NetKV: Scalable, Self-Managing, Load Balancing as a Network Function

Wei Zhang and Timothy Wood
The George Washington University

Jinho Hwang
IBM T.J. Waston Research Center

***Abstract*—Distributed key-value systems (e.g., memcached) are critical tools to cache popular content in memory, which avoids complex and expensive database queries and file system accesses. To efficiently use cache resources, balancing the load across a cluster of cache servers is important. Current approaches place a proxy at each client that can redirect requests across the cluster, but this requires modification to each client and makes dynamic replication of keys difficult. While a centralized proxy can be used, this traditionally has not been scalable.**

We design and implement NetKV, a scalable, self-managing, load balancer for memcached clusters. NetKV exploits recent advances in Network Function Virtualization to provide efficient packet processing in software, producing a high performance, centralized proxy that can forward over 10.5 million requests per second. NetKV efficiently and accurately detects hot keys using stream-analytic techniques, then replicates them to meet the allowed load imbalance bound set by administrators. NetKV uses "balls and bins" load analysis to adaptively determine the replication factor and set of hot keys. Our prototype adds minimal latency to each request, and our algorithms effectively balance load in both a 12 server cluster and a large-scale simulation driven by a trace of wikipedia requests.

***Index Terms*—Load Balancing; Network Function Virtualization; Key Value Stores; Scalability; Self Managing; Replication; Balls and Bins.**

I. Introduction

Key-value stores such as memcached have become critical tools for scaling up web applications. By caching popular content in an in-memory store, complex database queries and file system accesses can be avoided. However, since objects are cached in memory, a cluster of memcached servers must be deployed for large scale web applications—Facebook is reported to use more than 10,000 memcached servers [1].

Managing a distributed cluster of memcached servers is difficult since by their design, each memcached node is unaware of others in the cluster [2]. Further, web workloads typically exhibit highly skewed content request patterns, potentially causing a small number of keys to see far greater popularity than others [3]. Thus balancing the load across a cluster of memcached nodes is both a difficult and important challenge if server resources are to be used efficiently [4].

Existing approaches to memcached load balancing typically rely on placing a proxy at each client (typically a web application) so that requests can be load balanced across the cache cluster. However, this requires modifications to each client, and if there are many clients all accessing the cluster then coordinating them to maintain a consistent mapping of keys to cache servers can become difficult. Further, existing systems simply use consistent hashing to decide how keys should be distributed to servers, which will not evenly balance load across servers if there is a skewed distribution of requests. In this case, load can only be balanced by replicating hot content across multiple servers, but determining which keys should be replicated and where they should be placed is especially difficult if there are multiple client proxies that must coordinate to make these decisions.

Compounding all of these problems is the fact that a memcached cluster management system must be highly scalable and efficient. Redirecting requests must incur minimal latency on top of a normal memcached lookup, or the benefit of the in-memory cache will be lost. The massive number of unique objects stored in a memcached cluster means that it is infeasible to directly measure the popularity of all keys or to maintain lookup tables for where they are stored. Since these objects may be distributed across thousands of servers, it is also impractical to use resource management algorithms that require precise measurements of the load at each server.

To overcome these challenges, we present NetKV, a scalable, self-managing, load balancer for memcached clusters. NetKV exploits recent advances in Network Function Virtualization (NFV), which allows network services to be run in software while retaining performance on par with hardware implementations. NetKV can be transparently deployed within the network as an NFV-based middlebox, where it will automatically redirect requests to the appropriate servers. To balance load under skewed workloads, NetKV efficiently detects which keys are most popular using stream-analytic techniques [5], [6]. It then replicates keys across the cluster in order to probabilistically meet an administrator set bound on the allowed level of imbalance. NetKV makes the following contributions:

- A "balls and bins" based server load analysis that estimates bounds for the imbalance found in a cluster facing skewed workloads.
- A stream-analytics based hot key detection system that efficiently determines the most popular content in the cluster.
- A replication algorithm that uses the predicted load imbalance to decide how many times to replicate the detected hot keys, and which keys to store in a small local cache.
- A 'set' request re-ordering system that improves the consistency of writes to replicated keys with minimal

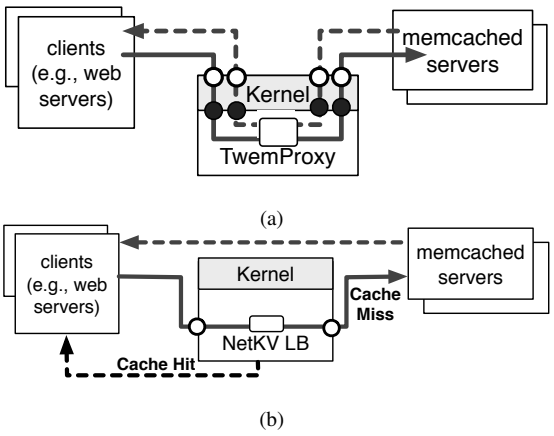

Fig. 1. (a) TwemProxy requires eight packet copies: 4 DMAs (light circles) and 4 kernel-user space copies (dark circles), to redirect a request and reply. (b) NetKV is built on an efficient NFV platform, eliminating packet processing overheads and allowing a single centralized load balancer to manage a large memcached cluster.

overhead.

- An efficient and scalable NFV-based load balancer prototype that can forward requests at line rates of 10Gbps or higher.

We have implemented NetKV using an NFV platform based on Intel's Data Plane Development Kit (DPDK) [7]. We demonstrate the load balancer's scalability up to 10.5 million requests per second and evaluate the replication algorithm's performance on a cluster of twelve servers, where it improves the throughput by approximately 20% compared to a static replication algorithm. We also use a simulator to illustrate how our load analysis and replication algorithms perform on a large cluster facing a trace of requests from Wikipedia. For extremely skewed loads, NetKV reduces load imbalance by 97% percent compared to a static replication algorithm.

II. Background and Motivation

Scalability & Consistency: Key Value stores such as memcached are simple independent caches, and require a separate load balancing system if data is to be distributed or replicated across a cluster. Load balancers for memcached must determine which keys to replicate and maintain a mapping of keys to servers responsible for storing the values.

