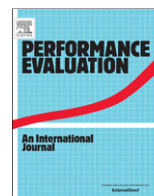
Performance Evaluation 79 (2014) 24–37

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva



Building a high-performance key–value cache as an energy-efficient appliance

Yuehai Xu [a], Eitan Frachtenberg [b], Song Jiang [a,*]

[a] ECE Department, Wayne State University, Detroit, MI 48202, USA
[b] Facebook, Menlo Park, CA 94025, USA

ARTICLE INFO

Article history:
Available online 11 July 2014

Keywords:
Key–value cache
Netfilter
Clock cache policy
Skb buffer reuse

ABSTRACT

Key–value (KV) stores have become a critical infrastructure component supporting various services in the cloud. 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, we present *Hippos*, a high-throughput, low-latency, and energy-efficient key–value store implementation. *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. *Hippos* 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.

We prototyped *Hippos* 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 1 Gbps network (up to 590% improvement on a 10 Gbps network) and 5%–20% saving of power compared with *Memcached*.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Key–value stores play a critical role in the improvement of service quality and user experience in many large-scale websites. Examples include *Voldermort* [1] at LinkedIn, *Cassandra* [2] at Apache, and *Memcached* [3,4] at Facebook. KV stores have received significant research and industry attention recently as high-throughput distributed caches [4,5]. 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 dataset 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 [6,4]. As an essential component in a datacenter's infrastructure, a KV cache plays a critical role in improving service quality and lowering operational cost.

* Corresponding author.
E-mail addresses: yhxu@wayne.edu (Y. Xu), eitan@frachtenberg.com (E. Frachtenberg), sjiang@wayne.edu (S. Jiang).

http://dx.doi.org/10.1016/j.peva.2014.07.002
0166-5316/© 2014 Elsevier B.V. All rights reserved.

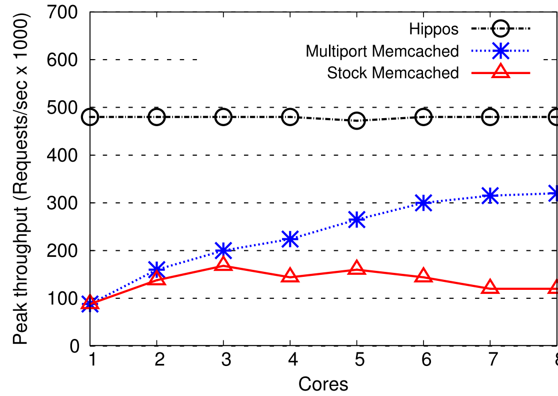


Fig. 1. Peak throughput of *Memcached*s in terms of requests per second with different number of enabled cores. In the figure, *Stock Memcached* refers to the open-source *Memcached* running as an application on Linux; *Multiport Memcached* refers to the optimized *Memcached* with multiport support. *Hippos* refers to the proposed in-kernel KV-cache implementation.

1.1. KV Cache: not CPU bound?

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. 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 on a key to locate the server that owns the requested data. 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 [7].

To investigate whether the KV cache is indeed bottlenecked by its CPU, we chose *Memcached* [3] 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 was observed at Facebook, one of the world's largest *Memcached* deployments [6]. As reported in the workload study [6], the ratio of GET to SET requests can be very high, sometimes exceeding 30:1. The key size is typically smaller than 30 B, and more than half of the value sizes can be smaller than 20 B in some traces. 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 [4]. 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 1 Gbps network. All machines used an Intel 8-core Xeon processor (more system details in Section 3). As in the rest of this paper, peak throughput is reported as the highest throughput observed while the corresponding mean request latency is kept under 1 ms, where a request's latency is measured by the client as total round-trip time.

1.2. KV cache: a CPU-demanding application!

We use the latest open-source *Memcached* version [3], 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[1] 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 serious socket lock contention [4], 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 [4]. 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 [8].

