1-1-2014

Building A Scalable And High-Performance Key-Value Store System

Yuehai Xu
Wayne State University,

Recommended Citation

Xu, Yuehai, "Building A Scalable And High-Performance Key-Value Store System" (2014). *Wayne State University Dissertations.* Paper 1060.

BUILDING A SCALABLE AND HIGH-PERFORMANCE KEY-VALUE STORE SYSTEM

by

YUEHAI XU

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2014

MAJOR: COMPUTER ENGINEERING

Approved by:

____

Advisor Date

____

____

____

ACKNOWLEDGEMENTS

One of my most major life decisions was to pursue a PhD program in the US. Without the help of so many people in various ways, this dissertation would never have been written. First and for formost, I would like to express deepest grateful to my advisor, Dr. Song Jiang, for his invaluable guidance, support and encouragement via this work. His critical thinking and spirit of innovation are always my irreplaceable assets that guide me in my daily life and work.

I would like to thank Dr. Eitan Frachtenberg, my mentor when I was an intern at Facebook. I still feel excited for the work we have done, that delivered meaningful impaction on production systems. In addition, A thank you to Berk Atikoglu and Mike Paleczny who introduced me the Memcached system, and shared me with their system knowledge.

I would also like to thank Dr. Cheng-Zhong Xu, Dr. Weisong Shi, and Dr. Nabil J. Sarhan for being my committee members and their insightful suggestions on this work.

Thank you to all of my dear friends with whom I shared interesting ideas, and sleepless nights of hacking. They are Xuechen Zhang, Yizhe Wang, Jia Rao, Kun Wang, Jianqiang Ou, Xingbo Wu, Wenbo Qiao, Hao Zhou and so many others.

Finally, this thesis would not be possible without the unconditional support from my parents Pingxiu Liang and Jisheng Xu. And my wife, Dan Ao. They are forever my driving force that propels me forward.

TABLE OF CONTENTS

Acknowledgements ii

List of Figures vi

List of Tables viii

Chapter 1 Introduction 1

1.1 Motivation 2

1.1.1 Demand on Study of Workloads' Characteristics 2

1.1.2 KV Cache: A CPU-demanding Application 3

1.2 Thesis Contributions 8

1.2.1 Characterizing Facebook's *Memcached* Workload 8

1.2.2 *Hippos*: A Light-weight and Energy-efficient Appliance 9

1.3 Dissertation Organization 10

Chapter 2 Characterizing Facebook's *Memcached* Workload 12

2.1 Introduction 12

2.1.1 Software Architecture 13

2.1.2 Deployment 14

2.2 Request Rates and Composition 15

2.2.1 Request Rates 15

2.2.2 Request Size 16

2.2.3 Request Composition 17

2.2.4 Discussion 19

2.3 Cache Effectiveness 20

2.3.1 Sources of Misses 21

2.3.2 Temporal Locality Measures 23

2.4 Related Work 26

2.4.1 Cache Replacement Policies 26

2.4.2 Web Cache 27

2.4.3 KV Stores 27

Chapter 3 Building a Key-value Cache to be Energy-efficient 29

3.1 Introduction 29

3.2 The Design of *Hippos* 32

3.2.1 Targeted Workloads 32

3.2.2 Locating the Position to Hook *Hippos* in 33

3.2.3 Removal of the Second Bottleneck 38

3.2.4 Handling TCP packets 40

3.2.5 Distribution of workload among cores 41

3.2.6 Reuse of *sk_buff* 42

3.3 Effectiveness of *Hippos* 42

3.3.1 Identifying Peak Throughput 44

3.3.2 Reducing Memory Operations 45

3.3.3 Mixing GETs with SETs 47

3.3.4 Replaying Facebook's Traces 48

3.4 Related Work 51

3.4.1 Optimization of *Memcached* 52

3.4.2 Moving Applications into the kernel 53

3.4.3 Making network resources accessible at the user level 53

3.4.4 *Netfilter* hooks 54

3.4.5 Reuse of sk_buff 55

Chapter 4 Conclusions and Future Work 56

4.1 Conclusions 56

4.2 Future Directions 58

References 60

Abstract 69

Autobiographical Statement 71

LIST OF FIGURES

Figure 1.1 Peak throughput of *Memcached*s in terms of requests per second with different number of enabled cores. In the figure, *Stock Memcached* refers to the open-source *Memcached* running as an application on Linux; *Multiport Memcached* refers to the optimized *Memcached* with multiport support. *Hippos* refers to the proposed in-kernel KV-cache implementation. 5

Figure 1.2 CPU time distribution on user-level code, kernel (system) code and being idle when one of three *Memcached*' implementations runs with various number of cores at their respective peak throughout. 6

Figure 2.1 (a) Request rates at different days (USR) and (b) times of day (ETC, Coordinated Universal Time—UTC). Each data point counts the total number of requests in the preceding second. 16

Figure 2.2 Key and value size distributions for all traces. The leftmost cumulative distribution function (CDF) shows the sizes of keys, up to *Memcached*'s limit of 250 B (not shown). The center plot similarly shows how value sizes distribute. The rightmost CDF aggregates value sizes by the total amount of data they use in the cache, so for example, values under 320 B or so in SM use virtually no space in the cache; 320 B values weigh around 8% of the data, and values close to 500 B take up nearly 80% of the entire cache's allocation for values. 18

Figure 2.3 Distribution of request types per pool, over exactly 7 days. UPDATE commands aggregate all non-DELETE writing operations. 18

Figure 2.4 Distribution of cache miss causes per pool. 22

Figure 2.5 CDF of key appearances, depicting how many keys account for how many requests, in relative terms. Keys are sorted from least popular to most popular. 24

Figure 2.6 Reuse period histogram per pool. Each hour-long bin n counts keys that were first requested n hours after their latest appearance. Keys can add counts to multiple bins if they occur more than twice. 25

Figure 3.1 The paths for a UDP GET request to travel in the network stack and *Memcached* (or respectively *Hippos*). 34

Figure 3.2 Latency observed at various observation positions in the network stack with different request arrival rates and one core in use. The latency of a packet is measured as the duration between when it is received (*netif_receive_skb()*) and when it reaches a particular observation position. Note that the Y axis is on the logarithmic scale. 36

Figure 3.3 CPU utilization at various observation positions in the network stack with different request arrival rates when one core is in use. 37

Figure 3.4 *Memcached* throughput (number of GETs processed per second) under various GET request arrival rates. The GET requests arrive either in the one-request-in-a-packet format (GET) or in the multiple-requests-in-a-packet format (MGET). For MGET, the packet arrival rate is $320K$ packets per second, and we change the number of requests in a packet. The lock may be applied ($LOCK$) or not ($NOLOCK$). 38

Figure 3.5 Request latencies for *Memcached* and *Hippos* with increasing request arrival rate in the 1Gbps network (a) and in the 10Gbps network (b). For each system, latencies for a low-cost setup (LOW-COST) are also reported, in which requests are for non-existent keys in a hash table not protected by locks. 44

Figure 3.6 Request latencies for *Hippos* with and without using the *sk_buff* reuse optimization when the request arrival rate increases. . . . 46

Figure 3.7 Latencies for workloads with different mixes of GETs and SETs and different request rates. 46

Figure 3.8 Peak throughput received by *Memcached* and *Hippos* for each of the five Facebbok's traces. The throughput is collected under the condition that the corresponding average request latency does not exceed $1ms$. 50

LIST OF TABLES

Table 2.1 *Memcached* pools sampled (in one cluster), including their typical deployment sizes, read request rates, and average hit rates. The pool names do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names. 14

Table 3.1 Distribution of the CPU cycles in different categories of functions at the user level (first row) and at the kernel level (other rows) during the execution of *Multiport Memcached*. 30

Table 3.2 The observation positions 35

Table 3.3 Distribution of request types in the Facebook traces: GET, UPDATE, and DELETE. SET belongs to the UPDATE category, which also includes REPLACE and other non-DELETE writing operations. 48

Table 3.4 Average request latency and power consumption of *Memcached*, and respective changes made by *Hippos* in percentage for the five traces with the 1Gbps and 10Gbps networks (only 10Gbps explicitly indicated). Latency larger than 1ms is denoted by ”-”. If *Memcached*’s latency is denoted as ”-”, *Hippos*’s counterpart is represented by its actual latency value, instead of a change in percentage. 49

Chapter 1: Introduction

Contemporary large-scale websites have to store and process very large amounts of data. To provide timely service to their users, many Internet products have adopted a simple but effective caching infrastructure atop the conventional databases that store these data, called key-value (KV) stores. Examples include *Voldermort* [57] at LinkedIn, *Cassandra* [27] at Apache, and *Memcached* [1, 52] at Facebook. A common use case for these systems is that they store and supply information that is cheaper or faster to cache than to re-obtain, such as commonly accessed results of database queries or the results of complex computations that require temporary storage and distribution [52]. In a KV cache system, data are organized in ordered (key, value) pairs, in which value is the data that are stored by user and key is the unique identification for user to operate data correspondingly. The KV cache interface usually provides primitives similar to those for a regular hash table, such as insertion (SET), retrieval (GET), and deletion (DEL). Clients use consistent hashing [19] on a key to locate the server that owns the requested data.

As an essential component in a datacenter's infrastructure, the KV cache is carefully designed for low response times, high hit rate, and low power consumption. To be effective, these efforts require a detailed understanding of realistic KV workloads on which the performance of KV caches are highly dependent. This dissertation first presents a workload study of a large-scale KV cache that runs at Facebook, revealing access patterns from various perspectives in detail. Our study also shows that current KV cache implementations grow increasingly CPU-bound, leading to under-utlization of network bandwidth and poor energy efficiency. Based on this study, we propose a high-throughput, low-latency, and energy-efficient KV cache implementation, which moves the KV cache into the operating system's kernel and thus removes most of the overhead associated with network stack and system calls.

In this chapter, we describe the motivation of this dissertation and present its contributions.

1.1 Motivation

Key-value (KV) stores play a critical role of caching in the improvement of service quality and user experience in many large-scale websites [1, 52, 27]. Be a high-throughput distributed cache layer, KV caches have received significant research and industry attention recently [52, 65]. In a KV cache, the data is usually cached in the DRAM memory of a server and is retrieved in response to network requests for it. Often, there are a large number of servers deployed to form a single memory pool, allowing a cache for a large data set with high request rate. One example is Facebook, which uses a very large number of *Memcached* servers supplying many terabytes of memory to the clients over the network [10, 52].

1.1.1 Demand on Study of Workloads' Characteristics

Because many data requests exhibit some form of locality, allowing a popular subset of data to be identified and predicted, a substantial amount of database operations can be replaced by quick in-memory lookups, for significantly reduced response time. To provide this performance boost, KV caches are carefully tuned to minimize response times and maximize the probability of caching request data (or hit rate). But like all caching heuristics, a KV-cache's performance is highly dependent on its workload. It is therefore essential to understand the workload's characteristics in order to understand and improve the cache's design.

In addition, analyzing such workloads can: offer insights into the role and effectiveness of memory-based caching in distributed website infrastructure; expose the underlying patterns of user behavior; and provide difficult-to-obtain data for future studies. But many such workloads are proprietary and hard to access, especially those of very large-scale. Such analyses are therefore rare and the workload characteristics are usually assumed in academic research and system design without substantial support from empirical evidence. This dissertation work aims to provide this support. To this end, we have collected detailed traces from Facebook's *Memcached* [1] deployment, arguably the world's largest. The analysis details many characteristics of the caching workload, it also reveals a number of surprises: a GET/SET ratio of 30:1 that is higher than assumed in the literature; some applications behave more like persistent storage than a cache; and strong locality metrics, such as keys accessed many millions of times a day.

1.1.2 KV Cache: A CPU-demanding Application

KV caches are designed to trade off DRAM capacity for reduced computation time, and are used as a distributed hash table to store *(key, value)* pairs. Intuitively, only minimal computation, or a minimum number of CPU cycles, should be required to look up and possibly modify a hash table datum. In that case, a low-power processor with a few cores, combined with large DRAM memory, could suffice to service a heavy request load with low latency. As such, the acquisition and energy costs of the CPU in a KV-cache server in a cluster specialized for in-memory data caching could be significantly lower than that of a general-purpose cluster [14].

To investigate whether the KV cache is indeed bottlenecked by its CPU, we chose *Memcached* as an experimental representative, as its variants are used in major websites, including Facebook, Twitter, Youtube, and Wikipedia. We used a request pattern similar to what we observed at Facebook, one of the world's largest *Memcached* deployments [10]. As reported in Chapter 2, the ratio of GET to SET requests can be very high, sometimes exceeding 30:1. The key size is typically smaller than 30 Bytes, and more than half of the value sizes can be smaller than 20 Bytes in some traces. Additionally, Our examination of the Facebook traces indicates that GET requests use the faster UDP protocol instead of TCP, consistent with what is reported on optimization efforts on *Memcached* at Facebook [52]. To evaluate CPU usage, we set up eight hosts, each running four *Memcached* clients that continually sent asynchronous UDP GET requests to one *Memcached* server, using 64-Byte request packets on the 1Gbps network. All machines used an Intel 8-core Xeon processor (more system details in Section 3.3). As in the rest of the dissertation, peak throughput is reported as the highest throughput observed while the corresponding mean request latency is kept under $1ms$, where a request's latency is measured by the client as total round-trip time.

We use the latest open-source *Memcached* version [1], which is referred to as *Stock Memcached* hereafter, to investigate whether *Memcached* is CPU demanding and how the CPU cycles are spent. We also made efforts within the application to minimize the chance for *Memcached* to be a CPU-demanding one. We disabled the lock on the hash table [*] and replaced the LRU algorithm with the lock-free CLOCK replacement policy. As it is well known that having multiple threads to access one UDP socket can cause

[*] Because we send only GET requests in this experiment, removal of the lock does not compromise hash table's consistency.

serious socket lock contention [52], possibly rendering the application CPU-bound, we modified *Memcached* so that each of its threads listens exclusively on its own UDP port to alleviate this contention, much like the optimization done at Facebook [52]. This improved *Memcached* is referred to as *Multiport Memcached*, which shares the same benefits as of running multiple *Memcached* instances, each on a separate core and on its exclusive network port [15].