Current approaches to memcached load balancing place a proxy at each client (e.g., each web server that will either contact the cache or a database to retrieve content) to redirect the requests across the cluster using an algorithm such as consistent hashing [8], [9]. However, this requires modifications to the clients, plus it makes it very difficult to perform dynamic cache management actions such as replication or key remapping since all of the client proxies must be kept consistent.

Deploying a centralized proxy that redirects requests across the cluster eliminates the consistency problem, but with current approaches is not scalable to large clusters. As Figure 1(a) shows, TwemProxy, a memcached proxy from Twitter, can incur eight packet copies for each request sent through the proxy, and all responses are mediated by the proxy, which adds further overhead. As a result, centralized proxies have not been practical when load balancing must be performed based on application level packet data.

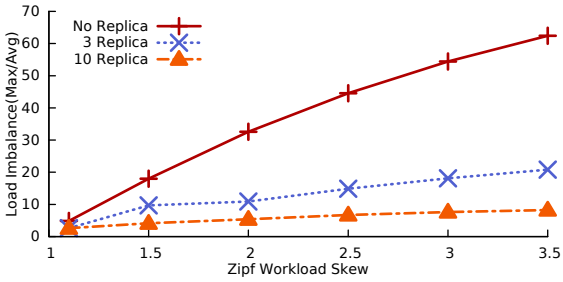

Fig. 2. Skewed workloads cause load imbalance, even with some replication.

Self Managing: Consistent hashing has been widely used to determine the key-server mapping for distributed data stores. However, web applications often face skewed Zipf distribution workloads [3], so servers hosting popular keys can quickly become a hotspot. As figure 2 shows, the most loaded server when using no replication is 78 times greater than the average load when zipf skew is 3.5, but this can be improved by replicating popular keys. Some systems such as Facebook's mcrouter [10] support static replication (i.e., maintaining N replicas for a fixed set of keys) to spread the load of popular content. However, dynamically determining which keys should be replicated and how many times to replicate each one to both balance load and minimize memory overhead remains a major challenge.

NetKV Approach: We have designed a scalable memcached-aware load balancer by exploiting recent advances in NFV to provide efficient packet processing. As Figure 1(b) shows, all requests to NetKV bypass the kernel, eliminating copies and context switches. The system is further optimized to send responses from memcached servers directly to clients, and by using a small local cache to speed up requests for the most popular content. This architecture allows NetKV to run as an efficient, centralized load balancer transparently deployed into the network.

NetKV targets large scale web applications with tens of millions of keys and dynamic workloads, where it is critical to efficiently detect hot keys and dynamically adjust their replication factor. To ensure NetKV's algorithms are both accurate and efficient, we employ stream analytic tools to detect hot keys and an automated "balls and bins" analysis that guides the replication to meet the acceptable load imbalance bounds set by administrators.

III. NetKV Architecture

Here we describe NetKV's core components which are illustrated in Figure 3.

Dispatcher: The Dispatcher interfaces with the NFV platform to efficiently read packets from the NIC and then determines how they should be handled: from a local cache, by a replication server, or through standard consistent hashing. If the request is for a key in the local cache, the dispatcher will

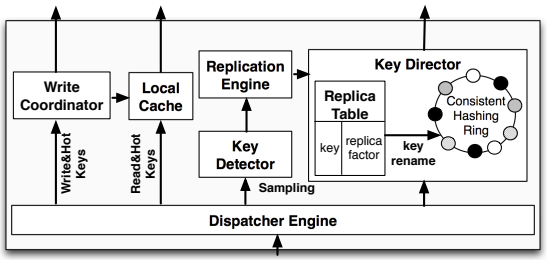

Fig. 3. NetKV Architecture

process the request and compose the response to send back to the client. Otherwise it forwards the request packet to a real memcached server.

Key Detector: The Key detector gathers statistics used by the replication algorithm to decide which keys are hot or cold. Since precisely tracking request rates for all keys would incur excessive overhead we rely on approximate data structures and sampling. During each measurement interval (e.g., 60 seconds) the Key Detector produces a list of potential hot keys with their estimated frequency, an approximation of the number of unique keys, and an estimate of the total number of requests.

Replication Engine: The Replication engine is triggered at the end of each measurement interval to determine which keys should be replicated and how often. The analysis is performed using the data gathered by the Key Detector, and the output is a lookup table that can be queried by the Dispatcher to quickly determine if a key is an unreplicated "cold key" or a "hot key" stored either in the local cache or across several servers.

Write Coordinator: NetKV uses multiple dispatcher threads to parallelize the forwarding of `get` requests, but doing so for `set` requests could cause consistency issues if keys are replicated. To prevent these problems, all requests that write to replicated data are sent to the Write Coordinator, which serializes the requests, similar to [11]. Since most web workloads are predominantly read intensive (e.g., 98% reads in Facebook [3]), this serialization incurs relatively little overhead while improving consistency.

Local Cache: NetKV maintains a Local Cache on the load balancer to quickly handle requests for keys which are popular and also frequently updated. Since this cache is explicitly managed with a single writer and multiple reader threads, and interacts directly with the NFV platform to send packets, it can achieve throughputs orders of magnitude greater than memcached (albeit with far less cache space than a full cluster).

IV. Replicating Hot Data

NetKV faces two challenges in order to balance load across the cluster's servers: 1) it must decide which keys are the most popular, and 2) it must determine how many times to replicate each of those keys in order to balance load without incurring undue memory overhead.

The novelty of our approach is to take a skewed key distribution and replicate the popular keys enough times so that their workload will appear uniformly distributed; we then can use a "balls and bins" style analysis to estimate the maximum load that will be incurred on any server under such a replication pattern. This then allows NetKV to automatically adapt the replication parameters to meet a target load–all while leveraging stream approximation techniques to ensure the analysis can be performed in a timely fashion.

A. Popularity Detection

NetKV's Key Detector first must separate popular keys from less frequently accessed ones. Efficiently determining the 'top-k' elements in a data stream (i.e., the k most popular keys) is an open challenge in streaming analytics [5]. We use the Lossy Counter [6] data structure to efficiently determine which are the most popular keys. A Lossy Counter uses a window to record recently observed keys and periodically removes the low count keys and keeps the most-frequently accessed ones.