Fig. 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

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

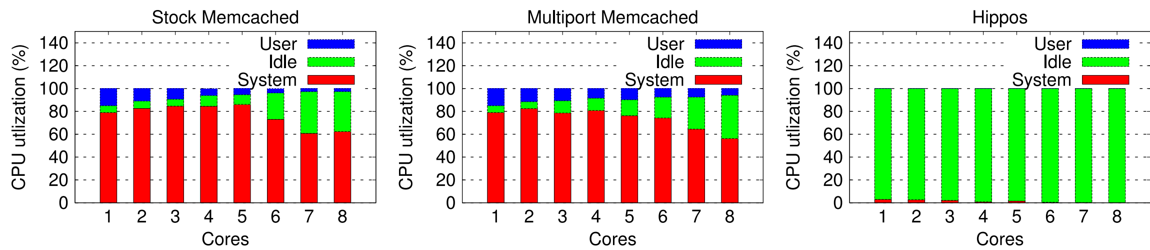


Fig. 2. CPU time distribution on user-level code, kernel (system) code and being idle when one of three *Memcached*' implementations runs with various number of cores at their respective peak throughout.

conclusion that *Multiport Memcached* is scalable on multicore CPUs without major changes to the kernel [8]. However, this may also demonstrate that the demand on CPU cores does not saturate 1 Gbps network card even with all eight cores enabled.

Fig. 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 1 ms threshold. Fig. 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.

1.3. Does optimizing network stack help?

Since we know that a CPU-demanding *Memcached* spends most of its time in the kernel, in particular on the network stack, our first mitigation approach is to reuse existing network techniques to reduce the CPU time. To this end, we examined *Multiport Memcached* with OProfile [9] to see how the CPU cycles are used across the network stack. Table 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 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 [10–12,5]. 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 [14], reducing the number of packets using jumbo frames [15], and mitigating interrupts [16,17], 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 [18–20]. To reduce overhead due to unnecessary sharing of network control states in a multicore system, *IsoStack* [19] 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 1, and could hardly benefit from this technique. Recent work (*Netmap* [21]) 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* [22] rely on user level networking [23] 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

Table 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*.

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 [13], *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%

NIC because the amount of NIC resources demanded for managing user-level connection endpoints increases linearly with the number of clients simultaneously issuing requests [23,24]. The number can be substantial in *Memcached* service [6].

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 [4], it is time to revisit the conventional wisdom that this network-intensive class of applications are improved only through optimization of the network stack.

1.4. Obtaining the data closer to the NIC

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. Our objective 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.

In this paper we describe *Hippos*, a KV cache that uses a hook provided in the *Netfilter* framework [25] 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. Foreshadowing a more comprehensive evaluation, Figs. 1 and 2 show the peak throughput and CPU time usages for *Hippos*. As shown, with only a single core, *Hippos* can reach a peak throughput much higher than those of the other user-level *Memcached*s running on eight cores. In Fig. 1, *Hippos*'s throughput is limited only by the 1 Gbps NIC in which there is only one hardware interrupt support. In addition, the CPU remains mostly idle, opening the door to a substantial energy saving.

In summary we make the following contributions in this paper.

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 [26] 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 1 Gbps network (up to 590% improvements on a 10 Gbps 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 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.

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

Table 2
The observation positions.

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*

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.

2.1. Targeted workloads

Hippos is motivated by the suboptimal performance of *Stock Memcached* under realistic workloads, taken from Facebook's workload study [6]. 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 (16 B and 21 B) and virtually only one value size (2 B). For the other four pools, APP, ETC, VAR, and SYS, the 99% percentile key sizes are 45 B, 80 B, 30 B, and 45 B, respectively. Almost all GET requests can be held in a single UDP packet. Their respective 99% percentile value sizes are 450 B, 512 B, 200 B, and 640 B. 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" [6].

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.