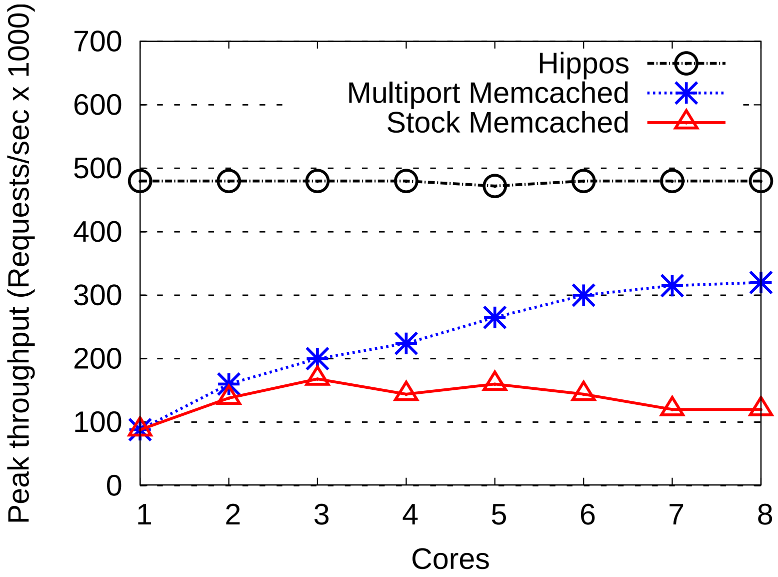


Figure 1.1: Peak throughput of *Memcached*s in terms of requests per second with different number of enabled cores. In the figure, *Stock Memcached* refers to the open-source *Memcached* running as an application on Linux; *Multiport Memcached* refers to the optimized *Memcached* with multiport support. *Hippos* refers to the proposed in-kernel KV-cache implementation.

Figure 1.1 shows measured peak throughput, in terms of number of requests per second, with various number of cores. When the core count increases from one to three, both *Stock Memcached* and *Multiport Memcached* increase their throughput. This suggests that *Memcached*'s performance is probably constrained by the CPU; in other words, *Memcached* requires more CPU cores to unlock its performance potential. When the CPU core count increases beyond three, *Stock Memcached*'s throughput begins to plateau and even drops off due to lock contention within the kernel network stack. In contrast, with multiple sockets *Multiport Memcached* sees its throughput still climbing, albeit at a slower rate. This observation may lead to the conclusion that *Multiport Memcached* is scalable on multicore CPUs without major changes to the kernel [15]. However, this may also demonstrate that the demand on CPU cores does not saturate 1Gbps network card even with all eight cores enabled.

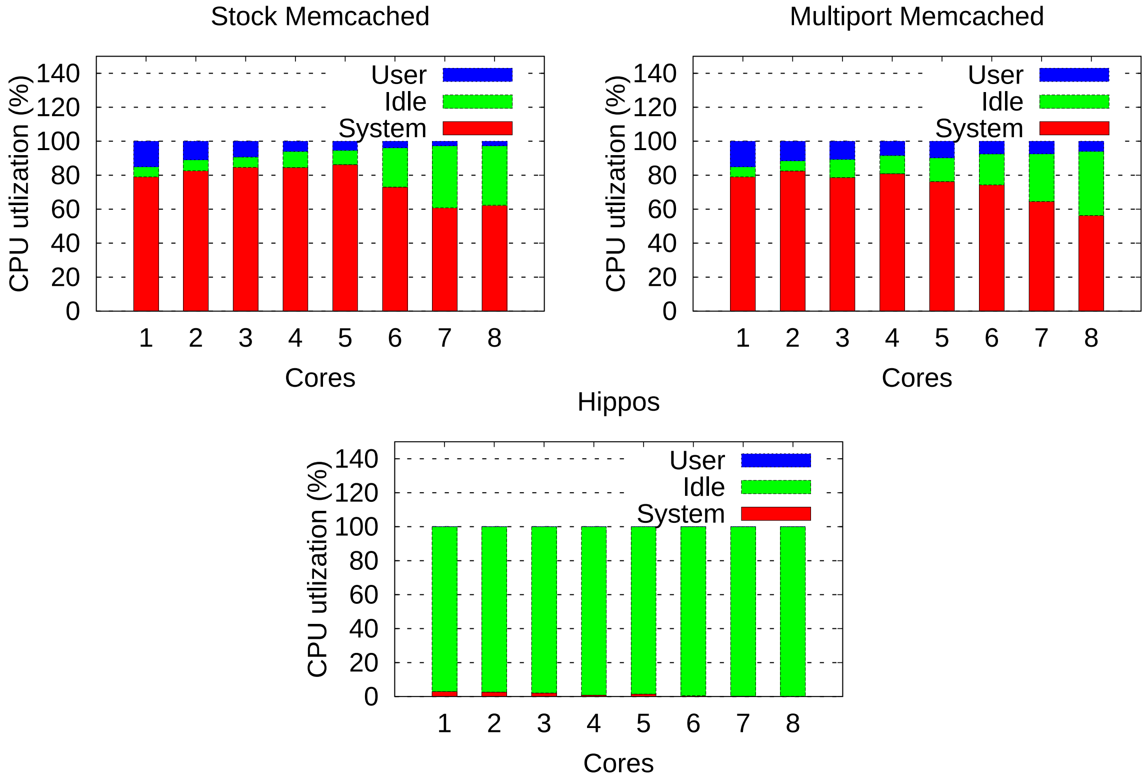


Figure 1.2: CPU time distribution on user-level code, kernel (system) code and being idle when one of three *Memcached*' implementations runs with various number of cores at their respective peak throughout.

Figure 1.2 shows percentages of the CPU cycles that are spent in user-level or kernel-level (system) functions, or when the CPU is idle. We can see that most of CPU time is spent in the kernel for both *Stock Memcached* and *Multiport Memcached*. This is expected as the computation within the application is indeed minimal. With the increase in core count, the user-time percentage for *Stock Memcached* is reduced more significantly than that for *Multiport Memcached*. This is consistent with *Multiport Memcached*'s higher peak throughput at higher core count. Accompanied with the reduction of user time is the increase of idle time. In the experiments for obtaining peak throughput, we did not push the throughput to its limit and allow idle CPU time so that the latency is maintained below the $1ms$ threshold. Figure 1.2 shows that system time accounts for significant percentage of CPU time, from 55% to 85%, depending on core count. This time is mostly spent on the Linux network stack. In Linux, a spinlock is used for exclusive access of the socket buffer queue(s). With only one queue, *Stock Memcached* contends heavily for the lock, resulting in wasted CPU cycles. By having multiple socket queues, fewer CPU cycles are used for spinning, leading to more productive packet processing and higher peak throughput in *Multiport Memcached*.

Considering the percentages of CPU times used in both user and system levels, *Memcached* turns out to be a CPU-demanding application. As such, a KV cache can have increased request latency and limited peak throughput if the CPU is not sufficiently powerful. It is also prone to creating bottlenecks on the request processing path, such as contention on various queue locks in the network stack. Yet another consequence is high power consumption, which can be a critical issue in data centers.

1.2 Thesis Contributions

1.2.1 Characterizing Facebook's *Memcached* Workload

This dissertation discusses five workloads from Facebook's *Memcached* deployment. Aside from the sheer scale of the site and data (over 284 billion requests over a period of 7 sample days), this case study also provides a description of several different usage scenarios for KV caches. This variability serves to explore the relationship between the cache and various data domains: where overall site patterns are adequately handled by a generalized caching infrastructure, and where specialization would help. In addition, this work offers the following key contributions and findings:

1. A workload decomposition of the traces that shows how different applications of *Memcached* can have extreme variations in terms of read/write mix, request sizes and rates, and usage patterns.

2. An analysis of the caching characteristics of the traces and the factors that determine hit rates. We found that different *Memcached* pools can vary significantly in their locality metrics, but surprisingly, the best predictor of hit rates is actually the pool's size.

3. An examination of various performance metricsre over time, showing diurnal and weekly patterns and their correlation to social networking. For example, we found that some load spikes (up to 10% hit rate change) can improve key locality and hit rate if the requested content is limited and static, thus helping to absorb the higher request rate. On the other hand, load spikes on varied keys or keys that are invalidated frequently actually hurt cache performance.

4. An exposition of a *Memcached* deployment that can shed light on real-world, large-scale production usage of KV-stores.

1.2.2 *Hippos*: A Light-weight and Energy-efficient Appliance

A KV cache uses dedicated servers, each configured with large memory and often a low-power processor, to form a large memory pool. It typically runs in a controlled environment (e.g., data centers) and its sole purpose is to provide caching service to other application servers. The objective of this dissertation is to build the KV cache as a data-center appliance with high performance and high energy efficiency. The method is to move it into the kernel in a position close to the NIC, so that it can directly take IP packets for the KV cache and process them *in situ*. Without concern of impacting other applications or any components in the network stack, this approach can remove most time-consuming network operations out of the KV-cache's critical processing path, including acquisition of exclusive access to UDP socket queues, data copies, scheduling and context switching associated with event notification.

We describe *Hippos*, a KV cache that uses a hook provided in the *Netfilter* framework [2] to directly unpack a complete *Memcached* UDP request before it is inserted into its corresponding socket's receive buffer queue. Subsequently, the request is immediately processed and the response is sent back to the device driver. Thus, *Hippos* can provide clients with a single UDP port without even setting up a UDP socket. Accordingly, the overhead for system calls, event notifications (via *libevent*), socket locks, and most of the overheads in the UDP and IP layers are eliminated.

In summary we make the following contributions in this dissertation:

1. We show that a KV cache running at the user level is CPU-demanding, spending significant portion of its processing time in the kernel.

2. We propose *Hippos* to bypass most of the operations for a UDP-based request on its path from the NIC to the user-level *Memcached* and for the corresponding reply request to reach the NIC. With this bypassing, the bottleneck on the network stack is removed. Such removal exposes another bottleneck, namely the one caused by the lock contention within *Memcached*. Accordingly, we applied the Read-Copy-Update (RCU) lock [67] and the lock-free CLOCK cache replacement algorithm in *Hippos* to substantially alleviate the performance impact of this lock contention.
3. We have implemented *Hippos* as a loadable Linux kernel module and extensively evaluated it on a recent Linux Kernel with micro-benchmarks and request traces taken from production systems at Facebook. The results show that *Hippos* can achieve 20–200% throughput improvements on a 1Gbps network (up to 590% improvements on a 10Gbps network) and 5–20% energy saving.
4. This work demonstrates that in the context of improving the performance and energy efficiency of data-center infrastructure, migrating network-intensive applications to the right positions in the kernel and running them as appliances is a viable and promising approach. Many prior projects on migrating applications into the kernel (see Section 3.4) faced challenges such as system security, reliability, and engineering efforts. Nevertheless, our experience shows that in the era of cloud computing, this approach can meet these challenges and gain significant advantages by turning a KV service into an appliance on the network.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows.

In Chapter 2, we begin with describing *Memcached*'s software architecture and its deployment at Facebook. Then we present the workload characteristics in terms of request rates, request size, and request composition. We also discuss cache effectiveness from the perspectives of sources of misses and temporal locality measures. We summarize the related work of workload analysis in the last section.

In Chapter 3, we first overview existing approaches that could improve the performance of *Memcached*. Then we give an extensive examination on CPU cycles consumption of *Memcached* under Facebook's workloads, and expose the appropriate positon in the kernel where we can build a high-performance KV cache system without involving additional modifications to the existing system. We demonstrate design details of *Hippos*, and evaluate it against *Memcached* by using various microbenchmarks and workloads from Facebook's production system.

In Chapter 4, we give our conclusion and future research directions.

Chapter 2: Characterizing Facebook's *Memcached* Workload

2.1 Introduction

Many Web services such as social networks, email, maps, and retailers must store large amounts of data and retrieve specific items on demand very quickly. Facebook, for example, stores basic profile information for each of its users, as well as content they post, individual privacy settings, etc. When a user logs in to Facebook's main page and is presented with a newsfeed of their connections and interests, hundreds or thousands of such data items must be retrieved, aggregated, filtered, ranked, and presented in a very short time. The total amount of potential data to retrieve for all users is so large that it is impractical to store an entire copy locally on each web server that takes user requests. Instead, we must rely on a distributed storage scheme, wherein multiple storage servers are shared among all Web servers

The persistent storage itself takes place in the form of multiple shards and copies of a relational database, such as MySQL. MySQL has been carefully tuned to maximize throughput and lower latency for high loads, but its performance can be limited by the underlying storage layer, typically hard drives or flash. The solution is caching—the selective and temporary storage of a subset of data on faster RAM. Caching works when some items are much more likely to be requested than others. By provisioning enough RAM to cache the desired amount of popular items, we can create a customizable blend of performance and resource tradeoffs.

This Chapter analyzes the workload of *Memcached* at Facebook, one of the world's largest KV deployments. We look at server-side performance, request composition, caching efficacy, and key locality. These observations lead to several design insights and new research directions for KV caches, such as the relative inadequacy of the least-recently-used replacement policy. But first, we describe how a KV cache such as *Memcached* is used in practice.

2.1.1 Software Architecture

Memcached is an open-source software package that exposes data in RAM to clients over the network. As data size grows in the application, more RAM can be added to a server, or more servers can be added to the network. Additional servers generally only communicate with clients. Clients use consistent hashing [19] to select a unique server per *key*, requiring only the knowledge of the total number of servers and their IP addresses. This technique presents the entire aggregate data in the servers as a unified distributed hash table, keeps servers completely independent, and facilitates scaling as data size grows.

Memcached's interface provides the basic primitives that hash tables provide, as well as more complex operations built atop them. The two basic operations are *GET*, to fetch the value of a given key, and *SET* to cache a value for a given key (typically after a previous GET failure, since *Memcached* is used as a look-aside, demand-filled cache). Another common operation for data backed by persistent storage is to *DELETE* a KV pair as a way to invalidate the key if it was modified in persistent storage. To make room for new items after the cache fills up, older items are evicted using the least-recently-used (LRU) algorithm [13].

Pool	**Size**	**GET/s**	Hit Rate	**Description**
USR	few	100, 500	98.2%	user-account status information
APP	dozens	65, 800	92.9%	object metadata of one application
ETC	hundreds	57, 800	81.4%	nonspecific, general-purpose
VAR	dozens	73, 700	93.7%	server-side browser information
SYS	few	7, 200	98.7%	system data on service location

Table 2.1: *Memcached* pools sampled (in one cluster), including their typical deployment sizes, read request rates, and average hit rates. The pool names do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names.

