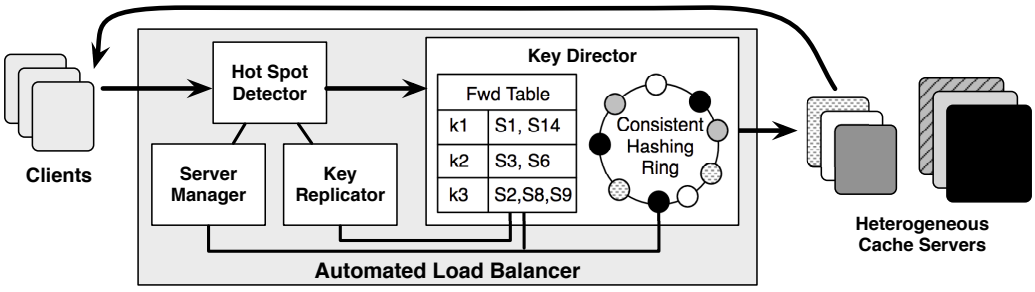

Figure 3: Requests are intercepted by the load balancer and directed based on either the forwarding table (hot items) or consistent hashing. Replies return directly back to the clients, minimizing the processing requirements of the load balancer.

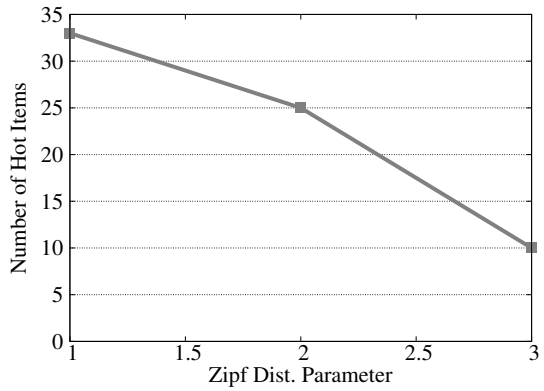

Figure 4: Number of (Steady-State) Hot Items with Different Workloads−Here, s is 1%, and ε is 0.1%

deterministic algorithm that efficiently calculates frequency counts over data streams by guaranteeing to identify hot items based on user-defined parameters−support threshold s and error rate ε, where $\varepsilon << s$. When the stream length is N, the lossy counting algorithm returns all keys with frequency at least sN, and there are no false negatives, which means no item with true frequency less than $(s-\varepsilon)N$ is returned.

Depending on the workload's skew, the actual number of hot items can be very different as shown in Figure 4. Thus the number of hot items found by the counter for a given s parameter depends both on the total number of keys being accessed in the cache and the workload distribution. Unfortunately, it is non-trivial to predict in advance what a workload will look like, and it may change with time. As a result, we have modified the Lossy Counting algorithm so that it will adjust itself to store a specified number of hot keys; we then adapt the target number of hot keys based on our observations of the workload during the previous observation window.

Hot Key Analysis: Figure 5 shows the estimated frequency (i.e., request rate) seen by the top keys measured by the Lossy Counter for a sample workload; the frequencies are an estimate guaranteed to be at most $\varepsilon * N$ smaller than the true counts [17]. Clearly, not all items reported by the counter should be treated identically since they have very different loads. The goal of our hot key analysis phase is to determine which of these potential keys need to be replicated or moved to faster servers.

The Lossy Counter can be used to separate keys into groups with similar request rates. Due to the nature of long tail distributions common in web workloads, the number of keys in the group with the highest average frequency will be smaller than the number of keys in the group with the second highest, and so on. This is shown in Figure 5 by the increasing width of each step. We consider the workload as a set of groups $g_1, g_2, ..., g_i, ..., g_n$ ordered such that g_1 is the key group with the highest request frequency, f_1. Our goal is to find the group g_i that splits the groups into two sets: $g_1...g_i$ represents the hot keys while $g_{i+1}...g_n$ are the "regular" keys.

The intuition behind our approach is to select g_i such that $|g_i|$, the number of keys in the group, is large enough to be evenly distributed. Any subsequent g_j where $j > i$ will have $|g_j| > |g_i|$ since we expect a heavy tailed workload distribution, and thus can be load balanced with simple consistent hashing.