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 Fig. 3). 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 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 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. Fig. 5 shows that CPU utilization at various observation positions with one core. Fig. 4 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 shown, at positions 1 and 2 the CPUs are almost all idle and the latency is minimal even when the arrival rate reaches 800 K packets per second. However, at position 3, system time starts to become substantial and even dominating when the arrival rate reaches 800 K packets per second, and the latency skyrockets from 10 μs to over 200 μs when the rate is beyond 480 K 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 320 K 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 Fig. 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

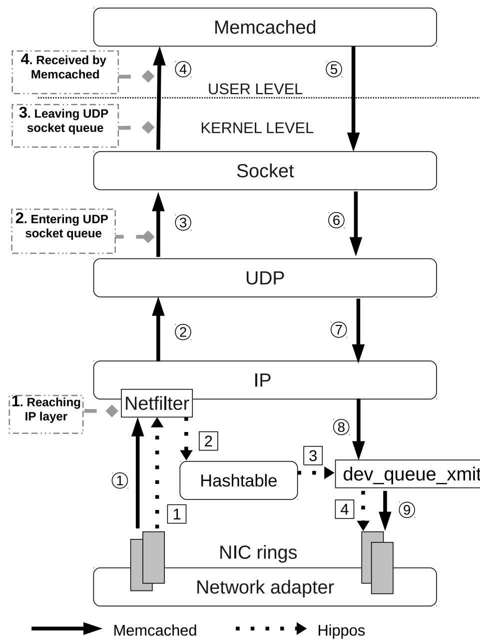


Fig. 3. The paths for a UDP GET request to travel in the network stack and *Memcached* (or respectively *Hippos*).

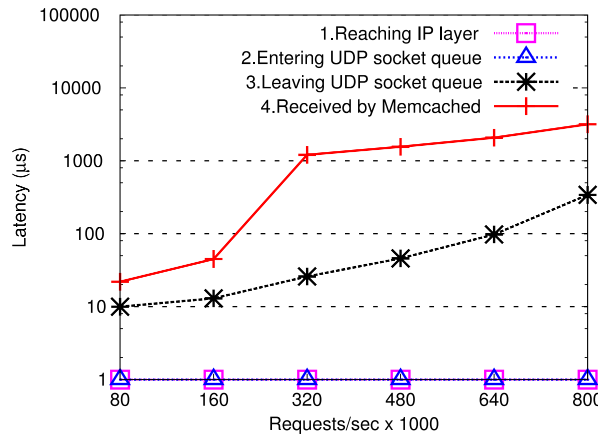


Fig. 4. Latency observed at various observation positions in the network stack with different request arrival rates and one core in use. The latency of a packet is measured as the duration between when it is received (*netif_receive_skb()*) and when it reaches a particular observation position. Note that the *Y* axis is on the logarithmic scale.

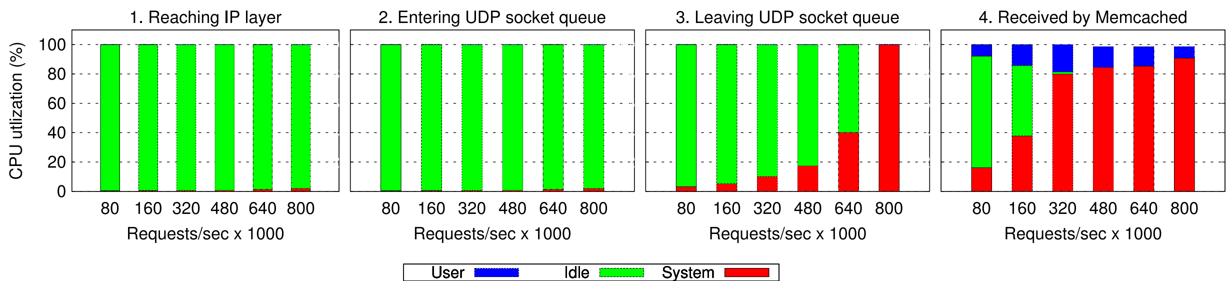


Fig. 5. CPU utilization at various observation positions in the network stack with different request arrival rates when one core is in use.