2.1.2 Deployment

Physically, Facebook deploys front-end servers in multiple datacenters, each containing one or more *clusters* of varying sizes. Front-end clusters consist of both Web servers and caching servers, including *Memcached*. These servers are further subdivided based on the concept of *pools*. A pool defines a class of *Memcached* keys. Pools are used to separate the total possible key space into buckets, allowing better efficiency by grouping keys of a single application, with similar access patterns and data requirements. Any given key will be uniquely mapped to a single pool by the key's prefix, which identifies an application.

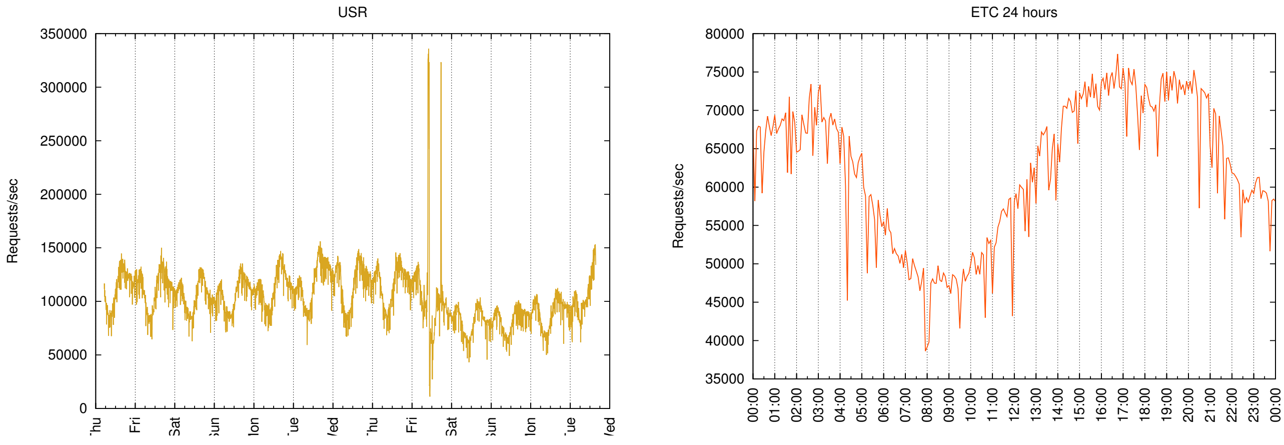
We analyzed one trace each from five separate pools. These pools represent a varied spectrum of application domains and cache usage characteristics (Table 2.1). We traced all *Memcached* packets on these servers using a custom kernel module [10] and collected between $3TB$ to $7TB$ of trace data from each server, representing at least a week's worth of consecutive samples. All five *Memcached* instances ran on identical hardware.

2.2 Request Rates and Composition

2.2.1 Request Rates

Many companies rely on *Memcached* to serve terabytes of data in aggregate every day, over many millions of requests. Average sustained request rates can reach $100,000+$ requests per second, as shown in Table 2.1. These request rates represent relatively modest network bandwidth. But *Memcached*'s performance capacity must accommodate significantly more headroom than mean sustained rates. Figure 2.1(a) shows that in extreme cases for USR, the transient request rate can more than triple the sustained rate. These outliers stem from a variety of sources, including high transient interest in specific events, highly popular keys on individual servers, and operational issues. Consequently, when analyzing *Memcached*'s performance, we focus on sustained end-to-end latency and maximum sustained request rate (while meeting latency constraints), and not on network bandwidth [13].

Figure 2.1 also reveals how *Memcached*'s load varies normally over time. USR's 13-day trace shows a recurring daily pattern, as well as a weekly pattern that exhibits a somewhat lower load approaching the weekend. All other traces exhibit similar daily patterns, but with different values and amplitudes. If we zoom in on one day for ETC for example (righthand figure), we notice that request rates bottom out around 08:00 UTC and have two peaks around 17:00 and 03:00. Although different traces (and sometimes even different days in the same trace) differ in which of the two peaks is higher, the entire period between them, representing the Western Hemisphere daytime, exhibits the highest traffic volume.



(a) (b)

Figure 2.1: (a) Request rates at different days (USR) and (b) times of day (ETC, Coordinated Universal Time—UTC). Each data point counts the total number of requests in the preceding second.

2.2.2 Request Size

Next, we turn our attention to the sizes of keys and values in each pool (Figure 2.2) for SET requests (GET requests have identical sizes for hits, and zero data size for misses). All distributions show strong modalities. For example, over 90% of APP's keys are 31 bytes long, and values sizes around 270 B show up in more than 30% of SET requests. USR is the most extreme: it only has two key size values (16 B and 21 B) and virtually just one value size (2 B). Even in ETC, the most heterogeneous of the pools, requests with 2-, 3-, or 11-byte values add up to 40% of the total requests. On the other hand, it also has a few very large values (around 1MB) that skew the weight distribution (rightmost plot in Figure 2.2), leaving less caching space for smaller values. Small values dominate all workloads, not just in count, but especially in overall weight. Except for ETC, 90% of all *Memcached*'s data space is allocated to values of less than 500 B.

2.2.3 Request Composition

Last, we look at the composition of basic request types that comprise the workload (Figure 2.3) and make the following observations:

USR handles significantly more GET requests than any of the other pools (at an approximately 30 : 1 ratio). GET operations comprise over 99.8% of this pool's workload. One reason for this is that the pool is sized large enough to maximize hit rates, so refreshing values is rarely necessary. These values are also updated at a slower rate than some of the other pools. The overall effect is that USR is used more like RAM-based persistent storage than a cache.

APP has high absolute and relative GET rates too—owing to the popularity of this application. But also has a large number of DELETE operations, which occur when a cached database entry is modified (but not required to be set again in the cache). SET operations occur when the Web servers add a value to the cache. The relatively high number of DELETE operations show that this pool represents database-backed values that are affected by frequent user modifications.

ETC has similar characteristics to APP, but with a higher fraction of DELETE requests (of which not all are currently cached, and therefore miss). ETC is the largest and least specific of the pools, so its workloads might be the most representative to emulate.

VAR is the only pool of the five that is write-dominated. It stores short-term values such as browser-window size for opportunistic latency reduction. As such,

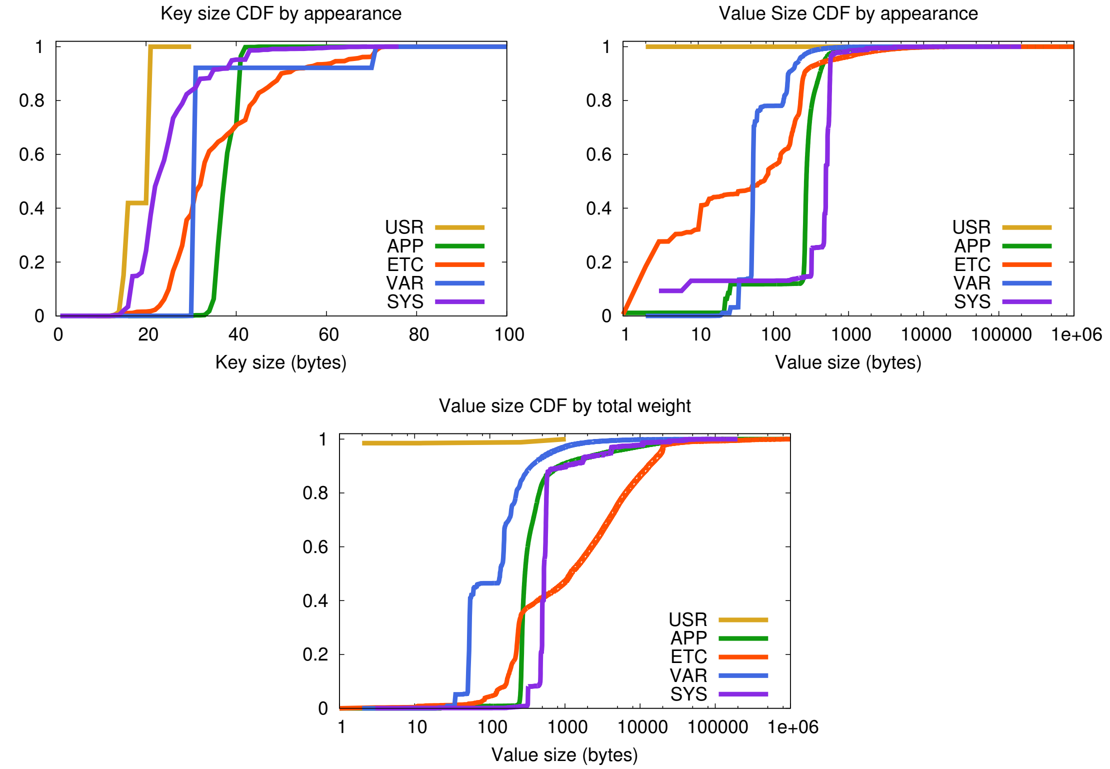


Figure 2.2: Key and value size distributions for all traces. The leftmost cumulative distribution function (CDF) shows the sizes of keys, up to *Memcached*'s limit of $250\,B$ (not shown). The center plot similarly shows how value sizes distribute. The rightmost CDF aggregates value sizes by the total amount of data they use in the cache, so for example, values under $320\,B$ or so in SM use virtually no space in the cache; $320\,B$ values weigh around 8% of the data, and values close to $500\,B$ take up nearly 80% of the entire cache's allocation for values.

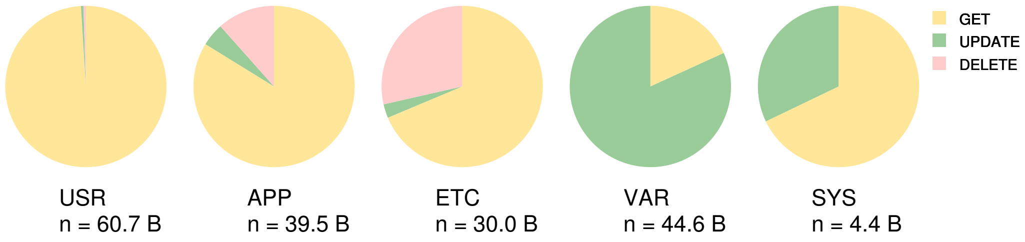


Figure 2.3: Distribution of request types per pool, over exactly 7 days. UPDATE commands aggregate all non-DELETE writing operations.

these values are not backed by a database (hence, no invalidating DELETEs are required). But they change frequently, accounting for the high number of UPDATEs.

SYS is used to locate servers and services, not user data. As such, the number of requests scales with the number of servers, not the number of user requests, which is much larger. This explains why the total number of SYS requests is much smaller than the other pools'.

2.2.4 Discussion

We found that *Memcached* requests exhibit clear modality in request sizes, with a strong bias for small values. We also observed temporal patterns in request rates that are mostly predictable and require low bandwidth, but can still experience very significant outliers of transient high load. There are several implications for cache design and system optimizations from these data:

1. Network overhead in the processing of multiple small packets can be substantial relative to payload, which explains why Facebook coalesces as many requests as possible in as few packets as possible [13].

2. *Memcached* allocates memory for KV values in slabs of fixed size units. The strong modality of each workload implies that different *Memcached* pools can optimize memory allocation by modifying the slab size constants to fit each distribution. In practice, this is an unmanageable and unscalable solution, so instead *Memcached* uses 44 different slab classes with exponentially growing sizes to reduce allocation waste, especially for small sizes. This does, however, result in some memory fragmentation.

3. The ratio of GETs to UPDATEs in ETC can be very high—significantly higher in fact than most synthetic workloads typically assume. For demand-filled caches where each miss is followed by an UPDATE, the ratios of GET to UPDATE operations mentioned above are related to hit rate in general and the relative size of the cache to the data in particular. So in theory, one could justify any synthetic GET to UPDATE mix by controlling the cache size. But in practice, not all caches or keys are demand-filled, and these caches are already sized to fit a real-world workload in a way that successfully trades off hit rates to cost.

These observations on the nature of the cache lead naturally to the next question (and section): how effective is *Memcached* at servicing its GET workload—its *raison d'être.*

2.3 Cache Effectiveness

Understanding cache effectiveness can be broken down to the following questions: how well does *Memcached* service GET requests for the various workloads? What factors affect good cache performance? What characterizes poor cache performance, and what can we do to improve it?

The main metric used in evaluating cache efficacy is hit rate: the percentage of GET requests that return a value. Hit rate is determined by three factors: available storage (which is fixed, in our discussion); the patterns of the underlying workload and their predictability; and how well the cache policies utilize the available space and match these patterns to store items with a high probability of recall. Understanding the sources of misses will then offer insights into why and when the cache wasn't able to predict a future item. We then look deeper into the workload's statistical properties to understand how amenable it is to this prediction in the first place. The overall hit rate of each server, as derived from the traces and verified with *Memcached*'s own statistics, are shown in Table 2.1.

SYS and USR exhibit very high hit rates. Recall from that same table that these are also the smallest pools, so the entire keyspace can be stored with relatively few resources, thus eliminating all space constraints from hit rates. Next down in hit-rate ranking are APP and VAR, which are larger pools, and finally, ETC, the largest pool, also exhibits the lowest hit rate. So can pool size completely explain hit rates? Is there anything we could do to increase hit rates except buy more memory? To answer these questions, we take a deeper dive into workload patterns and composition.

2.3.1 Sources of Misses

To understand hit rate, it is instructive to analyze its complement, miss rate, and specifically to try to understand the sources for cache misses. These sources can tell us if there are any latent hits that can still be exploited, and possibly even how.

We distinguish three types of misses:

- *Compulsory misses* are caused by keys that have never been requested before (or at least not in a very long time). In a demand-filled cache with no prefetching like *Memcached*, no keys populate the cache until they have been requested at least once, so as long as the workload introduce new keys, there is not much we can do about these misses.
- *Invalidation misses* occur when a requested value had been in the cache before, but was removed by a subsequent DELETE request.