We can use the common Balls and Bins analysis to understand how many keys (balls) are expected to be placed on each server (bin), if keys are uniformly assigned to each server. For a given group, we consider the request rate of all keys to be equivalent, so bounding the number of keys from a group that can be assigned to the most loaded server in turn bounds the maximum request rate it can achieve. For the case where there are fewer keys in a group than there are servers (i.e., $|g_j| < \#servers/log(\#servers)$, which is common for the hottest sets of keys) we can adapt the theorem from Mitzenmacher [18] that bounds the number of balls assigned to the most loaded bin with high probability ($p = 1 - 1/\#servers$):

$$MaxLoad \leq f_j \times \frac{log(\#servers)}{log(\#servers/|g_j|)} \qquad (1)$$

where f_j is the maximum request frequency to keys in

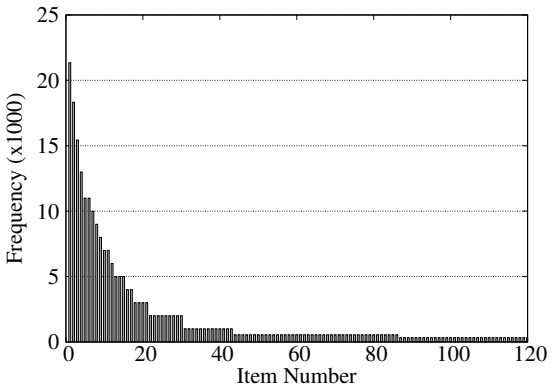

Figure 5: Frequency of each key measured by a Lossy Counter where s is 1%, ε is 0.1%.

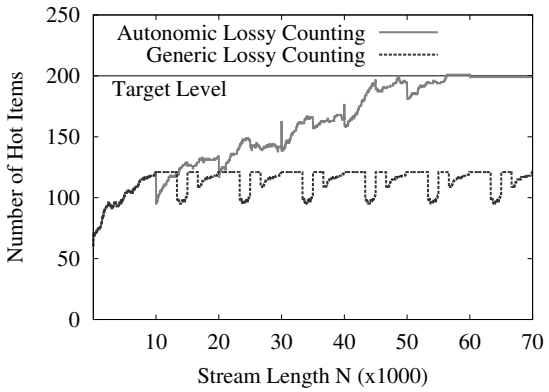

Figure 6: Autonomic Scaling on Number of Hot Items−Workload is a Zipfian distribution; initial s is 1%, ε is 0.1%, and α is 0.1%

g_j. The Hot Spot Detector considers each group in order starting with $j = 1$. If the maximum load exceeds a threshold, then group g_{j+1} will be considered for the split point. Once the split point g_i is found, all groups less than i are replicated as described in Section 3.3, and the rest are forwarded using consistent hashing.

Adaptive Sized Lossy Counter: The above analysis assumes that group g_i is included in the groups of keys returned by the Lossy Counter. However, if the counter is not configured with the appropriate support parameter, s, then the counter will track either too few groups (and not enough keys will be replicated) or too many groups (wasting memory and increasing the cost of lookups in the counter). To prevent this, we adapt the size of the Lossy Counter during each measurement interval to ensure it is tracking the correct number of keys.

The key request stream passes through the lossy counting algorithm for a configurable time window. At the end of the window, the algorithm compares the number of returned keys with T, which is a target level of hot items. We adapt T based on the request rates of the groups returned by the counter; if the last group returned by the counter will cause too much skew if it is not replicated, then T must be increased since the optimal g_i is not included in the counter's results.

Figure 6 illustrates the generic lossy counting algorithm and our autonomic lossy counting algorithm to show how we can adjust the desired level. Under the same workload, the generic lossy counting algorithm shows a steady amount of hot items, where as our autonomic lossy counting algorithm tries to reach the target level−here, the target level is defined as 200.