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 *Netfilter* framework [25] 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

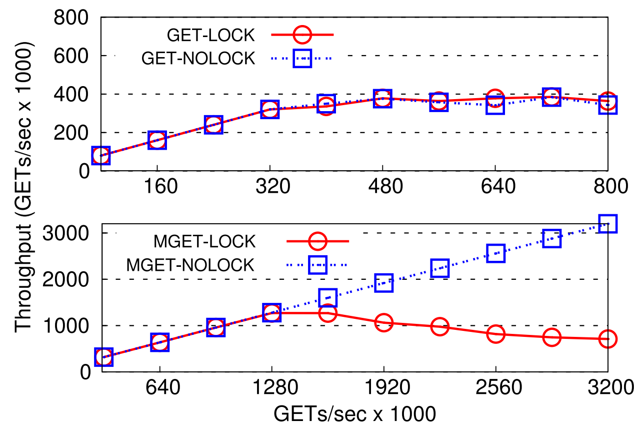


Fig. 6. *Memcached* throughput (number of GETs processed per second) under various GET request arrival rates. The GET requests arrive either in the one-request-in-a-packet format (GET) or in the multiple-requests-in-a-packet format (MGET). For MGET, the packet arrival rate is 320 K packets per second, and we change the number of requests in a packet. The lock may be applied (*LOCK*) or not (*NOLOCK*).

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, such as UDP layer.[2] 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 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 Fig. 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 Fig. 6, 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 320 K 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 320 K packets per second. Before the workload increases to 1280 K 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

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

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 [27,28]. 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.

2.4. Other design considerations

In the design of *Hippos*, a few other considerations and alternatives are worth discussing, as follows.

2.4.1. 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 [17] 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 [29] 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.

2.4.2. 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.

2.4.3. 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 [6], 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. Performance evaluation

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 2.33 GHz Xeon CPU, 64 GB DRAM, and Intel PRO/1000 1 Gbps NIC, running Linux 3.5.0. A server node is connected with another eight client nodes of identical configuration. The use of a 1 Gbps NIC, which is embedded in the motherboard, is quite common for clusters in large-scale data centers [30]. It provides a raw bandwidth larger than what is demanded by *Memcached* traffic discussed in the study of Facebook *Memcached* traces [6], which are also used in our evaluation. For a KV-cache workload dominated by small keys and values, whose combined sizes are less than 1 KB, 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 10 Gbps network by using the dual-port Intel 82599 10 Gb 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,

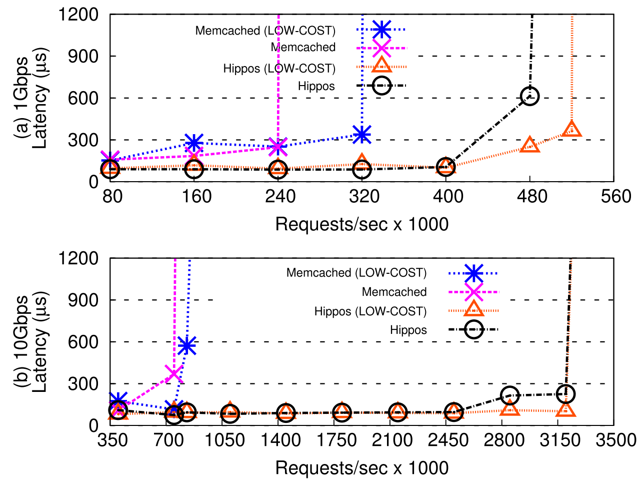


Fig. 7. Request latencies for *Memcached* and *Hippos* with increasing request arrival rate in the 1 Gbps network (a) and in the 10 Gbps network (b). For each system, latencies for a low-cost setup (LOW-COST) are also reported, in which requests are for non-existent keys in a hash table not protected by locks.

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 [4], 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 pre-populate the cache before each run and issue requests with random keys from the cache.

3.1. Micro-benchmarks

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.

3.1.1. 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 20 B and in the reply packet the value size is also 20 B. Fig. 7(a) and (b) show the latencies with the increasing request rate for 1 Gbps and 10 Gbps 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.

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 1 Gbps-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 (480 K Req/s vs. 240 K Req/s). The performance trend for the 10 Gbps 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 10 Gbps network is that *Hippos* shifts the throughput bottleneck from the CPU to the network. Accordingly a 10 Gbps network exposes more of *Hippos*'s potential. The smaller difference is because that in the 10 Gbps 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 10 Gbps NIC and for running application threads [31].