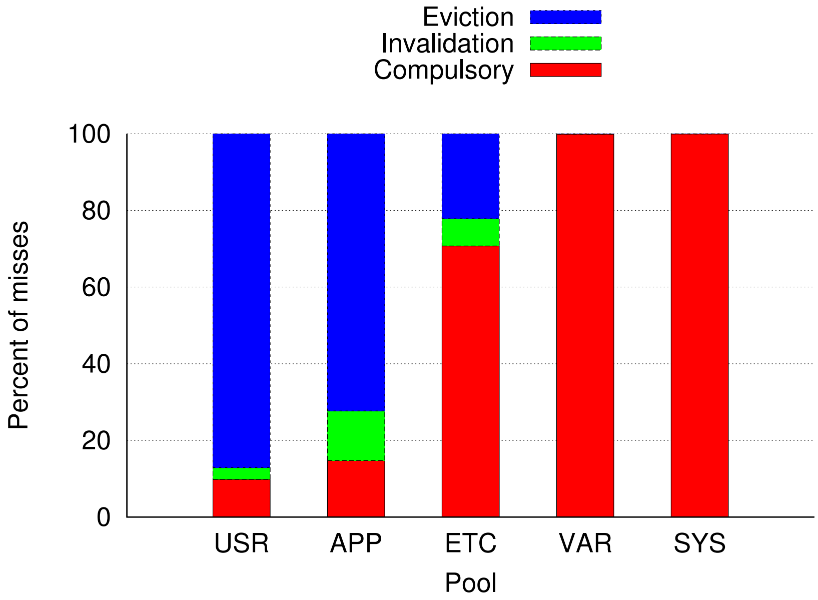


Figure 2.4: Distribution of cache miss causes per pool.

- *Eviction (capacity) misses* represent keys that had been in the cache, but were evicted by the replacement policy before the next access. If most misses are of this kind, then indeed the combination of pool size and storage size can explain hit rates.

Several interesting observations can be made. The first is that VAR and SYS have virtually 100% compulsory misses. Invalidation misses are absent because these pools are not database-backed, and eviction misses are nearly non-existent because of ample space provisioning. Therefore, keys are invariably missed only upon the first request, or when new keys are added.

On the opposite end, about 87% of USR's misses are caused by evictions. This is puzzling, since USR is the smallest of pools, enabling sufficient RAM provisioning to cover the entire key space. This larger percentage of eviction misses originates from service jobs that request sections of the key space with little discernible locality, such as data validation or migration to a new format. So the cache replacement policy has little effect in predicting future key accesses to these keys and preventing eviction misses.

At last we come to ETC and APP, the two largest pools, with 22% and 72% eviction misses, respectively. One straightforward measure to improve hit rates in these two pools would be to increase the total amount of memory in their server pool, permitting fewer keys to be evicted. But this solution obviously costs more money and will help little if the replacement policy continues to accumulate rarely used keys. A better solution would be to improve the replacement policy to keep valuable items longer, and quickly evict items that are less likely to be recalled soon. To understand whether alternative replacement policies would better serve the workload patterns, we next examine these patterns in terms of their key reuse over time, also known as temporal locality.

2.3.2 Temporal Locality Measures

We start by looking at how skewed is the key popularity distribution, measured as a ratio of each key's GET requests from the total (Figure 2.5). All workloads exhibit long-tailed popularity distributions. For example, 50% of ETC's keys (and 40% of APP's) occur in no more than 1% of all requests, meaning they do not repeat many times, while a few popular keys repeat in millions of requests per day. This high concentration of repeating keys is what makes caching economical in the first place. SYS is the exception to the rule, as its values are cached locally by clients, which could explain why some 65% of its keys hardly repeat at all.

We can conceivably use these skewed distributions to improve the replacement policy: By evicting unpopular keys sooner, instead of letting them linger in memory

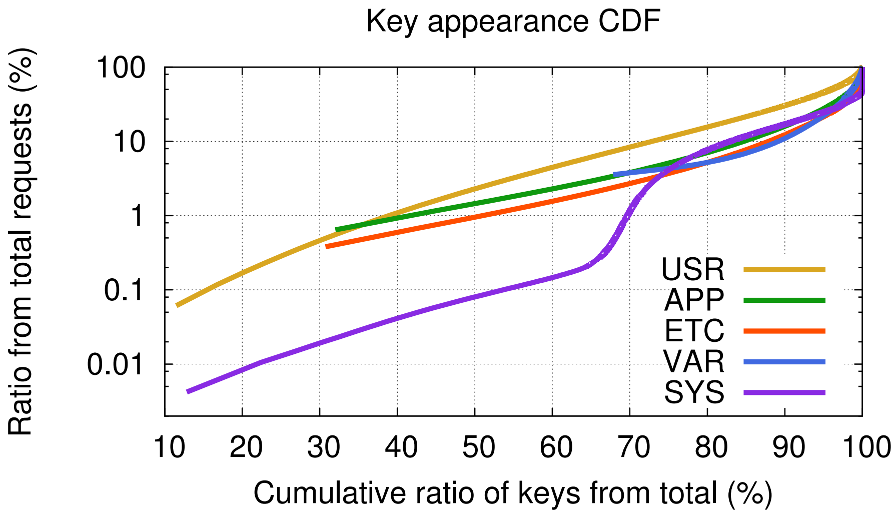


Figure 2.5: CDF of key appearances, depicting how many keys account for how many requests, in relative terms. Keys are sorted from least popular to most popular.

until expired by LRU, a full cycle of insertions later, we could leave room for more popular keys, thus increasing hit rate. For example, about a fifth of all of APP's and ETC's keys are only requested at most once in any given hour. The challenge is telling the two classes of keys apart, when we don't have a-priori knowledge of their popularity.

One clue to key popularity can be measured in reuse period—the time between consecutive accesses to the key. Figure 2.6 counts all key accesses and bins them according to the time duration from the previous access to each key. Unique keys (those that do not repeat at all within the trace period) are excluded from this count. The figure shows that key repeatability is highly localized and bursty, with some daily patterns (likely corresponding to some users always logging in at the same time of

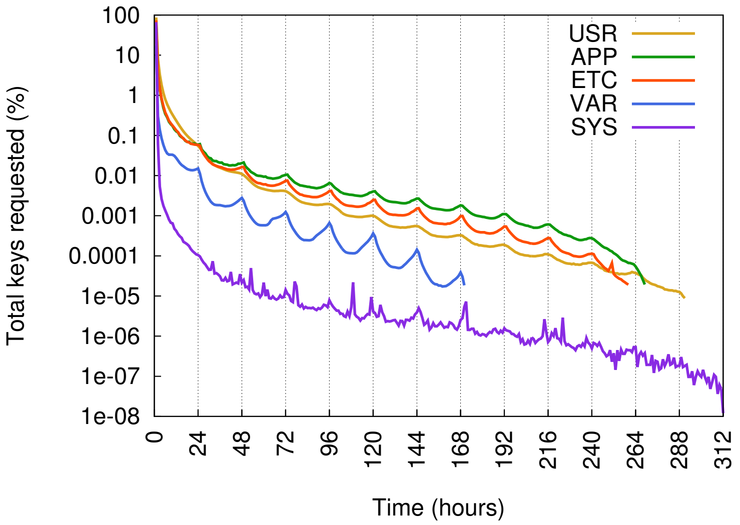


Figure 2.6: Reuse period histogram per pool. Each hour-long bin n counts keys that were first requested n hours after their latest appearance. Keys can add counts to multiple bins if they occur more than twice.

day). For the ETC trace, for example, 88.5% of the keys are reused within an hour, but only 4% more within two, and within six hours 96.4% of all non-unique keys have already repeated. The main takeaway from this chart is that reuse period decays at an exponential rate. This implies diminishing returns to a strategy of increasing memory resources beyond a certain point, because if we can already cache most keys appearing in a given time windows, and double it with twice the memory capacity, only a shrinking fraction of the keys that would have otherwise been evicted would repeat again in the new, larger time window.

As before, the SYS pool stands out. It doesn't show the same 24-hour periodicity as the other pools, because its keys relate to servers and services, not users. It also decays faster than the others. Again, since its data are cached locally by clients, it is likely that most of SYS's GET requests represent data that are newly available, updated, or expired from the client cache; these are then requested by many clients concurrently. This would explain why 99.9% of GET requests are repeated within an hour of the first key access. Later, such keys would be cached locally and accessed rarely, perhaps when a newly added client needs to fill its own cache.

2.4 Related Work

To the best of our knowledge, this is the first details description of a large-scale KV-store workload. Nevertheless, there are a number of related studies on traditional storage system, other caching systems and KV Stores that can shed light on the relevance of this work and its methodology.

2.4.1 Cache Replacement Policies

A core element of any caching system is its replacement algorithm. By analyzing the workloads' locality, source of misses, and request sizes, our original paper [10] suggested areas where an optimized replacement strategy could help. In fact, some of these optimizations have since been reportedly implemented [52], including an adaptive allocator to periodically rebalance the slab allocator, and the use of expiration time associated data items for early eviction of some short-lived keys.

Caching as a general research topic has been extensively studied. The LRU algorithm [68], which is adopted in *Memcached*, has been shown to have several weaknesses, and a number of algorithms have been proposed to improve it. The 2Q algorithm was proposed to evict cold data earlier from the cache so that relatively warm data can stay longer [34]. The LRFU algorithm introduced access frequency into the LRU algorithm, to improve its replacement decisions on data with distinct access frequencies [37]. These weaknesses of LRU also show in this workload study. To address both weaknesses with an efficient implementation, Jiang and Zhang proposed the LIRS algorithm to explicitly use reuse distance—in principle equivalent to the reuse period measured in this dissertation—to quantify locality and choose eviction victims [33]. The LRU algorithm requires a lock to maintain the integrity of its data structure, which can lead to a performance bottleneck in a highly contended environment such as *Memcached*'s. In contrast, the CLOCK algorithm [56] eliminates this need while maintaining similar performance to that of LRU. The CLOCK-Pro algorithm, which also removes this lock, has a performance as good as that LIRS' [32].

2.4.2 Web Cache

Web caches are another area of active research. In a study of requests received by Web servers, Arlitt and Williamson found that 80% of requested documents are smaller than $\approx 10KB$. However, requests to these documents generate only 26% of data bytes retrieved from the server [9]. This finding is consistent with the distribution we describe in [10].

2.4.3 KV Stores

KV stores are also receiving ample attention in the literature, covering aspects such as performance, energy efficiency, and cost effectiveness [13, 69, 29, 21, 22, 8, 11, 6, 20, 25]. Absent well-publicized workload traces, in particular large-scale production traces, many works used hypothetical or synthetic workloads [69]. For example, to evaluate SILT, a KV-cache design that constructs a three-level store hierarchy for storage on flash memory with a memory based index, the authors assumed a workload of 10% SET and 90% GET requests using $20B$ keys and $100B$ values, as well as a workload of 50% SET and 50% GET requests for $64B$ KV pairs [43]. In the evaluation of CLAM, a KV-cache design that places both hash table and data items on flash, the authors used synthetic workloads that generate keys from a random distribution and a number of artificial workload mixes [7].

Chapter 3: Building a Key-value Cache to be Energy-efficient

3.1 Introduction

In section 1.1.2, we presented that *Memcached*, which is one of the most common KV store implementation that has been wildly used in industry, is highly CPU demanding. Figure 1.2 shows that *Memcached* spends most of its time in the kernel, in particular on the network stack. Due to the relevant role it plays, this suboptimal implementation leads to the performance bottlenecks on the request processing path, resulting in high service latency and high power consumption.

To investigate the distribution of CPU consumptions, we examined *Multiport Memcached*[*] with OProfile [5] to see how the CPU cycles are used across the network stack. Table 3.1 shows distribution of the CPU cycles among eight categories of 289 functions, which span all networking layers of the system. Among the functions, the highest percentage of cycles consumed by a single function is 3.89% and there are only 20 functions consuming more than 1% of the cycles. The CPU time is distributed more or less evenly across the user layer, SOCKET layer, UDP layer, IP layer, and ETH and device driver layers. This flat profile defeats any cost-effective attempts to pinpoint specific functions or layers to optimize. In addition, among the function categories, the memory subsystem has the highest CPU percentage, and most of its functions are related to *sk_buff*, a fundamental data structure for describing the control information used in packet handling. Since the operations on the data

[*] *Multiport Memcached* is discussed in section 1.1.2

Description	**CPU**
Receive/transmit, event-handler functions in *Memcached* and *libevent*	8.26%
Memory copy between kernel and user levels, system calls and polling system routines	7.98%
SOCKET layer: receive/transmit functions	7.66%
UDP layer: receive/transmit functions	7.75%
IP layer: receive/transmit functions, connection tracking, filtering, and routing	11.64%
ETH and driver layer: RPS [31], *e1000e*, and receive/transmit functions	15.42%
Memory subsystem: skb/slab functions	23.32%
Scheduling, softirq, timers, and other routines as well as overheads from OProfile	17.21%

Table 3.1: Distribution of the CPU cycles in different categories of functions at the user level (first row) and at the kernel level (other rows) during the execution of *Multiport Memcached*.

structure—such as memory allocation/deallocation and modification—are required in each layer of the network stack, it is challenging to improve its performance at one layer without negative impact on other layers. Meanwhile, much effort has been spent on applications' in-kernel implementations using the kernel TCP/UDP sockets simply to remove overhead associated with the user-kernel border [3, 44, 51, 65]. However this approach may not suffice, at least for *Memcached*. As shown in the table, the total percentage for the user-level functions, including *libevent*, is only 8.26%, and kernel functions directly related to the user level, including memory copy, system calls, and the polling routines, consume only 7.89% of the total cycles.

Although there exist many studies on the optimization of the network stack via parallelization on multicore system, such as distributing packets among CPU cores [50], reducing the number of packets using jumbo frames [18], and mitigating interrupts [36, 61], efficient parallelization of the stack remains difficult due to overhead from synchronization, cache pollution, and scheduling in the layers of the network stack in a multicore system [55, 62, 72]. To reduce overhead due to unnecessary sharing of network control states in a multicore system, *IsoStack* [62] separates cores for supporting the network stack from those running applications. However, *Memcached* does not consume many CPU cycles for its own, as shown in Table 3.1, and could hardly benefit from this technique. Recent work (*Netmap* [60]) provides applications with line-rate access to raw packets by bypassing kernel network stack supporting the TCP/UDP protocols. However, it can be hard for a general-purpose application like *Memcached* to take advantage of this capability and retain compatibility with clients. Other works such as *Chronos* [35] rely on user level networking [70] enabled by NICs exposing user-level interface to handle requests without kernel intervention. However, it is still a significant challenge to effectively achieve scalable access to the user-level NIC because the amount of NIC resources demanded for managing user-level connection endpoints increases linearly with the number of clients simultaneously issuing requests [70, 45]. The number can be substantial in *Memcached* service [10].

Having shown that *Memcached*, as a representative KV cache implementation, is CPU-bound with the network stack at high loads, we cannot readily leverage existing network techniques to effectively address the issue. As the KV cache is such a critical component in today's data center infrastructure [52], it is time to revisit the conventional wisdom that this network-intensive class of applications are improved only through optimization of the network stack.