The Lossy Counter returns a list of key/frequency pairs, (k_i, f_i), from most to least popular. The number of keys tracked and accuracy of the counts depends on the support threshold s and error rate ϵ, which control the window size. The Lossy counter guarantees that all keys with frequency at least $s \times N$ will be returned. Any key with frequency less than $(s-\epsilon) \times N$ will not be returned. N is the stream length. We set these parameters conservatively so that the counter will track more keys than NetKV will choose to replicate. If during a measurement interval NetKV determines that all keys in the counter must be replicated, then it will automatically adjust ϵ to get a larger window size to ensure that in the subsequent measurement interval the counter will track a larger number of keys.

In addition to tracking the frequency of popular keys, the Key Detector also uses a HyperLogLog counter to estimate the total number of unique keys. HyperLogLog counters are extremely efficient cardinality estimators [12]. By inserting each sampled key into the HyperLogLog counter, NetKV can track the total unique keys, K, which in turn lets it estimate the ratio of hot and cold keys as described below. The Detector also uses a per-thread request counter and the sampling rate to estimate the total number of requests, F.

While the Lossy and HyperLogLog data structures are efficient both in space and time complexity, it still can be expensive to add every key into the counters when processing millions of keys per second. To reduce this cost, we use sampling to only send a fraction of the incoming keys to the Key Detector. While this may make NetKV underestimate the number of unique cold keys, it does not have a significant impact on the estimated hot key load distribution. The missed cold keys will be treated as a single "virtual cold key", which causes NetKV's estimate to be higher than the actual load. Since we seek to find an upper bound on the load, this is not a problem.

B. Hot Key Replication

Given the list of key/frequency pairs, (k_i, f_i), from the Lossy Counter, NetKV decides which keys to replicate by

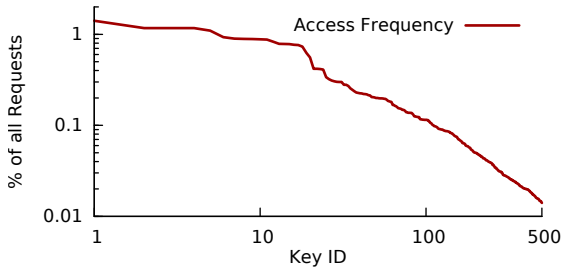

Fig. 4. Access frequency distribution of wikipedia trace

exploiting the shape of skewed probability distributions typically found in web workloads. Figure 4 shows the number of requests made to the most popular URLs (i.e., keys) in a trace of 30 million requests from Wikipedia. As expected, this shows a highly skewed distribution; the top 25 keys make up 22% of the entire trace, and 3.2 million keys are never accessed more than ten times. Thus only a small fraction of the total keys will need to be replicated.

NetKV uses a replication threshold, T, to determine what keys to replicate. Any key with estimated frequency $f_i > T$ is considered a hot key that will be replicated by the algorithm. The replication factor of each key is set to $r = ceil(f_i/T)$. It should be noted that this approach permits a key to have more replicas than there are servers in the cluster (if T is set low or the frequency of some key is very high). However, each key will only be stored in memory at most once per server, and we simply use the large number of "virtual replicas" to evenly distribute the requests across servers.

Adjusting T determines both the number of keys that will be replicated and their replication factor, thus it has a large impact on how evenly balanced the servers will be. Rather than require administrators to set T manually, NetKV automatically calculates T based on the acceptable level of load imbalance in the cluster, as described below.

C. Replica Distribution

Once hot keys have been selected, NetKV must determine where to place the replicas. In our original prototype, we used a lookup table that tracked each hot key, its replication factor, and the server it was assigned to. However, intelligently deciding which servers to place replicas on requires significant information on the current load of each server, as well as an understanding of how much of that load comes from hot or cold keys. Optimally solving this reduces to the bin packing problem, which becomes intractable for large numbers of keys and servers.

To avoid this complexity, NetKV uses randomization to evenly distribute hot key replicas across the cluster. To replicate hot key, k_i, NetKV must select $r = ceil(f_i/T)$ servers. The first server is selected by doing a consistent hash ring lookup for the renamed key "k_i–replica-1"; the second server by looking up "k_i–replica-2", and so on. This allows NetKV to produce r different "virtual key names" for k_i, which are evenly distributed around the consistent hash ring. When the replication factor for a key is changed or a new `set` for a replicated key arrives, NetKV will issue requests to all r servers to insert the key and value. Note that the virtual key name is used only to select the appropriate server—the original key name is still used in the actual request.

While this does not prevent several hot keys from being sent to the same server, in practice when there are a reasonable number of virtual hot keys and servers, we find that the load is evenly distributed. This is achieved with far less communication, storage overhead, and lookup cost compared to an approach that individually tracks server loads and does bin packing to optimally place keys.

D. Load Estimation

The above algorithms assume a predefined replication threshold, T; we now discuss how NetKV automatically adapts T based on the predicted level of load imbalance derived using a balls and bins analysis [13]. This allows NetKV to estimate the maximum load a server in the cluster will experience for a proposed T value.

We consider the memcached servers as a set of n bins, and the unique keys being requested as a set of m balls that must be assigned to them. Intuitively, if we use consistent hashing to randomly distribute all the balls across the bins, the average balls per bin will be $\frac{m}{n}$, but it is likely that some bins will have more balls than that. The analysis originally by Mitzenmacher lets us estimate with high probability the most balls (i.e., unique keys) that will be assigned to any one bin (server). Mitzenmacher's results for the case where $m = n$ have since been extended to give a set of equations that can be used depending on the relationship between the number of balls and bins [13], [14]:

$$MaxBalls(m,n) =$$

$$\begin{cases} \frac{log\, n}{log\, \frac{n}{m}} & : m < \frac{n}{log\, n} \\ \frac{log\, n}{log\, \frac{nlog\, n}{m}}(1 + \alpha\frac{log^{(2)}\frac{nlog\, n}{m}}{log\, \frac{nlog\, n}{m}}) & : \frac{n}{log(n)} \leq m < nlog\, n \\ \frac{m}{n} + \alpha\sqrt{2\frac{m}{n}log\, n} & : m > nlog\, n \end{cases} \quad (1)$$

This allows us to calculate the unique keys assigned to each server, but since all keys may have different request rates (particularly in a skewed web distribution), it is difficult to estimate the actual load that a server will see.