3.1.2. 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 packets' *sk_buff*, the operations' cost is proportional to the request arrival

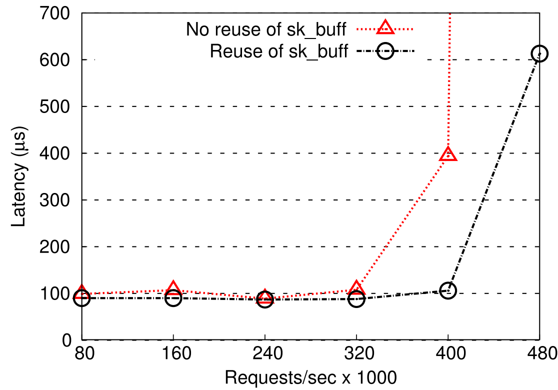


Fig. 8. Request latencies for *Hippos* with and without using the *sk_buff* reuse optimization when the request arrival rate increases.

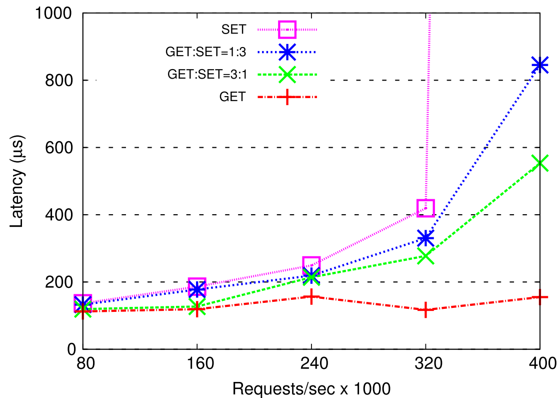


Fig. 9. Latencies for workloads with different mixes of GETs and SETs and different request rates.

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 20 B key and searching for a 20 B value, in the 1 Gbps network. Fig. 8 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 400 K req/s to 480 K req/s).

3.1.3. 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. Fig. 9 shows latencies for workloads with different mixes of GETs and SETs in the 1 Gbps network. With low request rate (80 K 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 320 K reqs/s, the workload with all SETs sees latencies jump beyond 1 ms. 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.

3.2. Replaying Facebook's traces

We replayed Facebook's production-representative *Memcached* traces on *Hippos* with both 1 Gbps and 10 Gbps 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 [6]. Here we summarize the distribution of requests in each trace in Table 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 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*. Fig. 10 (a) and (b) show the peak throughput received by *Memcached* and *Hippos* for the 1 Gbps and 10 Gbps networks, respectively.

Table 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.

	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 4
Average request latency and power consumption of *Memcached*, and respective changes made by *Hippos* in percentage for the five traces with the 1 Gbps and 10 Gbps networks (only 10 Gbps explicitly indicated). Latency larger than 1 ms 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.

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	226 μs	−11%	+11%	−15%
USR 10 Gbps	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	471 μs	−14%	−11%	−9%
ETC 10 Gbps	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	386 μs	+42%	−37%	−10%
APP 10 Gbps	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	174 μs	163 μs	N/A	−9%
VAR 10 Gbps	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	230 μs	231 μs	N/A	−6%
SYS 10 Gbps	232	992	978	N/A	181	−44%	−37%	N/A	−13%

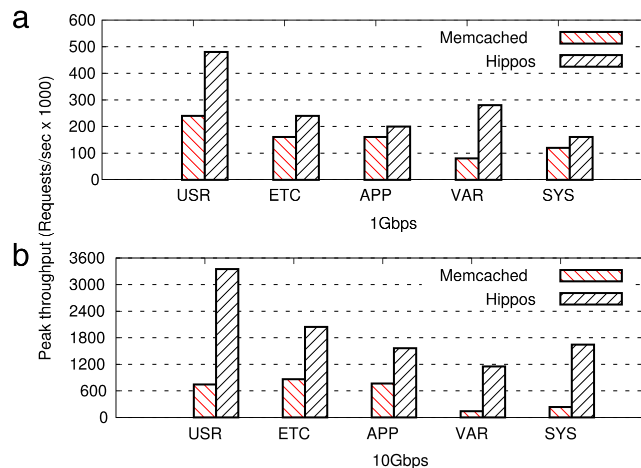


Fig. 10. Peak throughput received by *Memcached* and *Hippos* for each of the five Facebook's traces. The throughput is collected under the condition that the corresponding average request latency does not exceed 1 ms.