3.2 The Design of *Hippos*

Three principles guided *Hippos*'s design. First, it should take into account the characteristics of the KV-cache's expected workloads. Second, it should remove a substantial amount of network-related overhead. Last, it should require minimal or even no changes to the existing kernel network framework. In this section, we describe the design of *Hippos* in light of these principles, starting with its expected workloads.

3.2.1 Targeted Workloads

Hippos is motivated by the suboptimal performance of *Stock Memcached* under realistic workloads, taken from Facebook's workload study [10]. These workloads show a strong bias towards small requests and require that servers be provisioned to handle large traffic spikes. Below is a summary of relevant characteristics reported in the *Memcached* workload study.

- The ratio of the GET requests among all requests can be very high. Among the five separate caching pools, each dedicated for a different application or data domain, USR has the highest GET ratio (99.7%). The ratios for the other pools are 84% (APP), 73% (ETC), 18% (VAR), and 67% (SYS). Furthermore, all GET requests use UDP, instead of TCP, for higher efficiency.
- Small values and keys dominate GET requests. For the USR pool, there are only two key sizes (16B and 21B) and virtually only one value size (2B). For the other four pools, APP, ETC, VAR, and SYS, the 99% percentile key sizes are 45B, 80B, 30B, and 45B, respectively. Almost all GET requests can be held in a single UDP packet. Their respective 99% percentile value sizes are 450B,

512B, 200B, and 640B. Most of the GET requests and their replies can be held in one UDP packet.

- The request traffic can quickly surge by doubling or tripling the normal peak request rate. It has been suggested that "one must budget individual node capacity to allow for these spikes [...] Although such budgeting underutilizes resources during normal traffic, it is nevertheless imperative" [10].

Based on these workload characteristics, the design of *Hippos* is focused on improving the performance and efficiency of processing UDP-based GET requests, especially small ones. We believe this effort should also benefit other KV stores used in data centers supporting web-based applications in general.

3.2.2 Locating the Position to Hook *Hippos* in

While the general idea is to move the KV cache into the kernel and bring it closer the NIC, we must still identify a position in the network stack for an implementation that significantly reduces networking cost and is the least intrusive to the existing network architecture. To this end, we selected four observation positions along the traversal path of *Memcached*'s requests to evaluate CPU overhead and latency for the traffic to reach these positions (see Figure 3.1). To ensure that we only account for statistics taken before a certain position is reached, we intercepted and then dropped the packets at this position. Table 3.2 describes these selected positions. Among them, position 1 is the closest to the NIC and packets are intercepted immediately before they reach the IP layer. We use *Netfilter*'s hook (*NF_INET_PRE_ROUTING*) to obtain the packets and then drop them. Position 2 is selected immediately before UDP packets are added into the UDP socket buffer queue. To drop the packets, we open UDP socket(s) but do not read packets from them. When the socket queue is

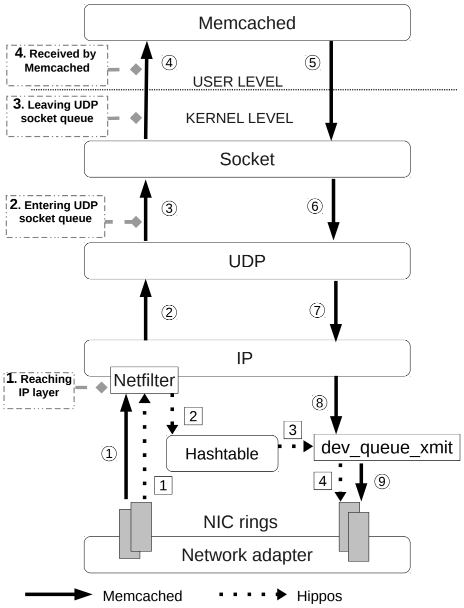


Figure 3.1: The paths for a UDP GET request to travel in the network stack and *Memcached* (or respectively *Hippos*).

filled, the subsequently arriving packets will be automatically discarded. At position 3, we use kernel-level thread(s) to pick up packets from the UDP socket buffer queue once they are notified that there are new packets inserted into the queue. Position 4 is the location conventionally used for *Memcached* to receive UDP packets.

In this investigation the workload is the same as that used for the experiments described in Section 1. Figure 3.3 shows that CPU utilization at various observation positions with one core. Figure 3.2 shows corresponding latency for the packets to reach these positions. In the measurement of latency, we may have to correct the skewed clocks between clients and the server as the packets are dropped on their way to the *Memcached*. To avoid possible errors in the correction, we chose to measure the start time of a packet when it is just received by the server (at the NIC driver). As

Position	Method to intercept packet
1. Reaching IP layer	via *Netfilter* hook NF_INET_PRE_ROUTING
2. Entering UDP socket queue	Open the socket w/o reading requests
3. Leaving UDP socket queue	Reading requests w/o sending them to *Memcached*
4. Received by *Memcached*	Process in *Memcached*

Table 3.2: The observation positions

shown, at positions 1 and 2 the CPUs are almost all idle and the latency is minimal even when the arrival rate reaches $800K$ packets per second. However, at position 3, system time starts to become substantial and even dominating when the arrival rate reaches 800K packets per second, and the latency skyrockets from 10μs to over 200μs when the rate is beyond $480K$ packets per second. When the packets reach the user level at position 4, the system's packet processing capacity is saturated by an arrival rate of only $320K$ packets per second. Note that position 4 is at only the half way of a round-trip request and reply path in *Memcached*. If the full path is considered, the saturation arrival rate would come much earlier, as illustrated in Figure 1.1. The experiments to run *multiport Memcached* on multiple cores reveal similar performance trend at these observation positions, except that higher peak throughput are observed.

A major reason why receiving packets at positions 3 and 4 is expensive is the context switch between threads placing packets into the socket buffer queue and retrieving them out of it. Position 4 is additionally associated with overhead related to passing packets between the kernel and the user-level applications. Between positions 1 and 2, *Hippos* chooses the first position to intercept packets as it can leverage the

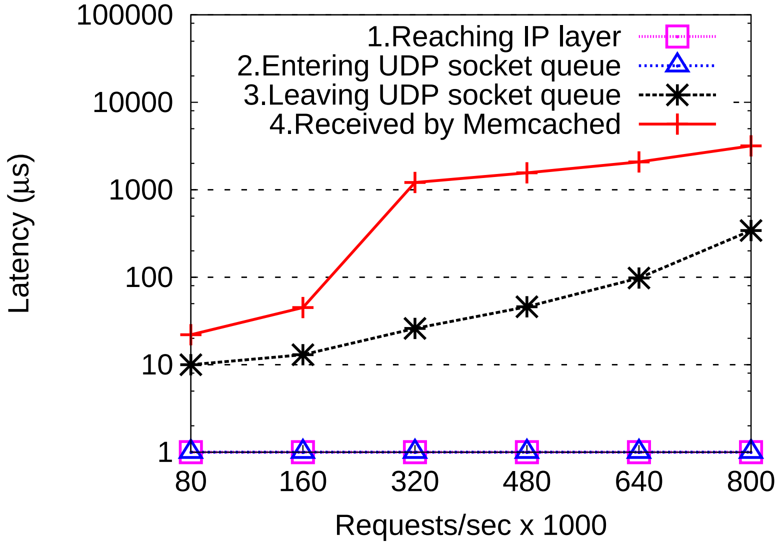


Figure 3.2: Latency observed at various observation positions in the network stack with different request arrival rates and one core in use. The latency of a packet is measured as the duration between when it is received (*netif_receive_skb()*) and when it reaches a particular observation position. Note that the Y axis is on the logarithmic scale.

Netfilter framework [2] to obtain packets without any modification of the operating system. *Netfilter* provides a number of hooks within the Linux network stack. These hooks can be used to register kernel modules for manipulating network packets. *Hippos* uses the *NF_INET_PRE_ROUTING* hook. Although packets received from the hook are still at the IP layer, all the information needed for the KV cache is available, such as operation type, number of keys, key contents, or values. After receiving a packet, *Hippos* will first check it to see whether it is a UDP GET packet, and if so, whether its destination port is the one defined by the KV cache. If a packet does not satisfy both conditions, *Hippos* will return *NF_ACCEPT* in its hook function to allow the packet to resume its journey in the network stack towards the upper layers,

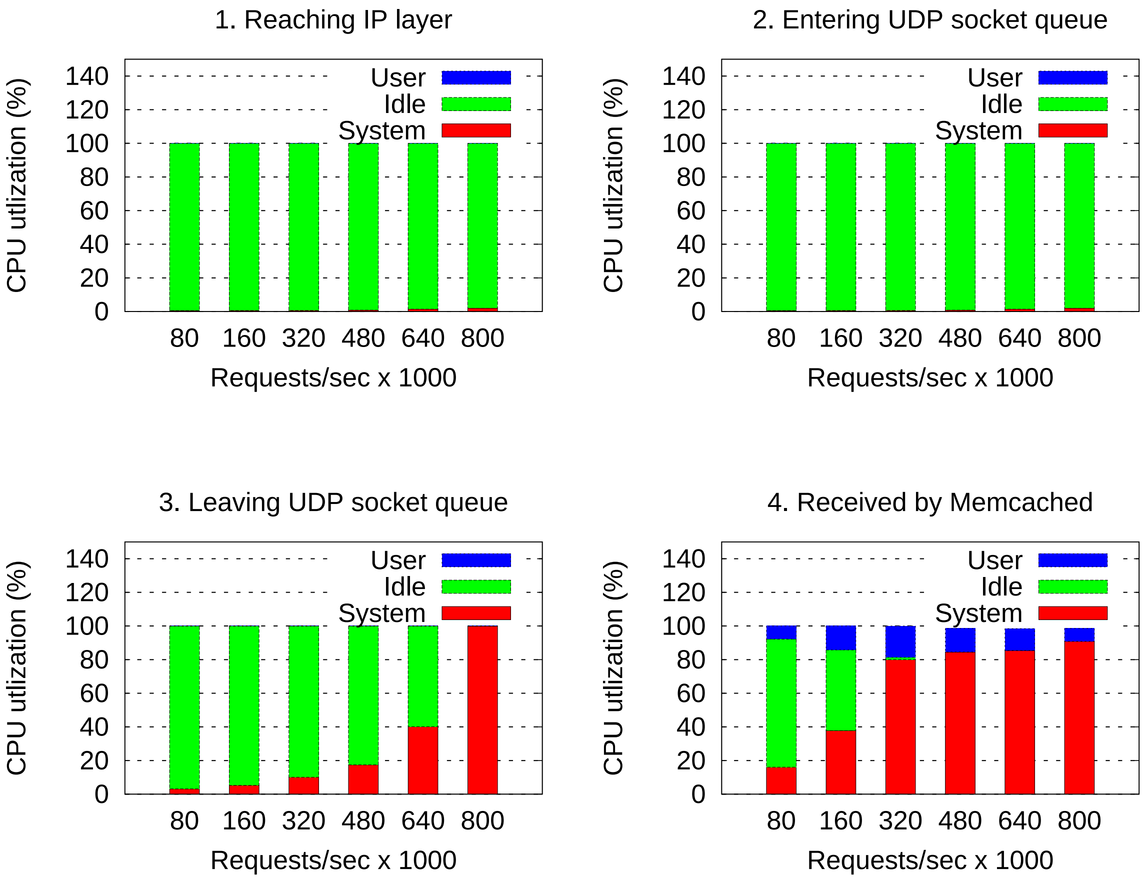


Figure 3.3: CPU utilization at various observation positions in the network stack with different request arrival rates when one core is in use.

such as UDP layer[†]. Otherwise, *Hippos* retrieves the request from the packet and feeds it into the in-kernel KV cache for processing similar as that in *Memcached*. The query result will be sent in a packet directly from the IP layer (via function *dev_queue_xmit()*). If the *key* or *value* cannot be held in one UDP packet, a number

[†] It is noted that *Hippos* does not have any UDP sockets at all.

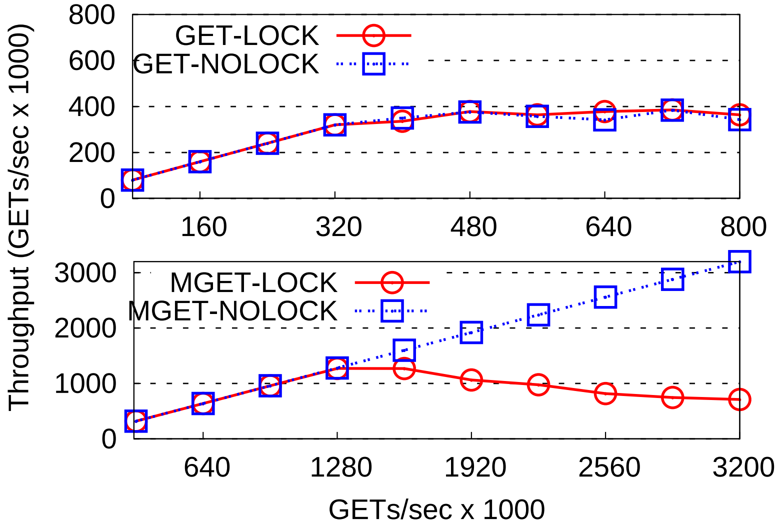


Figure 3.4: *Memcached* throughput (number of GETs processed per second) under various GET request arrival rates. The GET requests arrive either in the one-request-in-a-packet format (GET) or in the multiple-requests-in-a-packet format (MGET). For MGET, the packet arrival rate is $320K$ packets per second, and we change the number of requests in a packet. The lock may be applied ($LOCK$) or not ($NOLOCK$).

of UDP packets will be created and sequence numbers are placed in them, as what is done in *Memcached*.

Note that a GET request is processed in the context of *softirq* handling, rather than by another thread. This avoids the context switch between network stack routines and worker threads for reading and processing requests. The path for the UDP packets to travel in *Hippos* is shown in Figure 3.1.

3.2.3 Removal of the Second Bottleneck

In the previous investigation, we assumed that locks in *Memcached* are disabled to take out lock-related cost and highlight the cost related to the packet processing in the network stack. Now we have two questions to answer: (a) Did we overestimate the performance of *Memcached* by removing lock contention? (b) If the packet processing in the network stack is not the bottleneck, what is the effect of the lock-related cost on *Memcached*'s performance? To answer the first question, we ran *Multiport Memcached* on eight cores with RPS (Receive Packet Steering) enabled and with the same workload as before except that GET requests retrieve data that have been in the KV cache. Because of maintenance of data structures for the Least-Recently-Used (LRU) replacement policy, lock operations can be required even for GETs. As shown in the upper graph of Figure 3.4, after we enabled the locks at the increasing packet arrival rate the system achieves the same throughput as that for its counterpart with *Memcached* internal locks disabled. In other words, the lock overhead is overshadowed by the network cost and thus is not a performance issue unless the network cost is sufficiently reduced. After the load increases beyond $320K$ packets per second the throughput increases little, which indicates that *Memcached* cannot receive sufficient GET requests to allow its lock use to become a performance bottleneck (here we assume one GET request per packet).