To solve this problem, we take advantage of NetKV's replication algorithm that will replicate keys $k_1 ... k_h$, with frequency f_i greater than T each f_i/T times. Thus key renaming in effect transforms the original h hot keys into a set of h^* "virtual hot keys" each with frequency T:

$$h^* = \sum_{i=1}^{h} \frac{f_i}{T} \quad (2)$$

Thus h^* can be used as the number of balls being assigned to the bins. Since each virtual hot key will have a maximum frequency of T, we can estimate the maximum load caused by the replicated hot keys as:

$$MaxHotLoad = MaxBalls(h^*, n) \times T \quad (3)$$

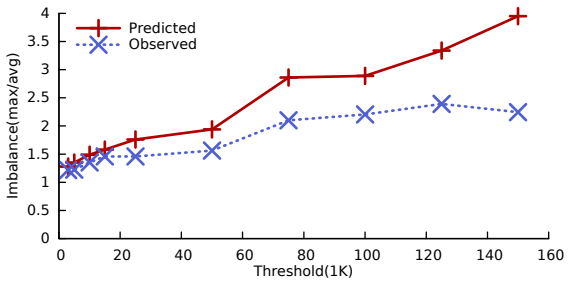

Fig. 5. Load imbalance versus threshold

Note that since there typically are relatively few hot keys, one of the first two ball and bin analysis equations will generally be used.

Next we must estimate the load incurred by the cold keys. While the lossy counter provides frequency information for all of the hot keys (and some of the cold ones), we do not know exact frequencies of all keys, nor the exact total number of unique keys. However, we do know the total number of requests, F, and an estimate of the number of unique keys, K measured by the Hot Key Detector's counters. Thus we know that the $c = K - h$ cold keys have a total request frequency:

$$F_C = F - \sum_{i=1}^{h} f_i \tag{4}$$

While some cold keys are more popular than others, typically $c >> n$, meaning that there is a very large number of cold keys assigned to each server. As a result, each server is likely to get a relatively even mix of popular and unpopular cold keys. Thus we estimate the maximum load caused by the cold keys as:

$$MaxColdLoad = MaxBalls(c, n) \times \frac{F_C}{c} \tag{5}$$

Finally, we can conservatively estimate the maximum load on any server as the sum $MaxColdLoad + MaxHotLoad$.

To evaluate the accuracy of our analysis, we estimate the load imbalance (i.e., $maxLoad/avgLoad$) for the 28 million requests in the wikipedia trace when varying the replication threshold T and then compare it to the observed imbalance when running the trace through our NetKV simulation platform. The results in Figure 5 show that our estimates track the simulated values relatively closely. As expected, our estimate is consistently higher than the observed imbalance, since it is based on the ball and bin analysis that gives an upper bound.

E. Adaptive Replication

At the end of every measurement interval, NetKV uses the information from the Hot Key Detector to calculate the expected load imbalance with the current value of T. If due to recent workload changes the predicted value exceeds an administrator set bound (e.g., the most loaded server should see at most 50% more requests than average), NetKV will automatically search for a new T value that will produce a sufficiently balanced allocation. By doing a parameter sweep through possible T values, the proceeding analysis can be used to find a replication threshold that will meet the administrator specified load imbalance.

F. Handling Replicated Writes

NetKV offers two approaches for providing consistent updates to hot keys.

Write Ordering: Naively replicating keys and forwarding requests to them can easily cause consistency issues. For example, two dispatcher threads might receive write requests for the same key, but with different value content. If these two requests are forwarded to the replication servers in an interleaved order, the replica servers will see inconsistent content for the same key.

To reduce the likelihood of such conflicts, NetKV delegates all hot key writes to a single thread.[1] This thread then processes each write sequentially, forwarding the write to all replicated servers.

Local Cache: Often, very hot content is read frequently but not written to. As a result, there may only be a small number of keys which are both read and written to frequently enough to require replication. The write rate for these keys may exceed the capacity of a single server, so NetKV can use its fast local cache to store them. When a hot key receives a `set` request, NetKV updates the replication table to instead direct requests to its local cache. This cache is capable of serving millions of requests per second, far beyond the rate of a normal server, and since the key is not replicated there are no ordering concerns. Any additional space in the local cache can be used for hot read-only keys.

G. From Hot to Cold

Memory is an expensive and limited resource on both the NetKV proxy and memcached servers. NetKV has to evict items to free space for new hot keys when its local cache is full, and remove the unnecessary replicas throughout the cluster when hot keys become cold.

Once its cache is full, a memcached server uses the LRU (Least Recently Used) eviction policy, which may cause it to automatically evict a replica of a formerly-hot key. Rather than relying on this indirect form of eviction, NetKV tracks which keys change from hot to cold between each measurement interval. It stores two hot replication key tables: one for the previous interval and one for the current. If a key only showed up in the previous replication table, or if its replication factor decreases, the NetKV writer thread will send a delete request to the local cache or replicated servers to free space. Since NetKV uses consistent hashing to determine where a key is replicated, adjusting the number of replicas does not require any data movement, just invalidation requests.

While LRU is effective for tracking key popularity, it is quite expensive. NetKV uses a lighter weight LFU (Least Frequently Used) policy to evict items from its local cache when there is insufficient space. This keeps the local cache as simple as possible, improving its maximum throughput.

[1]If the hot key write rate exceeds the capacity of one thread, then the key space can be partitioned with one thread handling each partition.

V. NetKV Implementations

NFV Prototype: We have implemented our prototype on top of DPDK 1.4.1, a library that allows direct access to packet data from user space applications for efficient I/O. Bypassing the kernel reduces many overheads, but also means our load balancer cannot use the kernel's TCP stack. While memcached supports TCP and UDP, many large-scale memcached deployments such as Facebook's rely on UDP for all get requests to reduce overheads [1]. Currently NetKV only supports UDP requests, but could incorporate a user space TCP stack such as MTCP if desired [15].

Our prototype runs a configurable number of dispatcher threads that handle incoming packets. Traffic from the NIC is evenly distributed across dispatcher threads using RSS (receive-side scaling) to manage the mapping between RX queues and dispatcher threads by hashing the packet header 5-tuple. NetKV also runs a key detector thread that implements the lossy counter, a replication thread that runs the replication algorithm, and a writer thread that serializes set requests. The writer thread can also be used to evict former hot key replicas and to setup new key replicas when needed. Communication between all threads is achieved with lockless message buffers to prevent synchronization overheads.