Lossy Counter Overhead: Since the Lossy Counter runs within the packet processing path, minimizing its overhead is critical. Our tests give an average total processing time of 367 nanoseconds per key counted (269 ns for lookup and 98 ns for insertion). Memcached request latencies are typically in the hundreds of microseconds, so this processing will add negligible performance impact. However, to achieve full line rate with a single core on the load balancer, each packet must be examined and redirected within about 80 nanoseconds. We are investigating how the counter operations can be done outside the critical path by separate cores that only sample a portion of the requests passing through the load balancer. The memory size is also very small: only 48 bytes plus 8 bytes (key hash and frequency) + key size per key.

3.3 Request Redirection

Directing keys to servers should be as fast as possible. Consistent Hashing is well known to effectively balance load across servers while supporting fast lookups and the flexibility to add or remove servers. We use consistent hashing to direct most keys to their destination, but use a lookup table to provide a more flexible mapping for hot keys. Incoming requests are first queried against the lookup table, and if not found there they are handled by the consistent hash ring.

The Hot Spot Detector provides a set of key/frequency pairs $(k_1, f_1)...(k_i, f_i)$ where k_i is the last key in g_i. The Key Redirector can then either replicate each key proportionally to its request frequency, or select some keys to be forwarded to servers that are known to be more powerful or less loaded. Recently there have been several efficient, concurrent hash table data structures proposed which we are exploring to allow the load balancer to efficiently check whether an incoming request is for a hot or regular key [19–21].

3.4 Server Management

The final component of our load balancer is the server manager. This component aggregates information from both the hot spot detector and from the memcached

servers themselves. This will allow the load balancer to also respond to broader workload dynamics that require servers to be added or removed from the memcached cluster.

There has been a significant amount of related work on how to dynamically manage virtual servers in response to workload changes. For example, VMware's Distributed Resource Scheduler can dynamically adjust CPU and memory allocations or migrate virtual machines. Yet without insight into statistics such as the cache's hit rate and the overall application's performance, these management actions will not be effective. An important distinction when managing caching servers is that workload skew can have a significant effect not only on request rates (as described in the previous sections), but also on the cache hit rate. Skewed workloads are actually easier to cache, meaning that an intelligent controller may be able to safely reduce the number of cache servers with minimal impact on the cache's hit rate [22]. Therefore, we are investigating what new control mechanisms and algorithms are necessary to provide a QoS management system that controls a cache cluster based on its internal behavior and the overall application's needs.

4 Related Work

There has been a large amount of work on improving the performance and scalability of individual memcached servers. Some work proposes using new hardware such as RDMA [23], high speed NICs [4], and FPGAs[24], while others have improved memcache's internal data structures [1, 20, 21]. These approaches all focus on maximizing the performance or energy efficiency of a single cache node.

A large scale analysis of Facebook's workload was presented by Atikoglu et al., which illustrates the high skew and time variation seen by what is probably the largest memcached deployment [2]. Facebook has also improved the efficiency of individual nodes and has deployed a key replication system to help balance load and increase the chance of finding multiple related keys in one server lookup [1]. Their system relies on individual clients knowing the mapping of all keys to servers, which we argue reduces the agility of the system compared to an in-network load balancer like we propose. They also appear to rely on manual classification of application workloads into different pools.

Fan et al. propose using a fast, small cache in front of a memcached cluster to prevent workload skew across servers even under adversarial workloads [25]. Our load balancer could potentially include a cache for fast local lookups, although the scalability of such an approach may be limited. This also increases the complexity of maintaining consistency. Our approach of forwarding some requests to high powered servers should provide a similar load balancing effect. Replication of Memcached keys was proposed by Hong et al [26]. Their system requires modification to both the clients and servers to maintain state about replicated keys, which we try to avoid with our transparent middle-box approach.

5 Conclusions and Future Work

Large-scale web applications rely on in-memory caches such as memcached to reduce the cost of processing common user requests. However, memcached deployments are typically statically sized and provisioned for peak workloads. We are developing a load balancing network middlebox that can automatically detect hotspots and balance load across memcached servers through request redirection and replication. Contrary to most resource management systems that require software to be installed either on clients or on the servers being managed, our in-network approach can be transparently deployed without any changes to applications. We believe that recent advances that allow such middleboxes to run at high speed even in virtual machines will open up new possibilities for a wide range of resource management systems that can be flexibly reconfigured and deployed.