To answer the second question, we need to increase the number of GETs without increasing network cost. To this end, we placed multiple GETs in a UDP packet and kept packet arrival rate constant at $320K$ packets per second. Before the workload increases to $1280K$ GETS per second (by placing more GETs in a packet), the throughput in terms of number of GETs serviced in one second almost linearly increases. But beyond this point the throughput peaks and starts to drop. This is attributed to intensified contention on the *Memcached*'s internal locks as we observed that the cores still have idle time. If we disable the locks in the experiment, the throughput maintains its linear increase. Ostensibly, this represents the best-case performance, because the locks cannot be disabled in a real workload that includes mutating requests, such as SET and DELETE.

Currently *Memcached* uses a set of locks for its hash table, each for a number of buckets in a hash value range, and one lock to maintain consistency of the data structure for its LRU cache replacement policy. When traffic to *Memcached* is high, the request processing can become serialized by these locks. Even worse, a thread owning a hash table lock cannot release it until it acquires the LRU lock and completes its operations on the LRU stack to keep the consistency of these two data structures. To address the issue, we synergistically apply two techniques. First, we replace the spinlock for the hash table with RCU (Read-Copy Update) lock [47, 48]. RCU allows readers to access the shared data without any conventional lock. For writes, it creates new copies to accommodate updates before old copies are freed. In RCU, reads can be much cheaper than writes. As it has been shown that in the *Memcached* workloads, GETs can be much more frequent than update requests, RCU is an ideal fit in the enforcement of mutual exclusiveness. Second, we adopt the CLOCK policy instead of LRU to completely remove the use of locking for cache replacement.

3.2.4 Handling TCP packets

Hippos uses the in-kernel TCP socket to receive SET, REPLACE, DELETE, and other writing requests. However, it does not optimize its reception and processing of TCP packets except that it handles them in the kernel. This relieves us from re-implementing the complex TCP stack. For NICs that have multiple hardware receive queues, we run one thread on each core to handle TCP connections. For NICs with only one queue when NAPI [61] is enabled, *Hippos* needs to spread the load across cores. It accomplishes this by creating a worker thread listening on the incoming TCP connections on the core responsible for polling the NIC for incoming packets in NAPI. *Hippos* creates $N - 1$ worker threads to handle connections on top of the socket layers, where N is the number of cores, and each of the threads runs on one of the remaining cores. The threads are woken up via the *sk_data_ready* callback function to serve incoming connections from clients in a round-robin manner. We chose *TCP_NODELAY* to disable the Nagle algorithm [4] to reduce the response time to clients. Though *Hippos*'s TCP packet handling is at a high position in the network stack, it does avoid memory copy and other overheads associated with the user-level applications.

3.2.5 Distribution of workload among cores

In a NIC with only one hardware receive queue or one *rx_ring*, NAPI is used to change the packet reception from the interrupt-driven mode into polling mode when the flow of incoming packets exceeds a certain threshold. In the polling mode, only one core polls the device for incoming packets. *Hippos* may choose to use only this core to invoke *softirq* for processing UDP GET requests. The advantages of this approach include no incurring of the cost for delivering packets to the backlog queue of other cores and leaving those cores mostly idle to save energy. However, when the workload on this core is very high, especially when expensive TCP packets are frequent, the core can be overwhelmed. To address this issue, we enable RPS to spread the load across the cores when this core's utilization reaches a threshold, which is set at 70% by default. Our experience indicates that *Hippos*'s performance is not sensitive to the threshold. RPS will be turned off when NAPI is disabled at a lower packet rate.

3.2.6 Reuse of *sk_buff*

The data structure *sk_buff* is used to store data and control information for packets. If a GET is a miss or the value retrieved from the KV cache is smaller than payload of the original GET packet, *Hippos* reuses the packet by directly storing the value in it. Accordingly it switches the source and destination addresses for various layers, including those in the UDP headers, IP headers, and MAC headers, and sends the packet back to the client. Considering the potentially large number of cache values whose sizes are only a few bytes [10], this optimization can effectively reduce the cost associated with allocations and de-allocations of *sk_buff*s. To enable this reuse, *Hippos* returns *NF_STOLEN*, rather than *NF_DROP*, in its *Netfilter* hook function. So that it can retain the *sk_buff* for updating and creating a reply packet. If the reply data is larger than the capacity of the *sk_buff*, it will expand the buffer.

3.3 Effectiveness of *Hippos*

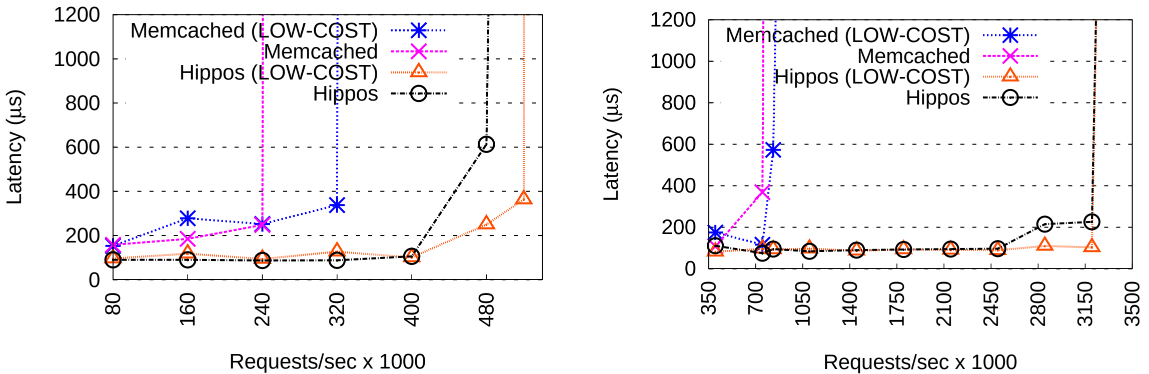
Hippos was implemented as a separate Linux kernel module that can be easily loaded without requiring any modifications to the kernel itself. The experiments for this evaluation were first conducted on the same platform as before: each node has 8-core Intel 2.33GHz Xeon CPU, 64GB DRAM, and Intel PRO/1000 1Gbps NIC, running Linux 3.5.0. A server node is connected with another eight client nodes of identical configuration. The use of a 1Gpbs NIC, which is embedded in the motherboard, is quite common for clusters in large-scale data centers [28]. It provides a raw bandwidth larger than what is demanded by *Memcached* traffic discussed in the study of Facebook *Memcached* traces [10], which are also used in our evaluation. For a KV-cache workload dominated by small keys and values, whose combined sizes are less than 1KB, it is the network stack, rather than the hardware's raw bandwidth, that is stressed. The client-side software interacting with the *Memcached* server does not need to make any changes after *Hippos* replaces *Memcached*. On each client machine there are four processes generating *Memcached* workloads, each sending asynchronous requests to the server at a settable rate, either as a micro-benchmark or by replaying the Facebook traces. In addition, we demonstrate how the benefits of *Hippos* can be scaled up with a 10Gbps network by using the dual-port Intel 82599 10Gbit Ethernet cards with the 3.10.16 IXGBE driver. To saturate the higher bandwidth, we used 24 client machines to issue requests. In the meantime, we used a more powerful machine as the server, a DELL PowerEdge R410 with two Intel Xeon X5650 processors and 32 GB memory. As each processor has six cores and with hyperthreading each core has two logical cores, we consider the server to have 24 logical cores.

In this section we also evaluate the open-source *Memcached* v1.4.15 for comparison. Considering the apparent weakness of using only one UDP socket in the open-source *Memcached* and the adoption of its multiple-UDP-port version in the industry [52], we use *Multiport Memcached* in this evaluation to represent *Memcached*. In addition to peak throughout and average latency, in the experiments we also measured the electric power consumed at the server's socket. Unless otherwise indicated, we prepopulate the cache before each run and issue requests with random keys from the cache.

We first used micro-benchmarks to evaluate the performance of *Hippos* under a controlled workload and observed how its various design aspects respond to the changes of workload characteristics. Unless otherwise specified, a packet is sized for a 64 B payload.

3.3.1 Identifying Peak Throughput

Generally speaking, increasing request arrival rate in a KV store system would increase average request latency until peak throughput is reached and latency grows unacceptably high. To see how the latency grows and when the peak throughput is reached, we let clients send UDP GET requests to *Memcached* and *Hippos* with increasingly higher rate. In the request packet, the key size is 20B and in the reply packet the value size is also 20B. Figure 3.5(a) and Figure 3.5(b) show the latencies with the increasing request rate for 1Gbps and 10Gbps networks, respectively. As a reference point for the best-case scenario, we also plot the latencies for an undemanding workload, in which only non-existent keys are requested and the lock for the hash table is disabled as its protection is not necessary for the 100%-miss requests.



(a) 1Gbps

(b) 10Gbps

Figure 3.5: Request latencies for *Memcached* and *Hippos* with increasing request arrival rate in the 1Gbps network (a) and in the 10Gbps network (b). For each system, latencies for a low-cost setup (LOW-COST) are also reported, in which requests are for non-existent keys in a hash table not protected by locks.

In both systems, the latency does not increase substantially when the request rate is low, though *Hippos* produces latencies lower than *Memcached*'s. However, the latency skyrockets when the request rate reaches its peak rate (corresponding to peak throughput). Observe the 1Gbps-network scenario for example: in the undemanding set up *Hippos* improves *Memcached*'s peak throughput by 63% (520 Req/s vs. 320 Req/s). In the normal setup both have their peak throughput reduced, but *Memcached* by a larger amount. This is because *Hippos* has already eliminated the cost of lock protection associated with GETs with the use of the RCU lock and the CLOCK replacement, and its undemanding setup has only the benefit of reduced search cost in the hash table due to mapping non-existent keys to an empty bucket. Consequently, *Hippos* doubles *Memcached*'s peak throughput ($480K$ Req/s vs. $240K$ Req/s). The performance trend for the 10Gbps network is similar except that (1) *Hippos* has a larger improvement of peak throughput (more than 4×); (2) the difference of undemanding setup and normal setup for either *Memcached* or *Hippos* is smaller. The reason for the larger improvement in the 10Gbps network is that *Hippos* shifts the throughput bottleneck from the CPU to the network. Accordingly a 10Gbps network exposes more of *Hippos*'s potential. The smaller difference is because that in the 10Gbps network the system time holds a larger percentage in the program's execution. This is likely attributed to the aggravated cache line miss due to the fact that different cores are used for delivery of packets using RSS (Receive Side Scaling) in the 10Gbps NIC and for running application threads [54].

3.3.2 Reducing Memory Operations

Hippos has attempted to reduce memory allocation and de-allocation operations by reusing the *sk_buff* data structure. For small values that can be held in the request

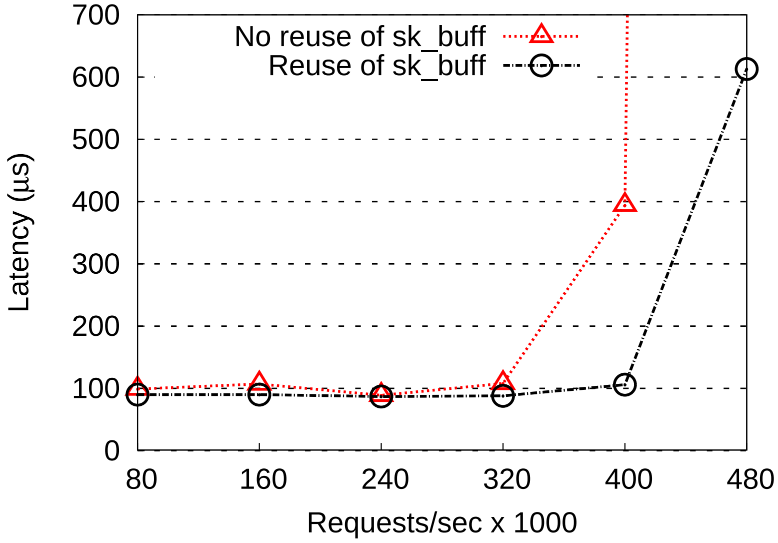


Figure 3.6: Request latencies for *Hippos* with and without using the *sk_buff* reuse optimization when the request arrival rate increases.

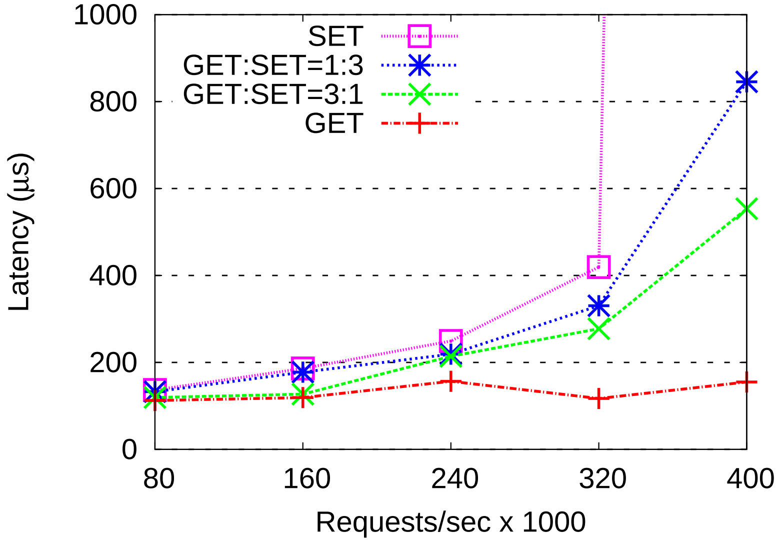


Figure 3.7: Latencies for workloads with different mixes of GETs and SETs and different request rates.

packets' *sk_buff*, the operations' cost is proportional to the request arrival rate. So we increase the rate to see how much performance benefit can be received by using this optimization in *Hippos*. In this experiment, we send UDP GETs, each with a 20B key and searching for a 20B value, in the 1Gbps network. Figure 3.6 shows request latencies under different request rates when the optimization is applied or not. Although the latency reduction is small with the reuse when the request rate is low, the technique is effective at high request rates. In particular, it successfully increases the peak throughput by 20% (from $400K$ req/s to $480K$ req/s).