Trace-Driven Simulator: We also implement our algorithms in a Java-based simulator so we can evaluate their performance at larger scale. The simulator can either generate a set of requests following a Zipf distribution or it can take a trace of requests. We use a trace of 28 million URL requests to the popular wikipedia website; while these are not identical to memcached requests we expect that they will follow a similar distribution.

VI. Evaluation

Our goals for the evaluation are to see the overheads and scalability of our NetKV prototype, the performance improvement in real and simulated environments, and the hot key detection accuracy.

A. Environmental Setup

In our experimental setup, we use 12 servers with dual Xeon X5650 @ 2.67GHz CPUs (2x6 cores), 384KB of L1 cache, 1536KB of L2 cache per core, and a shared 12MB L3 cache. Each server has an Intel 82599EB 10G Dual Port NIC (with only one port used for our experiments) and 48GB memory. We deploy the NetKV load balancer on one server, memcached servers run on eight servers f(each is deployed with only one worker thread), and the remaining servers act as clients. Servers run Ubuntu 14.04 and Memcached 1.4.22.

We use the Facebook mcblaster benchmark as the memcached client in our experiments to measure latency. We use UDP requests in all cases except when running TwemProxy because it only supports TCP. We have found that memcached requests over TCP and UDP have similar latency at low and moderate load. Since the maximum request rate of mcblaster is less than the throughput NetKV can support, we modify Pktgen-DPDK 2.7.7 (a high speed packet generator) to support memcached's ASCII protocol. The generator creates different skewed workloads according to Zipfian access pattern with different Zipf parameters.

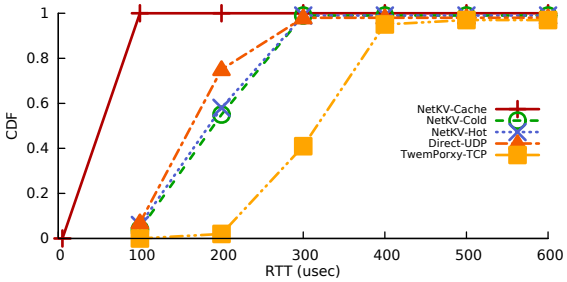

Fig. 6. Get Requests CDF for RTT

B. NFV Load Balancer Overhead

Latency: Memcached is often used in web applications to cache the result of expensive database queries, so load balancer should perform the requests as quickly as possible. Thus the overhead of the load balancer becomes very important. We compare four approaches to forwarding requests: "NetKV-Cold" and "NetKV-Hot" respectively represent requests to NetKV that only need a consistent hash lookup or also require a replication table lookup to perform virtual key renaming. The "NetKV-Cache" case is for a local cache hit within the NetKV load balancer. "TwemProxy-TCP" uses TwemProxy from Twitter as the load balancer, and "Direct-UDP" is a baseline where the client issues UDP requests directly to the memcached server.

Figure 6 shows the cumulative distribution function (CDF) of the request latency under light load. We can see that NetKV-hot and NetKV-cold have very similar latency to directly contacting a memcached server (a 20 us average difference), which means that our replication selection module for hot keys and consistent hashing module for normal keys have little overhead. This illustrates the benefit of building NetKV on a high performance NFV platform that minimizes packet processing costs.

When NetKV has a cache hit, we further speed up the response time, reducing the average latency to 21 us. In contrast to NetKV, TwemProxy has much higher overhead (average latency 315 us). This is because it uses interrupt driven packet processing, has memory copies and context switch between user space and kernel space, and TwemProxy must mediate all responses returned to the client.

Throughput: We next measure the maximum throughput[2] achieved by NetKV (without a local cache) and TwemProxy when adjusting the size of keys in requests and values in responses. Adjusting the key size has no impact on TwemProxy, since its bottlenecks are not related to the packet size, and it has a maximum throughput of 90,000 requests per second. As shown in Table I, the performance of NetKV varies based on the key size. For all but the smallest key size, NetKV is limited

[2]To determine the maximum possible forwarding rate of NetKV we use PktGen to send memcached traffic. The load balancer then returns each packet directly back to PktGen after performing the necessary lookups to determine how the request would be forwarded. This approach is necessary since we do not have enough memcached servers in our cluster to handle the maximum rate of NetKV. We find the maximum throughput with less than 1% drop rate.

	Max Tput (Reqs/sec)
TwemProxy (≤196 byte key)	90,000
NetKV (4 byte key)	10,474,702
NetKV (78 byte key)	6,019,425
NetKV (196 byte key)	4,698,778

TABLE I
MAX THROUGHPUT OF DIFFERENT KEY SIZES

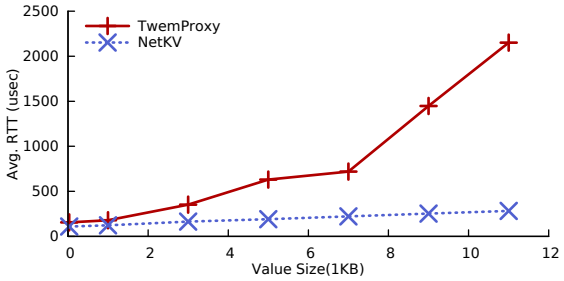

Fig. 7. Packet size vs Average RTT

by the 10Gbps line rate, not the packet processing overheads. For 4 byte keys, NetKV achieves 115 times greater throughput than TwemProxy, demonstrating the benefits of using polling and zero-copy I/O in the NFV platform.

NetKV is designed so that responses from the server are sent directly to the client, avoiding the load balancer, so its performance is independent of the value size. In contrast, TwemProxy acts as an intermediary for all requests, and incurs expensive memory copies of all packet data. Figure 7 shows the average response time with value sizes from 32 bytes to 11KB. This clearly shows that TwemProxy brings greater overhead because of its need to copy response data from the server socket to the client socket. When the value size is 11KB, it quickly goes up to 2151 us. As expected, NetKV is not affected by the value size, since all responses directly return to the clients from the memcached servers.

Scalability: Finally, we investigate how NetKV's performance changes as we adjust the number of Dispatcher threads, each of which is dedicated a CPU core. Figure 8 shows the throughput for different types of requests when using 31 byte keys (which Facebook reports is the size for over 90% of their application's keys [16]). The "Forwarding" line illustrates the baseline throughput achieved by a simple packet forwarder NF that only performs address rewriting, but does not do the hot key detection, consistent hash table lookups, etc. required by NetKV. On our platform, it takes two threads running the forwarding function to meet the 10Gbps line rate.