In our ongoing work, we are continuing to extend our load balancer to optimize the number of servers and replicas of hot keys. We are also exploring how this kind of middle-box platform can be used to transparently monitor and manage other types of data center applications.

Acknowledgments

We thank the reviewers for their help improving this paper. This work was supported in part by NSF grants CNS-1253575, CNS-1350766, OCI-0937875, National Natural Science Foundation of China under Grant No. 61370059 and No. 61232009, and Beijing Natural Science Foundation under Grant No. 4122042.

References

[1] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, and Paul Saab, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, 2013, p. 385398.

[2] 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, New York, NY, USA, 2012, SIGMETRICS '12, p. 5364, ACM.

[3] Maysam Lavasani, Hari Angepat, and Derek Chiou, "An FPGA-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, vol. 99, no. RapidPosts, pp. 1, 2013.

[4] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky, "MICA: a holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, 2014, NSDI 14, p. 429444, USENIX.

[5] Wei Zhang, Timothy Wood, K.K. Ramakrishnan, and Jinho Hwang, "Smartswitch: Blurring the line between network infrastructure & cloud applications," in *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, June 2014, USENIX Association.

[6] Intel Corporation, "Intel data plane development kit: Getting started guide," 2013.

[7] Luigi Rizzo, "netmap: A novel framework for fast packet I/O," in *Presented as part of the 2012 USENIX Annual Technical Conference*, Berkeley, CA, 2012, pp. 101–112, USENIX.

[8] Jinho Hwang, K.K. Ramakrishnan, and Timothy Wood, "NetVM: high performance and flexible networking using virtualization on commodity platforms," in *Symposium on Networked System Design and Implementation*, Apr. 2014, NSDI 14.

[9] 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.*, vol. 31, no. 11-16, pp. 1203–1213, May 1999.

[10] 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*, New York, NY, USA, 2009, SOSP '09, p. 114, ACM.

[11] "Twemproxy: A fast, light-weight proxy for memcached," Feb. 2012, https://blog.twitter.com/2012/twemproxy.

[12] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman, "The case for ramcloud," *Commun. ACM*, vol. 54, no. 7, pp. 121–130, July 2011.

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

[14] Redis, "http://redis.io," .

[15] Joseph Issa and Silvia Figueira, "Hadoop and memcached: Performance and power characterization and analysis," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 1, no. 1, pp. 10, July 2012.

[16] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park, "mTCP: a highly scalable user-level TCP stack for multicore systems," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, 2014, NSDI 14, p. 489502, USENIX.

[17] Gurmeet Singh Manku and Rajeev Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*. 2002, VLDB '02, pp. 346–357, VLDB Endowment.

[18] Michael Mitzenmacher, *The Power of Two Choices in Randomized Load Balancing*, Ph.D. thesis, UC Berkeley, 1996.

[19] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky, "Scalable, high performance ethernet forwarding lookup," in *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.

[20] Bin Fan, David G. Andersen, and Michael Kaminsky, "MemC3: compact and concurrent memcache with dumber caching and smarter hashing," *Proc. 10th USENIX NSDI*, 2013.

[21] Yandong Mao, Eddie Kohler, and Robert Tappan Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, New York, NY, USA, 2012, EuroSys '12, p. 183196, ACM.

[22] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch, "Saving cash by using less cache," in *Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2012, HotCloud 12, USENIX.

[23] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler, "Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost memcached," in *USENIX Annual Technical Conference*, Boston, MA, 2012, USENIX ATC 12, p. 347353, USENIX.

[24] Michaela Blott, Kimon Karras, Ling Liu, Kees

Vissers, Jeremia Br, and Zsolt Istvn, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013, USENIX.

[25] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *ACM Symposium on Cloud Computing*, New York, NY, USA, 2011, SOCC '11, p. 23:123:12, ACM.

[26] Yu-Ju Hong and Mithuna Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *ACM Symposium on Cloud Computing*, New York, NY, USA, 2013, SOCC '13, p. 13:113:17, ACM.