3.3.3 Mixing GETs with SETs

Processing both GETs and SETs in *Hippos* takes place in the kernel to eliminate the cost associated with interactions between the kernel and user-level *Memcached*. However, *Hippos* makes more aggressive optimizations for GETs. In this experiment we show how mixing SETs with GETs would change the performance observations we have made on the all-GETs workloads. Figure 3.7 shows latencies for workloads with different mixes of GETs and SETs in the 1Gbps network. With low request rate ($80K$ reqs/s), having SETs in the workload almost does not increase latency. However, with the increase of request rate the workloads with higher proportion of SETs have higher latencies. For example, at $320K$ reqs/s, the workload with all SETs sees latencies jump beyond $1ms$. This is the result we expect as TCP-based SETs are more expensive to process. In the meantime, even under mixed workloads, *Hippos* outperforms *Memcached* since it can also improve performance for SETs, albeit at a smaller scale.

	USR	**ETC**	**APP**	**VAR**	**SYS**
GET	99.7%	73.4%	83.4%	18.0%	67.5%
UPDATE	0.2%	2.3%	5.2%	82.0%	32.5%
DELETE	0.1%	24.0%	11.4%	N/A	N/A

Table 3.3: Distribution of request types in the Facebook traces: GET, UPDATE, and DELETE. SET belongs to the UPDATE category, which also includes REPLACE and other non-DELETE writing operations.

3.3.4 Replaying Facebook's Traces

We replayed Facebook's production-representative *Memcached* traces on *Hippos* with both 1Gbps and 10Gbps NICs. The five traces (USR, ETC, APP, VAR, and SYS) have been briefly described in Section 2. An extensive description and analysis can be found in [10]. Here we summarize the distribution of requests in each trace in Table 3.3. The requests are categorized into types: GET, DELETE, and all non-DELETE writing operations such as SET and REPLACE, which are collectively named UPDATE. Table 3.4 lists the average latencies of the three types of requests and power consumption for *Memcached* and respective changes made by *Hippos* in percentage for all five traces. For each trace, we use three request arrival rates, representing low, medium, or high loads on *Memcached*. Figure 3.8 (a) and (b) show the peak throughput received by *Memcached* and *Hippos* for the 1Gbps and 10Gbps networks, respectively.

From the experimental results we gathered several interesting observations. First, for the 1Gbps network *Hippos* achieves the most impressive improvements for traces USR and VAR, each for a different reason. According to Table 3.3, USR consists

Type	Rate	Multiport Memcached				Hippos			
	K/sec	GET(μs)	UPDATE(μs)	DELETE(μs)	POWER(Watt)	GET	UPDATE	DELETE	POWER
USR	160	173	206	194	330	-28%	+6%	-1%	-19%
	240	235	234	220	343	-45%	-6%	+3%	-16%
	320	—	286	273	347	226μs	-11%	+11%	-15%
USR 10Gbps	750	680	640	650	191	-26%	-35%	-35%	-20%
ETC	80	327	183	166	302	-41%	-17%	-14%	-7%
	160	916	224	207	324	-72%	-20%	-17%	-9%
	240	—	279	263	337	471μs	-14%	-11%	-9%
ETC 10Gbps	845	842	694	670	200	-39%	-8%	-7%	-14%
APP	80	289	185	167	303	-48%	-14%	-7%	-10%
	160	547	230	214	324	-53%	-18%	-15%	-10%
	200	—	237	361	337	386μs	+42%	-37%	-10%
APP 10Gbps	763	713	665	650	194	-24%	-21%	-6%	-14%
VAR	40	163	163	N/A	286	-23%	-11%	N/A	-5%
	80	186	179	N/A	316	-32%	-12%	N/A	-10%
	120	—	—	N/A	326	174μs	163μs	N/A	-9%
VAR 10Gbps	150	920	965	N/A	178	-35%	-36%	N/A	-13%
SYS	80	376	174	N/A	304	-54%	-10%	N/A	-6%
	120	331	201	N/A	319	-52%	-19%	N/A	-5%
	160	—	—	N/A	323	230μs	231μs	N/A	-6%
SYS 10Gbps	232	992	978	N/A	181	-44%	-37%	N/A	-13%

Table 3.4: Average request latency and power consumption of *Memcached*, and respective changes made by *Hippos* in percentage for the five traces with the 1Gbps and 10Gbps networks (only 10Gbps explicitly indicated). Latency larger than $1ms$ is denoted by "-". If *Memcached*'s latency is denoted as "-", *Hippos*'s counterpart is represented by its actual latency value, instead of a change in percentage.

of almost entirely GETs (99.7%). Both request packets (with only 16B and 21B keys) and reply packets (with virtually only 2B values) are small. This is exactly the type of workload *Hippos* excels at. GET latency is reduced significantly, especially when the request rate is high. The peak throughput is increased by 98% and energy consumption is reduced by 19%, 16%, or 15%, depending on the request rate. In contrast, VAR is UPDATE-dominated (82%). We found that *Memcached* is especially ineffective in processing SETs or other update requests. With an arrival rate of only $120K$ reqs/s, the latency increases to a couple of milliseconds. This allows *Hippos* to achieve a high increase (2.5×) of peak throughput. However, the power savings (5%, 10%, and 9%) are less significant because TCP-based UPDATEs keep all cores busy and *Hippos* can hardly use only one core to serve requests.

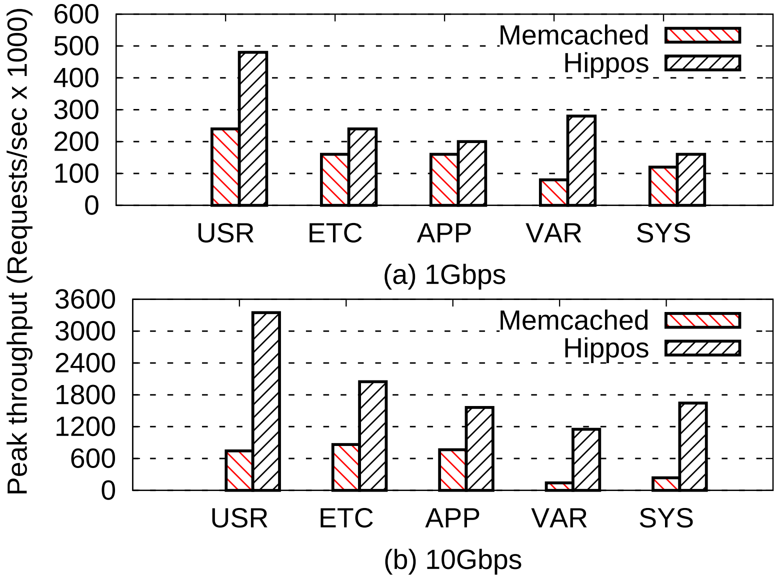


Figure 3.8: Peak throughput received by *Memcached* and *Hippos* for each of the five Facebbok's traces. The throughput is collected under the condition that the corresponding average request latency does not exceed $1ms$.

Second, ETC, APP, and SYS have relatively moderate improvements in the 1Gbps network. Both have substantial portion of GETs (73.4%, 83.4%, and 67.5% for ETC, APP, and SYS, respectively). However, they have relatively large values. For example, in more than 30% of APP's SETs, value sizes are around 270B. ETC also has a significant portion of large value size, even a few of around 1MB. GET requests for these large values will produce large reply packets. This can bring packet bandwidth close to the NIC's raw bandwidth, which then turns into the bottleneck and limits the potential improvement by *Hippos*. *Hippos* improves the peak throughput of ETC, APP, and SYS by 41%, 15%, and 33%, respectively. When the 10Gbps NIC is used, it breaks the limit and gives *Hippos* a larger room for improvement. As shown in Figure 3.8(b), the peak throughput of ETC, APP, and SYS is improved by 140%, 100%, and 590%, respectively, in the 10Gbps network.

Third, the improvement trends with increasing request arrival rates are different for latency and power. In general, at low request rate the latencies for *Memcached* are acceptable and do not leave too much room for *Hippos* to improve. When the request rate approaches *Memcached*'s peak throughput, the latency with *Memcached* quickly rises, and accordingly *Hippos* usually produces a big improvement, especially for GETs. However, the improvement on power consumption is usually consistent across different request rates. For example, with $80K$ reqs/s, $160K$ reqs/s, and $240K$ reqs/s for ETC in the 1Gbps network, the improvements of GET latency are 41%, 72%, and 92%, respectively, while the improvements on power consumption are more consistent (7%, 9%, and 9%, respectively). To understand the consistency of power saving, we used the Linux performance counter profiling tool *perf* to measure the number of instructions executed with *Memcached* and *Hippos.* For ETC, with the three request rates *Hippos* reduces the instruction count by 45%, 53%, and 51%, respectively. These reductions are less correlated to request rate but correlated to power saving. So even for KV store users who see relatively low request rate and might not be interested in latency improvements as long as the latency is not too high, such as exceeding $1ms$, *Hippos* can be still appealing with its advantage on power saving across the different request rates.

3.4 Related Work

In this section, we briefly describe the efforts in the literature for optimizing KV store in general, and *Memcached* in particular, and the techniques enabling the optimizations.

3.4.1 Optimization of *Memcached*

While *Memcached* usually runs on multicore processors, it remains a concern whether operating-system support for multicores can hamper its scalability. It has been found that running multiple *Memcached* instances, each on a dedicated core with a separate worker thread, allows it to scale with increased core count [14, 15]. In contrast, *Hippos* addresses the performance issue of *Memcached* from a different angle. Instead of making increased CPU cycles available to *Memcached* to meet its high CPU demand, *Hippos* reduces its reliance on powerful processors, making *Memcached* a much lighter KV cache. In doing so, *Hippos* still provides one port per server to all clients and the memory is fully shared by all cores, facilitating ease of management. In contradistinction, the approach of running multiple *Memcached* instances in one server has to partition memory among instances or cores, and can lead to load imbalance: if some items in one instance are accessed more frequently than others in a different instance, the demands on different cores can differ significantly. The load imbalance issue also exists in CPHASH [49], a hash table designed for KV stores, as it also needs to partition the hash table in advance.

Recently there have been optimized synchronization mechanisms [26, 52, 67] proposed to reduce or eliminate lock contentions within *Memcached*. However, the lock contention on the network stack can still dominate *Memcached*'s performance. *Hippos* reduces or removes lock contentions on both the KV cache's implementation and the network stack.

Contemporary Linux kernels also provide some mechanisms that help with *Memcached*'s network efficiency. For example, NAPI [61], RPS [31], and RSS [50] address the efficiency issue on selecting incoming packets from the NIC driver under heavy network loads. *Hippos* adopts these techniques in its implementation. However, using the network optimizations alone cannot address network efficiency issues challenging *Memcached* as long as it stays on top of network stack as a user-level application.

3.4.2 Moving Applications into the kernel

Many solutions have been proposed for saving energy at the server side [41, 40, 39, 42]. Migration of services that are considered integral to a server's operation into the kernel has been in practice for this purpose. kHTTPd [44] and TUX [38] are two projects that moved web server into the Linux kernel with the aims of elimination of data copies and reads, reduction of thread scheduling and context switching overhead due to event notification, and reduction of overall communication overhead in the network stack. Click is an in-kernel modular router allowing fast access to NIC [23]. *SPIN* [66] is an operating system that blurs the distinction between kernels and applications, and has a web server running entirely in its kernel address space to reduce response times. *Hippos* is also an in-kernel implementation that maximizes the performance and energy efficiency. Since a KV caching service is usually provided on dedicated servers to other internal applications, integrating it within the kernel and approaching the servers as appliances mean fewer negative implications—such as security concerns, in the data-center environment—and several positive implications such as improved performance and energy efficiency.

3.4.3 Making network resources accessible at the user level

To allow packets to be sent or received more quickly by applications, many efforts have been made to provide them with more direct and efficient interfaces to access network resources. *Netmap* is a framework providing applications with a fast channel to exchange raw packets with the network adapter to achieve at-line rate for packet transmission [60]. Though it provides an opportunity for user-level *Memcached* to directly access packets, this approach can be difficult to implement. For example, the handling of TCP needs to be reimplemented at the user level, which can be more expensive than in the kernel. *Netslice* is a framework within a kernel module that uses the *Netfilter* hooks to pass packets directly to the user level [46]. By using *Netfilter* hooks for intercepting packets, *Netslice* is similar to *Hippos*. However, by directly passing packets to the user level, it shares the concern with *Netmap* had *Memcached* been built in its framework.

System call can place a major burden for applications to access network resources. Soares *et al.* [64, 63] proposed a system mechanism, named as exception-less system calls, enabling efficient data access between use- and kernel levels. In addition, the design and implementation of OS support for asynchronous operations along with event-based notification interfaces to support event-driven architecture, have been an active area of both research and development [12, 16, 58, 24, 17, 53, 71, 59]. All these works aim to reduce the communication overheads between kernel and user level. As *Hippos* is implement in the kernel, there overheads have been fully removed.

3.4.4 *Netfilter* hooks

Hippos highly relies on the convenience and efficiency in packet interception provided by *Netfilter* [2]. *Netfilter* provides a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the network stack. A registered callback function is then called back for every packet that traverses the respective hook within the network stack. A most known application of *Netfilter* is to construct firewall through inspecting packets taken from *Netfilter*. Similar framework is also provided in FreeBSD (Netgraph) and Windows (Ndis Miniport drivers).

3.4.5 Reuse of sk_buff

It has been found that allocation/de-allocation of *sk_buff* can be a major consumer of CPU cycles – *sk_buff*-related operations take up 63.1% of the total CPU usage [30]. To address this issue, a new buffer allocation scheme is used for acquiring a large packet buffer in one allocation for many *sk_buff*s to amortize the cost. The cost of *sk_buff* can be related to where it is allocated in a NUMA system. It can incur serious lock contention if many allocators access the same free *sk_buff* list. By allocating from a local list, the contention can be alleviated and the allocation of *sk_buff* can be more efficient [15]. *Hippos* significantly reduces the *sk_buff*-related operations, especially allocations/de-allocations, using a simple strategy: reuse of the buffer of an incoming request packet for constructing outgoing reply packet.

Chapter 4: Conclusions and Future Work

This dissertation presents a detailed workload study on a large-scale KV store system that runs at Facebook, and proposes a novel way to build a high-throughput, low-latency, and energy-efficient KV cache system. In this chapter, we summarize the work that have been done and suggest directions for future work.