The "NetKV" line shows the maximum load balancer throughput when all requests are for cold keys (performance is similar for hot keys). In this case, three threads are almost sufficient to meet the line rate. NetKV incurs the greatest cost when it must produce a response for each request from its local cache. Then it can take up to five threads, as shown by the "NetKV-Cache" lines; returning 500 byte value sizes is slightly more expensive than 32 byte values since the data must be copied into the packet sent back to the client.

C. NFV Load Balancing Performance

We now demonstrate NetKV's load balancing potential when deployed in front of a cluster of twelve servers. We

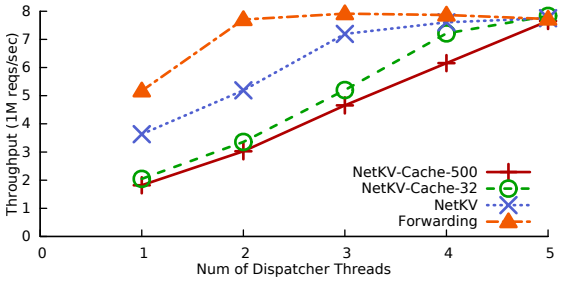

Fig. 8. Througput vs. Number of Threads

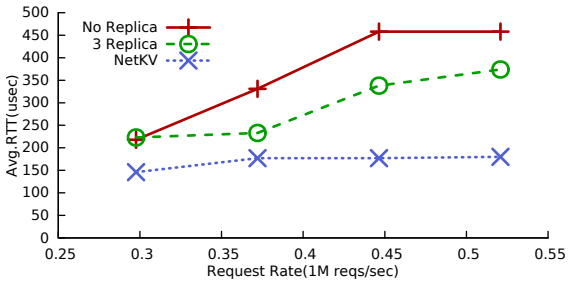

Fig. 9. Average round trip time of the most loaded server versus request rate

compare the performance with no replication, a simple triple replication system that triplicates all keys tracked by the Lossy Counter, and the full NetKV replication system. We use PktGen to drive varying levels of load against the cluster; however, since it can only measure throughput, we also run one mcblaster client for each server to gather response time and drop rate information.

Figure 9 and Figure 10 respectively show the latency and packet drop rate of the most loaded server under zipfan skew 3.0 workload. Without any replication, latency quickly rises to 458us. The load across the servers has very large variance: the load of the most loaded server is 77.82 times greater than the least loaded server and 3.36 times the average load. Even with three replicas of all potential hot objects, the latency still rises, although the more even load distribution prevents an excessive drop rate. NetKV adaptively adjusts the number of replicas based on the workload volume, and thus maintains a smaller latency and avoids packet drops, which make the load evenly distributed. The load of most loaded server compared to the least loaded server and the average load respectively drop to 1.10 times and 1.05 times.

We next compare the throughput with zipf skewed workloads in Figure 11. Without replication, the maximum throughput is quite limited, since a few servers quickly become overloaded, while others remain idle. Triple replication reduces this problem, but NetKV's adaptive replication algorithm improves the throughput by approximately 20% and maintains a steady throughput even for extremely skewed loads.

D. Impact of Replication Parameters

The replication threshold T determines the number of keys picked to replicate and the number of replications, which both affect the load balance. Here we explore the impact of different settings, and why it is important for NetKV to dynamically adapt T to meet the target load imbalance level.

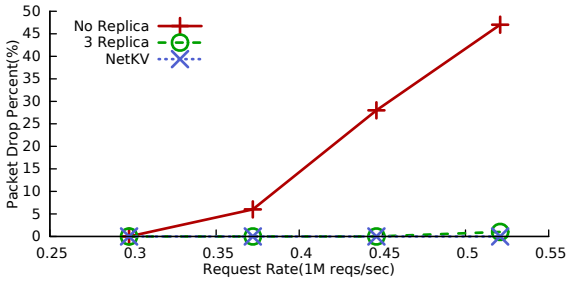

Fig. 10. Packet Drop Percentage of the most loaded server versus request rate

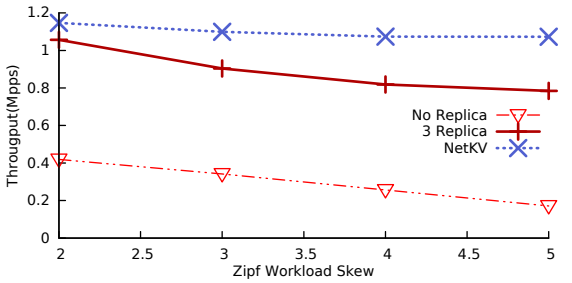

Fig. 11. Maximum Throughput versus different skewed workload

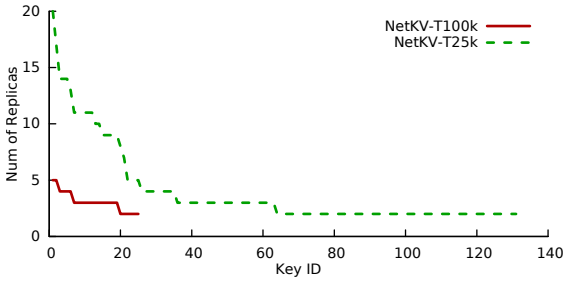

Fig. 12. Number of replicas versus Key ID

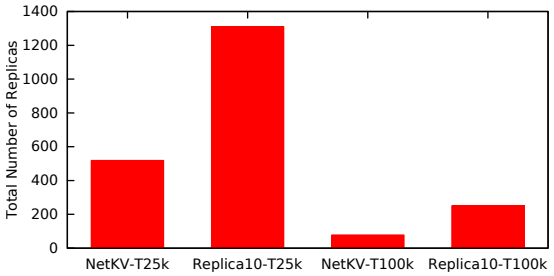

Fig. 13. Total number of replications for different cases

We use our simulation platform with a trace of requests from Wikipedia and 100 servers. Figure 12 shows the number of replicas NetKV makes for the hot keys when T is statically set to 25,000 or 100,000. As expected, using a smaller threshold results in each of the hot keys being replicated more times, but it also causes substantially more keys to be picked for replication (141 versus 25).