From the experimental results we gathered several interesting observations. First, for the 1 Gbps network *Hippos* achieves the most impressive improvements for traces USR and VAR, each for a different reason. According to Table 3, USR consists of almost entirely GETs (99.7%). Both request packets (with only 16 B and 21 B keys) and reply packets (with virtually only 2 B 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 120 K 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.

Second, ETC, APP, and SYS have relatively moderate improvements in the 1 Gbps 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 270 B. ETC also has a significant portion of large value size, even a few of around 1 MB. 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 10 Gbps NIC is used, it breaks the limit and gives *Hippos* a larger room for improvement. As shown in Fig. 10(b), the peak throughput of ETC, APP, and SYS is improved by 140%, 100%, and 590%, respectively, in the 10 Gbps 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 80 K reqs/s, 160 K reqs/s, and 240 K reqs/s for ETC in the 1 Gbps 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 1 ms, *Hippos* can be still appealing with its advantage on power saving across the different request rates.

4. Related work

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

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 [7,8]. 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 [32], 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 [33,4,26] 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 [17], RPS [13], and RSS [14] 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.

Moving applications into the kernel. Migration of services that are considered integral to a server's operation into the kernel has been in practice for other purposes. *SPIN* [34] 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.

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 [21]. 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 [35]. 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.

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 [36]. 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 [8]. *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.

5. Conclusions

We have described the design and implementation of *Hippos*, an in-kernel key–value 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.

Acknowledgments

This work was supported by US National Science Foundation under CAREER CCF 0845711 and CNS 1217948. The authors are also thankful to the anonymous reviewers for their constructive comments and suggestions.

References

[1] Project Voldermort. A distributed key-value storage system. http://project-voldemort.com.
[2] Foundation, A. http://cassandra.apache.org/.
[3] http://memcached.org/.
[4] 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, V. Venkataramani, Scaling memcache at facebook, in: Proc. 10th USENIX Symposium on Networked System Design and Implementation (NSDI 2013), LOMBARD, IL, USA, April 2013.
[5] P. Stuedi, A. Trivedi, B. Metzler, Wimpy nodes with 10GbE: Leveraging one-sided operations in soft-rdma to boost memcached, in: Proc. USENIX Annual Technical Conference (USENIX ATC '12), Boston, MA, USA, June 2012.
[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny, Workload analysis of a large-scale key-value store, in: Proc. ACM SIGMETRICS/Performance, London, UK, June 2012.
[7] M. Berezecki, E. Frachtenberg, M. Paleczny, K. Steele, Power and performance evaluation of memcached on the TILEPro64 architecture, Sustainable Computing: Inform. Syst. 2 2 (2012) 81–90.
[8] S. Boyd-Wickizer, A.T. Clements, Y. Mao, A. Pesterev, M.F. Kaashoek, R. Morris, N. Zeldovich, An analysis of linux scalability to many cores, in: Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10), Vancouver, Canada, October 2010.
[9] A system profiler for Linux. http://oprofile.sourceforge.net/news/.
[10] http://en.wikipedia.org/wiki/Network_File_System.
[11] Linux HTTP accelerator. http://www.fenrus.demon.nl/.
[12] MSRPC. http://en.wikipedia.org/wiki/Microsoft_RPC.
[13] T. Herbert, RPS: receive packet steering. http://lwn.net/Articles/361440/.
[14] Microsoft. Scalable networking: eliminating the receive processing bottleneck-introducing RSS. WinHEC 2004 (Apr. 2004).
[15] J.S. Chase, A.J. Gallatin, K.G. Yocum, End-system optimizations for high-speed TCP, IEEE Communications, Special Issue on High-Speed (2001).
[16] Large receive offload. http://en.wikipedia.org/wiki/Large_receive_offload.
[17] J. H. Salim, R. Olsson, A. Kuznetsov, Beyond softnet, in: Proc. USENIX Annual Technical Conference (USENIX ATC 2001), Oakland, CA, USA, November 2001.
[18] A. Pesterev, N. Zeldovich, R. T. Morris, Locating cache performance bottlenecks using data profiling, in: Proc. ACM SIGOPS/EuroSys European Conference on Computer Systems, Paris, France, April 2010.
[19] L. Shalev, J. Satran, E. Borovik, M. Ben-Yehuda, IsoStack—Highly efficient network processing on dedicated cores, in: Proc. USENIX Annual Technical Conference (USENIX ATC '10), Boston, MA, USA, June 2010.
[20] P. Willmann, S. Rixner, A.L. Cox, An evaluation of network stack parallelization strategies in modern operating systems, in Proc. USENIX Annual Technical Conference (USENIX ATC '06), Boston, MA, USA, June 2006.