4.1 Conclusions

This dissertation exposes five workloads from one of the world's largest KV-cache deployments. These five workloads exhibit both common and idiosyncratic properties that must factor into the design of effective large-scale caching systems. For example, all user-related caches exhibit diurnal and weekly cycles that correspond to users' content consumption; but the amplitude and presence of outliers can vary dramatically from one workload to the next. We also investigate at depth the properties that make some workloads easier or harder to cache effectively with *Memcached*. For example, all workloads but one (SYS) exhibit very strong temporal and "spatial" locality. But each workload has a different composition of requests (particularly the missing ones) that determine and bound the cap for potential hit-rate improvements.

One particular workload, ETC, is interesting and useful to analyze because it is the closest workload to a general-purpose one, i.e., not tied to any one specific Facebook application. The description of its performance and locality characteristics can therefore serve other researchers in constructing more realistic KV-cache models and synthetic workloads.

This dissertation also describes the design and implementation of *Hippos*, an in-kernel KV cache implementation to support cloud services. We believe that a KV cache should be memory-intensive and network-intensive, but not CPU intensive, in accordance to its role as a large on-network caching facility. In this paper, we show that current user-level *Memcached* is a highly CPU-demanding application. Together, packet processing in the kernel and the use of locks within *Memcached* can dominate processing time.

Considering that *Memcached* provides caching services as part of the infrastructure in a data center, we move it into the kernel to remove most of network-related costs. In addition, we use the RCU lock and a lock-free CLOCK replacement to substantially remove lock contention within the KV store. The resulting *Hippos* is a high-performance and high-efficiency KV system with three distinct advantages: (1) It is highly CPU efficient: with a single core its throughput outperforms open-source *Memcached* running on eight cores; (2) It is energy efficient: it can reduce power consumed by a *Memcached* server by up to 20% for production-representative workloads. (3) Its design is based on observations from real-world workloads and its performance about replaying the workload traces shows substantial gains.

Exploiting the readily available *Netfilter* interface in the kernel, *Hippos*'s implementation does not require any kernel modifications. Our experience suggests that in data-centers specialized clusters, providing network-intensive services can be optimized with in-kernel implementation. The servers' dedicated use removes typical concerns with in-kernel implementations and the use at scale with tens of hundreds of servers warrants significant performance and energy benefit to justify the engineering effort. While *Hippos* is described and evaluated in the context of *Memcached*, it is applicable to any in-memory KV store systems, and its approach can be instrumental in optimizing other network-intensive applications.

4.2 Future Directions

In this dissertation work, we analyzed the workload of *Memcached* at Facebook from various perspectives, but the efficiency of cache replacement policy itself was not discussed. Looking at hit rates has shown that there is room for improvement, especially with the largest pools, ETC and APP. By analyzing the types and distribution of misses, we were able to quantify precisely the potential for additional hits. They are the fraction of GET request that miss because of lack of capacity: 4.1% in ETC (22% of the 18.4% miss rate) and 5.1% in APP (72% of 7.1% misses). This potential may sound modest, but it represents over 120 million GET requests per day per server, with noticeable impact on service latency.

There are two possible approaches to tackle capacity misses: increasing capacity or improving the logic that controls the composition of the cache. The former is expensive and yields diminishing returns (Figure 2.6). Recall that within 6 hours, 96% of GET requests in ETC that would be repeated at all, have already repeated—far above the 81.2% hit rate. On the other hand, non-repeating keys—or those who grow cold and stop repeating—still occur in abundance and take up significant cache space. LRU is not an ideal replacement policy for these keys, which has been demonstrated in all five pools. And since all pools exhibit strong temporal locality, even those pools with adequate memory capacity could benefit from better eviction prediction, for example by reducing the amount of memory (and cost) required by these machines.

One of our directions of future investigation is therefore to replace *Memcached*'s replacement policy. One approach could be to assume that keys that don't repeat within a short time period are likely cold keys, and evict them sooner. Another open question is whether the bursty access pattern of most repeating keys can be exploited to identify when keys grow cold, even if initially requested many times, and evict them sooner.

Though *Hippos* successfully eliminates performance bottlenecks that are imposed by system calls and network stack without modifying any kernel components, the design assumes the DRAM is the only storage media. As flash memory drive, such as solid state drive (SSD), is becoming main stream storage media, one of the directions is to rethink the design that takes the usage of SSD into account. For example, *Hippos* uses hashtable to locate all the data that are stored in memory. However, as the size of SSD is magnitude larger than that of memory, to store the data in SSD means that the size of hashtable would easily reach to tens of Gigabytes, or even larger. So how to optimally manage the metadata of KV cache system is still an open question.

REFERENCES

[1] http://memcached.org/.

[2] http://www.netfilter.org/.

[3] http://en.wikipedia.org/wiki/Network_File_System.

[4] http://en.wikipedia.org/wiki/Nagle's_algorithm.

[5] A System Profiler for Linux. http://oprofile.sourceforge.net/news/.

[6] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *Proceedings of the 7th USENIX Symposium on Networked System Design and Implementation (NSDI 2010)*, San Jose, CA, USA, April 2010.

[7] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation*, Apr. 2010.

[8] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposimu on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, USA, October 2009.

[9] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5:631–645, October 1997.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the ACM SIGMETRICS/Performance*, London, UK, June 2012.

[11] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI 2009)*, Boston, MT, USA, April 2009.

[12] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC 1999)*, Monterey, CA, USA, June 1999.

[13] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-Core Key-Value Store. In *Proceedings of the 2nd International Green Computing Conference (IGCC 2011)*, Orlando, FL, USA, July 2011.

[14] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Power and performance evaluation of memcached on the TILEPro64 architecture. *Sustainable Computing: Informatics and Systems*, 2(2):81–90, 2012.

[15] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[16] Z. Brown. Asynchronous System Calls. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 81 – 86, Ottawa, Canada, June 2007.

[17] A. Chandra and D. Mosberger. Scalability of Linux Event-Dispatch Mechanisms. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '01)*,

Boston, MA, USA, June 2001.

[18] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications, Special Issue on High-Speed*, Apr. 2001.

[19] Consistent Hashing. `http://en.wikipedia.org/wiki/Consistent_hashing/`.

[20] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '10)*, Boston, MA, USA, June 2010.

[21] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent KeyValue Store. In *Proceedings of the VLDB*, Singapore, September 2010.

[22] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proceedings of the SIGMOD 2011*, Athens, Greece, June 2011.

[23] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposimu on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, USA, October 2009.

[24] K. Elmeleegy, A. Chanda, and A. L. Cox. Lazy Asynchronous I/O For Event-Driven Servers. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '04)*, Boston, MA, USA, June 2004.

[25] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A Distributed, Searchable Key-Value Store for Cloud Computing. *Computer Science Department, Cornell University Technical Report*, Dec. 2011.

[26] B. Fan, D. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked System Design and Implementation (NSDI 2013)*, LOMBARD, IL, USA, April 2013.

[27] A. Foundation. `http://cassandra.apache.org/`.

[28] E. Frachtenberg, A. Heydari, H. Li, A. Michael, J. Na, A. Nisbet, and P. Sarti. High-Efficiency Server Design. In *Proceedings of the ACM/IEEE Supercomputing Conference*, Seattle, WA, November 2011.

[29] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[30] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM 2010)*, New Delhi, India, August 2010.

[31] T. Herbert. *RPS: receive packet steering*. `http://lwn.net/Articles/361440/`.

[32] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference*, pages 323–336, Apr. 2005.

[33] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS'02, pages 31–42. ACM, 2002.

[34] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, pages 439–450, Sept. 1994.

[35] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 3rd ACM Symposimu on Cloud Computing*, San Jose, CA, USA, October 2012.

[36] Large Receive Offload. `http://en.wikipedia.org/wiki/Large_receive_offload`.

[37] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 134–143, June 1999.

[38] C. Lever, M. A. Eriksen, and S. P. Molloy. An Analysis of the TUX Web Server. *Technical report*, Nov. 2000.

[39] Z. Li. *GreenDM: A Versatile Tiering Hybrid Drive for the Trade-Off Evaluation of Performance, Energy, and Endurance.* PhD thesis, Computer Science Department, Stony Brook University, May 2014.

[40] Z. Li, M. Chen, A. Mukker, and E. Zadok. On the Trade-Offs among Performance, Energy, and Endurance in a Versatile Hybrid Drive. *ACM Transactions on Storage (TOS)*, 0(0), May 2014. under review.

[41] Z. Li, K. M. Greenan, A. W. Leung, and E. Zadok. Power Consumption in Enterprise-Scale Backup Storage Systems. In *Proceedings of the Tenth USENIX*

Conference on File and Storage Technologies (FAST '12), San Jose, CA, February 2012. USENIX Association.

[42] Z. Li, A. Mukker, and E. Zadok. On the Importance of Evaluating Storage Systems' $Costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[43] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-eficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Oct. 2011.

[44] Linux HTTP Accelerator. `http://www.fenrus.demon.nl/`.

[45] K. Magoutis, M. Seltzer, and E. Gabber. The Case Against User-Level Networking. In *Third Workshop on Novel Uses of System Area Networks (SAN-3)*, Madrid, Spain, Feburary 2004.

[46] T. Marian. Operating Systems Abstractions for Software Packet Processing in Datacenters. *PhD Dissertation, Cornell University*, Aug. 2010.

[47] P. McKenney and J. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. *Parallel and Distributed Computing and Systems*, pages 509 – 518, Oct. 1998.

[48] P. E. McKenney and J. Walpole. Exploiting deferred destruction: An Analysis of Read-Copy-Update Techniques in Operating System kernels. *PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University*, 2004.

[49] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHASH: A Cache-Partitioned Hash Table. *MIT-CSAIL-TR-2011-051*, Nov. 2011.

[50] Microsoft. Scalable Networking: Eliminating the Receive Processing Bottleneck-Introducing RSS. *WinHEC 2004*, Apr. 2004.

[51] MSRPC. `http://en.wikipedia.org/wiki/Microsoft_RPC`.

[52] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked System Design and Implementation (NSDI 2013)*, LOMBARD, IL, USA, April 2013.

[53] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the Performance of Web Server Architectures. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisboa, Portugal, March 2007.

[54] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, Bern, Switzerland, April 2012.

[55] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, Paris, France, April 2010.

[56] C. C. R. Policy. `http://en.wikipedia.org/wiki/Page_replacement_algorithm#Clock`.

[57] Project Voldermort. A distributed key-value storage system.. `http://project-voldemort.com`.

[58] N. Provos and C. Lever. Scalable network i/o in linux. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC 2000)*, San Diego, CA, USA, June 2000.

[59] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Cassyopia: Compiler assisted system optimization. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS 2003)*, Lihue, Hawaii, USA, May 2003.

[60] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '12)*, Boston, MA, USA, June 2012.

[61] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC 2001)*, Oakland, CA, USA, November 2001.

[62] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack - Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '10)*, Boston, MA, USA, June 2010.

[63] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[64] L. Soares and M. Stumm. Exception-Less System Calls for Event-Driven Servers. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '11)*, Portland, OR, USA, June 2011.

[65] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '12)*, Boston, MA, USA, June 2012.

[66] The SPINOperating System. `http://www-spin.cs.washington.edu/`.

[67] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '11)*, Portland, OR, USA, June 2011.

[68] L. R. Used. `http://en.wikipedia.org/wiki/Least_Recently_Used#LRU`.

[69] V. R. Vasudevan. *Energy-Efficient Data-intensive Computing with a Fast Array of Wimpy Nodes*. PhD thesis, Carnegie Mellon University, Oct. 2011.

[70] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposimu on Operating Systems Principles (SOSP 1995)*, Copper Mountain, Colorado, USA, December 1995.

[71] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well conditioned, scalable internet services. In *Proceedings of the 19th ACM Symposimu on Operating Systems Principles (SOSP 2001)*, Chateau Lake Louise, Canada, October 2001.

[72] P. Willmann, S. Rixner, and A. L. Cox. An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC '06)*, Boston, MA, USA, June 2006.

ABSTRACT

BUILDING A SCALABLE AND HIGH-PERFORMANCE KEY-VALUE STORE SYSTEM

by

YUEHAI XU

December 2014

Advisor: Dr. Song Jiang

Major: Computer Engineering

Degree: Doctor of Philosophy

Contemporary web sites can store and process very large amounts of data. To provide timely service to their users, they have adopted key-value (KV) stores, which is a simple but effective caching infrastructure atop the conventional databases that store these data, to boost performance. Examples are Facebook, Twitter and Amazon. As yet little is known about the realistic workloads outside of the companies that operate them, this dissertation work provides a detailed workload study on Facebook's *Memcached*, which is one of the world's largest KV deployment. We analyze the *Memcached* workload from the perspective of server-side performance, request composition, caching efficacy, and key locality. The observations presented in this dissertation lead to several design insights and new research direction for KV stores – *Hippos*, a high-throughput, low-latency, and energy-efficient KV-store implementation.

Long considered an application that is memory-bound and network-bound, recent KV-store implementations on multicore servers grow increasingly CPU-bound instead. This limitation often leads to under-utilization of available bandwidth and

poor energy efficiency, as well as long response times under heavy load. To address these issues, *Hippos* moves the KV-store into the operating system's kernel and thus removes most of the overhead associated with the network stack and system calls. It uses the *Netfilter* framework to quickly handle UDP packets, removing the overhead of UDP-based GET requests almost entirely. Combined with lock-free multithreaded data access, *Hippos* removes several performance bottlenecks both internal and external to the KV-store application.

Hippos is prototyped as a Linux loadable kernel module and evaluated it against the ubiquitous *Memcached* using various micro-benchmarks and workloads from Facebook's production systems. The experiments show that *Hippos* provides some 20–200% throughput improvements on a 1Gbps network (up to 590% improvement on a 10Gbps network) and 5–20% saving of power compared with *Memcached*.

AUTOBIOGRAPHICAL STATEMENT

YUEHAI XU

Yuehai Xu is a graduate student in the Department of Electrical and Computer Engineering at Wayne State University. He received his B.S. degrees in College of Software Engineering at Southeast University, Nanjing, China in 2004.

His research interests include file system, operating system, and storage system. He has published five international conference papers, including one published at USENIX Conference on File and Storage Technologies (FAST'11), and one published at ACM SIGMETRICS/performance (sigmetrics'12). He also has published three journal papers, two of which were at ACM Transcations on Storage, and one at IEEE Internet Computing.

He was a lab instructor for the undergraduate course Digital Circuits for two semesters and had been a teaching assistant for another undergraduate course Basic Engineering I for three semesters.