Facebook's approach to memcached load balancing relies on a fixed replication level for each pool of keys [1]. However, the skewed popularity seen by web workloads makes a static setting inefficient. Figure 13 shows the total key replicas of the detected hot keys when the threshold is used to determine the set of hot keys and then there is either a fixed replication level per key or NetKV's automatic replication level. Replica10-T25k has 1,310 replications which is 2.5 times larger than NetKV with the same threshold. This wastes memory, since more keys are replicated, but provides minimal improvement to the server load. On the other hand, with $T = 100K$, the fixed replication system uses more memory for replication than NetKV, but neither approach replicates the hottest keys enough times.

The impact of replication factor and threshold on the level of load imbalance is further shown in Figure 14. If there is not any replication, the load on the most loaded server is 5 times larger than the least loaded server. Replica10-T25k partly alleviates the hot spot and drops the imbalance to a factor of 3 times. However, the load imbalance of NetKV-T25k is even better, since it replicates a few of the most popular keys as many as twenty times. By intelligently selecting which keys to replicate based on their estimated frequency, NetKV reduces the replication memory cost by 2.53 times, while simultaneously providing a substantial improvement in load balance.

When the threshold is set to 100K, neither fixed replication nor NetKV perform that well because the hotest keys are not replicated enough times. This shows the danger of setting fixed thresholds or replication factors—unless these parameters are carefully set, memory can be wasted for unneeded replicas, or insufficient keys may be replicated to keep the load evenly balanced.

E. Adaptive Replication Threshold

We now explore NetKV's ability to dynamically set the replication factors to keep the cluster balanced, despite skewed workloads. We first test a synthetic workload where we vary the Zipf skew parameter with a cluster with 100 servers. We use the imbalance factor Eq. 6 to compare the level of imbalance between servers.

$$\sum_{i=1}^{n} \frac{|Rate_i - AvgRate|}{AvgRate * n} \tag{6}$$

Where $Rate_i$ is the request rate on server, i and $AvgRate$ is the average requests across all servers.

Figure 15 shows that as expected, the imbalance factor quickly rises with no replication. While fixed levels of replication reduce the variation, it is difficult to know in advance how to set the replication factor, and even replicating all keys 10 times does not prevent imbalance as the workload skew rises. NetKV maintains a small and stable imbalance factor by automatically adjusting the replication threshold based on the observed workload skew.

Figure 16 lists the load of each server. It is clear that with 10 replications (10 Replica line), the variance of server load is very large. The most loaded server has 216 times load than the least loaded server. While in our "NetKV", each server has similar load.

The imbalance between the most and least loaded servers in a cluster also tends to rise as the number of servers is increased. Thus horizontally scaling a cluster can sometimes lead to worse tail latency values since the average latency is improved more than the maximum. To explore how NetKV can

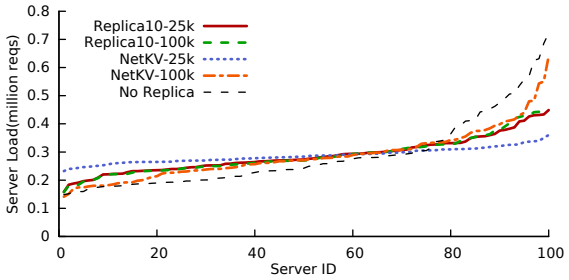

Fig. 14. Server load distribution for wiki workload

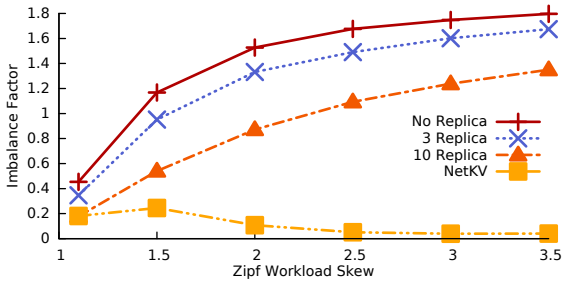

Fig. 15. The average variance of all servers' request rate relative to the average request rate with different skewed workloads

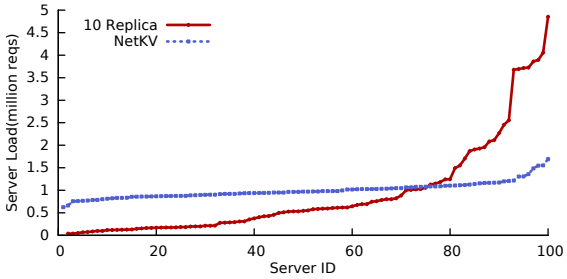

Fig. 16. Server Load under Skewed Workoad

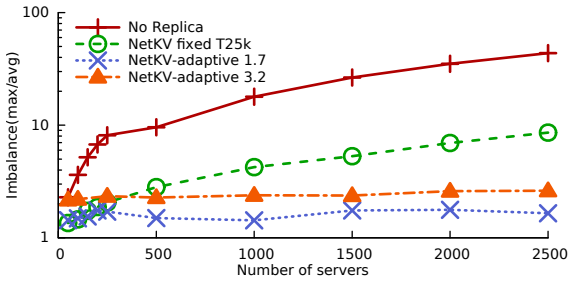

Fig. 17. Load imbalance versus the number of servers

assist with this problem we use our simulator and wikipedia trace and vary the number of servers from 50 to 2,500 to measure the load imbalance (i.e., maximum server request rate divided by average request rate).

Figure 17 shows that if there is no any replication, adding more servers does not alleviate the hot spot since one server still receives each hot key. When using NetKV with a fixed threshold of $T = 25,000$, the load imbalance is initially acceptable, but becomes worse as the number of servers rise. This happens because the average request rate drops linearly with more servers, but when the replication factor is not adjusted, the extra requests caused by hot keys are not rebalanced.

When NetKV's adaptive threshold feature is used, the system automatically retunes the replication factor to meet the acceptable imbalance level. We illustrate target imbalance levels of 1.7 and 3.2 in Figure 17. By utilizing the balls and bins analysis, NetKV can automatically predict how adjusting the number of servers (bins) affects the difference between the most and average loaded servers. NetKV maintains a load imbalance within 16% of the target in all cases.