[21] L. Rizzo, netmap: A novel framework for fast packet I/O, in: Proc. USENIX Annual Technical Conference (USENIX ATC '12), Boston, MA, USA, June 2012.
[22] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, A. Vahdat, Chronos: Predictable low latency for data center applications, in: Proc. 3rd ACM Symposimu on Cloud Computing, San Jose, CA, USA, October 2012.
[23] T. von Eicken, A. Basu, V. Buch, W. Vogels, U-Net: a user-level network interface for parallel and distributed computing, in: Proc. 15th ACM Symposimu on Operating Systems Principles (SOSP 1995), Copper Mountain, Colorado, USA, December 1995.
[24] K. Magoutis, M. Seltzer, E. Gabber, The case against user-level networking, in: Third Workshop on Novel Uses of System Area Networks (SAN-3), Madrid, Spain, Feburary 2004.
[25] http://www.netfilter.org/.
[26] J. Triplett, P. E. McKenney, J. Walpole, Resizable, scalable, concurrent hash tables via relativistic programming, in: Proc. USENIX Annual Technical Conference (USENIX ATC '11), Portland, OR, USA, June 2011.
[27] P. McKenney, J. Slingwine, Read-copy update: Using execution history to solve concurrency problems. Parallel and Distributed Computing and Systems (Oct. 1998), pp. 509–518.
[28] P. E. McKenney, J. Walpole, Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels (Ph.D. thesis), OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.
[29] http://en.wikipedia.org/wiki/Nagle's_algorithm.
[30] E. Frachtenberg, A. Heydari, H. Li, A. Michael, J. Na, A. Nisbet, P. Sarti, High-efficiency server design, in: Proc. ACM/IEEE Supercomputing Conference, Seattle, WA, November 2011.
[31] A. Pesterev, J. Strauss, N. Zeldovich, R. T. Morris, Improving network connection locality on multicore systems, in: Proc. ACM SIGOPS/EuroSys European Conference on Computer Systems, Bern, Switzerland, April 2012.
[32] Z. Metreveli, N. Zeldovich, M. F. Kaashoek, CPHASH: A cache-partitioned hash table. MIT-CSAIL-TR-2011-051 (Nov. 2011).
[33] B. Fan, D. Andersen, M. Kaminsky, MemC3: Compact and concurrent memcache with dumber caching and smarter hashing, in: Proc. 10th USENIX Symposium on Networked System Design and Implementation (NSDI 2013), LOMBARD, IL, USA, April 2013.
[34] The SPINOperating System. http://www-spin.cs.washington.edu/.
[35] T. Marian, Operating systems abstractions for software packet processing in datacenters Ph.D. Dissertation, Cornell University, 2010.
[36] S. Han, K. Jang, K. Park, S. Moon, PacketShader: a GPU-accelerated software router, in: Proc. of ACM Special Interest Group on Data Communication (SIGCOMM 2010), New Delhi, India, August 2010.

Yuehai Xu is a Ph.D. student in the Electrical and Computer Engineering Department at Wayne State University. His research interests include operating systems, file and storage systems. Xu has a Bachelor's degree in software engineering from Southeast University.

Eitan Frachtenberg is a Data Scientist at Facebook, analyzing social behavior on large-scale datasets. His research interests include data mining, performance evaluation and optimization, Web technologies, and computer architecture. He obtained his Ph.D. in Computer Science from Hebrew University, and is a senior member of the IEEE.

Song Jiang is an Associate Professor at Wayne State University. His research interests include file and storage systems, operating systems and high-performance computing. Jiang has a Ph.D. in Computer Science from the College of William and Marry.