F. Hot Key Detector Sampling Accuracy

Finally, we evaluate the impact of sampling on the Hot Key Detector's accuracy. Figure 18 shows that sampling rates up to 1 in 1000 have minimal impact on the accuracy of the hot key detector—the estimated frequency of each hot key remains very similar to the real trace data. As a result, we set 1000 as the sampling rate in all our experiments.

VII. Related Work

NFV: Network Function Virtualization facilitates the deployment of network services by running them as software on virtual machines. In recent years, there have been several projects to increase packet performance on commodity servers. Netmap [17] bypasses network stack layer overheads by directly mapping packets from kernel space to user space. Similarly, the Intel DPDK [18] library uses zero-copy packet I/O and a polling-based driver to allow applications to do line rate packet processing. NetKV leverages DPDK for fast access to raw packets, and then provides the network and application-layer processing needed to interpret and forward memcached requests.

In-Memory Key-Value Stores: In standard memcached, the data structures and fine-grained mutex restrict the performance. There are a bunch of work to optimize and improve it. MemC3 [19] uses efficient concurrent cuckoo hashing and CLOCK cache replacement to improve the memcached performance. However, it still suffers from the fine-grained mutex overhead. [20] and [21] pipeline or parallelize request parse, hash calculation, value store access and response formatter to improve the performance depending on FPGA. MICA [22] bypasses OS, which uses high-speed key-value data structures to enable parallel data access, then get good performance. [23] further optimizes MICA's certain inefficiencies to have higher performance. These approaches all focus on maximizing the performance of a single cache node and can be deployed as in-memory key-value servers in our environment. Our work is orthogonal to these work, focusing on managing large scale in-memory key-value cluster.

Load Balancing: Load balancing in clustered key-value stores such as FAWN [8] rely on consistent hashing to redirect the requests across nodes. [24] proposes an adaptive hash space partitioning approach that can dynamically shift hash space boundary across servers to keep them more evenly loaded, although the requests for a popular key still only go to one server. However, as [16] and [3] show, highly skewed workloads are often seen, which can cause individual servers to become hot spots. Facebook [1] has deployed a pre-configured key replication system to help balance the load. The individual client needs to coordinate the mapping of replicated keys to servers, which reduces the agility of the system compared to an

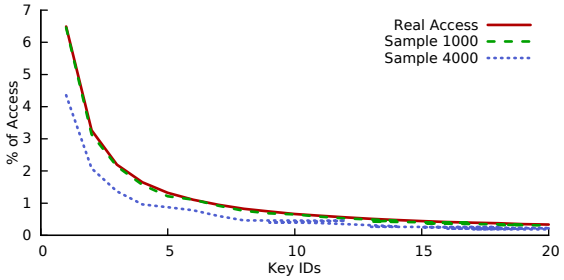

Fig. 18. Sample Accuracy

in-network load balancer like we propose. Hong et al. propose a memcached load balancing system that uses key replication and renaming similar to ours [25]. However, their approach relies on a negotiation protocol between the clients and servers to be aware of the mapping of hot keys and servers, and it does not provide guidelines on how many times they should be replicated. In [26], each key-value server is not standalone and maintains the state of the replicated keys. In NetKV, we seek to both build a scalable memcached load balancing system and provide a complete replication algorithm that determines to replicate keys while maintaining a target level of load balance.

Ball and bin based load analysis [13], [14] has been applied in a range of areas including distributed hash tables [27], [28] and relay allocation [29]. We combine these algorithmic results, recent stream analytic data structures, and advances in NFV software to build a complete solution that efficiently detects workload characteristics and uses that information to guide replication in an actual prototype.

VIII. Conclusions

We have presented a scalable and self-managing load balancer, NetKV. Our approach is based on 1) exploiting recent advances in software-based NFV for efficient packet processing, 2) stream approximation techniques that scalably measure workload characteristics, and 3) balls and bins based load analysis that drives an adaptive replication algorithm. Together, these allow NetKV to efficiently and accurately load balance over 10 million memcached requests per second, while keeping server load levels within an administrator specified bound. NetKV is further optimized with a local cache that can be used for extremely hot data or to eliminate write consistency issues. We have evaluated NetKV both in simulation and as an NFV prototype. Our experimental results show that NetKV adds minimal latency to each request and linearly scales up to line rates of 10 Gbps. NetKV's replication algorithm effectively balances the load, keeping server load imbalance bounded even under highly skewed workloads.

Acknowledgements: This work was supported in part by NSF grants CNS-1253575 and CNS-1422362.

References

[1] 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 memcache at facebook. In *Proc. NSDI*, 2013.

[2] Wei Zhang, Timothy Wood, K.K. Ramakrishnan, and Jinho Hwang. Smartswitch: Blurring the line between network infrastructure & cloud applications. In *HotCloud*, Philadelphia, PA, June 2014.

[3] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 2014.

[4] Wei Zhang, Jinho Hwang, Timothy Wood, K.K. Ramakrishnan, and Howie Huang. Load balancing of heterogeneous workloads in memcached clusters. In *Feedback Computing*, Philadelphia, PA, June 2014.

[5] Graham Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, March 2005.

[6] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, 2002.

[7] Intel Corporation. Intel data plane development kit: Getting started guide. 2013.

[8] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *SOSP*, 2009.

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

[10] Memcached Router. https://github.com/facebook/mcrouter, 2014.

[11] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. NSDI*, 2015.

[12] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, (1), 2008.

[13] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems*, 2001.

[14] Martin Raab and Angelika Steger. balls into binsa simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[15] 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 *Proc. NSDI*, 2014.

[16] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In Proc. SIGMETRICS. ACM, 2012.

[17] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *Proc. ATC*, 2012.

[18] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *In Proc. NSDI*, April 2014.

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

[20] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *In Proc. 40th ISCA*, 2013.

[21] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *HotCloud*, 2013.

[22] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. NSDI*, 2014.

[23] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. ISCA, 2015.

[24] Jinho Hwang and Timothy Wood. Adaptive performance-aware distributed memory caching. In *Proc. ICAC*, 2013.

[25] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013.

[26] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. EuroSys*, 2015.

[27] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*. 2003.

[28] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proc. 2nd ACM Symposium on Cloud Computing*, October 2011.

[29] H.X. Nguyen, D.R. Figueiredo, M. Grossglauser, and P. Thiran. Balanced relay allocation on heterogeneous unstructured overlays. In *Proc. INFOCOM*, 2008.