2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing

KV-Cache: A Scalable High-Performance Web-Object Cache for Manycore

Daniel Waddington, Juan Colmenares, Jilong Kuang
Computer Science Lab
Samsung Research America - Silicon Valley
San Jose, CA 95134
{d.waddington,juan.col,jilong.kuang}@samsung.com

Fengguang Song
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN 46202
fgsong@cs.iupui.edu

***Abstract*—Latency and cost of Internet-based services are driving the proliferation of web-object caching. *Memcached*, the most broadly deployed web-object caching solution, is a key infrastructure component for many companies that offer services via the Web, such as Amazon, Facebook, LinkedIn, Twitter, Wikipedia, and YouTube. Its aim is to reduce service latency and improve processing capability on back-end data servers by caching immutable data closer to the client machines. Caching of key-value pairs is performed solely in memory.**

In this paper, we present a novel design for a high-performance web-object caching solution, *KV-Cache*, that is Memcache-protocol compliant. Our solution, based on TU Dresden's Fiasco.OC microkernel operating system, offers scalability and performance that significantly exceeds that of its Linux-based counterpart. KV-Cache's highly optimized architecture benefits from truly "absolute" zero copy by eliminating any software memory copying at the kernel level or in the network stack, and only performing direct memory access (DMA) for each transmit and receive path. We report experimental results for the current prototype running on an Intel E5-based 32-core server platform. Our results show that KV-Cache offers significant performance advantages over optimized Memcached on Linux for commodity x86 server hardware.

I. INTRODUCTION

Minimizing service response time (*i.e.,* latency) is important to global Internet-based service providers, such as Amazon, Facebook, Google, Samsung, and Twitter. In particular, it is a principal concern of social-networking and big-data infrastructure deployment. Service response time is a performance metric providers use to differentiate themselves from the competition in order to help retain existing users and also attract new ones. The general recommendation is that systems that respond to user actions in less than 100 ms make the user experience more fluid and natural [1] – this is the total round-trip time budget for end-system and network processing.

Meeting this latency budget in large-scale deployments is inherently difficult. Providers look to parallelization and caching as two key tactics in keeping end-to-end latency at a minimum. As part of this general strategy, social networking companies (*e.g.,* Facebook and Twitter) are turning to web-object caching as a means to serve a very large number of clients with a satisfactory end-to-end latency. Furthermore, web-object caching services have become an integral part of many Platform-as-a-Service (PaaS) offerings from prominent providers such as Google, Microsoft, Amazon and Heroku.

Memcached [2] is one of the most popular web-object caching solutions. Prior work on optimizing Memcached has mainly focused on lock removal and data-structure optimization for scalable concurrent access as is required for a "scale-up" deployment (*i.e.,* increasing the number of on-chip cores). On commodity server hardware, Intel researchers [3] report a peak performance of ∼3.1 million requests per second (RPS) by using alternative data structures and locking designs as compared to the vanilla Memcached implementation. Other work has explored hardware customization as a means to improve performance and energy efficiency for Memcached [4], [5].

The rest of the paper presents KV-Cache, a web-object caching solution conforming to the Memcache protocol. KV-Cache exploits a software *absolute zero-copy* approach and aggressive customization to offer significant performance improvements over existing Memcached-based solutions. Experimental data from our current prototype shows that KV-Cache is able to provide more than 2x the capacity and performance over Intel's optimized versions of Memcached [3] and more than 24x capacity of off-the-shelf open source Memcached.

II. KV-CACHE DESIGN OVERVIEW

In the development of KV-Cache we take a holistic approach to optimization by explicitly addressing performance and scalability throughout both the application and operating-system layers. Our design consists of the following key attributes:

1) *Microkernel-based*: Our solution uses the Fiasco.OC L4 microkernel-based operating system. This provides improved resilience and scaling as compared to monolithic kernels (*e.g.,* Linux) and allows aggressive customization of the memory and network sub-systems.
2) *Scalable concurrent data structures*: Data partitioning and NUMA-aware memory management are used to maximize performance and minimize lock contention. Critical sections are minimized and protected with fair and scalable spin-locks.
3) *Direct copy of network data to user-space via direct memory access (DMA)*: Data from the network is asynchronously DMA-transferred directly from the network interface card (NIC) to level-3 cache, where data becomes

F. Song performed this research during his position at Samsung Research America – Silicon Valley.

CFP13UCC-USB/13 $26.00 © 2013 IEEE
DOI 10.1109/UCC.2013.34

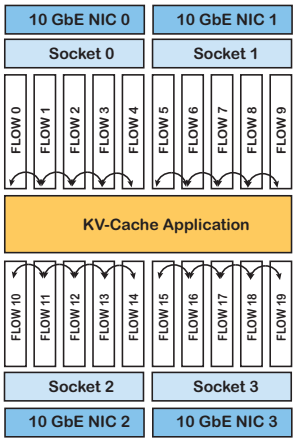

Fig. 1. KV-Cache's column architecture.

directly accessible to KV-Cache in user-space. Only network protocol headers are copied.

4) *Packet-based value storage*: Data objects (*i.e.,* key-value pairs) are maintained in the originally received on-the-wire packet format. IP-level reassembly and fragmentation are avoided.
5) *NIC-level flow direction*: Use of 10-Gbps NICs that support flow steering across multiple receive queues. Multiple receive and transmit queues, in combination with Message Signaled Interrupts (MSI-X), allow us to distribute I/O processing across multiple CPU cores.
6) *Work-stealing for non-uniform request distributions*: Work- stealing techniques are used to load-balance across work queues when the request distribution is non-uniform across the key space.

The KV-Cache architecture is based on a column-centric design whereby the execution "fast path" flows up and down a single column (see Figure 1). A "slow path" is executed when work-stealing occurs as a result of request patterns with a non-uniform key distribution. This design improves performance by minimizing cross-core cache pollution and remote NUMA memory accesses, as well as providing an effective design for performance scaling by dynamically adjusting the total number of active columns.

Memcached defines both text and binary protocols [6]. To maximize performance, our implementation uses the binary protocol. Table I summarizes the five key Memcache protocol commands implemented by KV-Cache.

TABLE I. KEY COMMANDS OF MEMCACHE PROTOCOL

Command	Input	Output
`GETK`	Key	Status,[Key],[Value]
`SET,ADD, REPLACE`	Flags, Expiration, Key, [Value]	Status, Data Version
`DELETE`	Key	Status

III. OPERATING-SYSTEM FOUNDATION

A key element of our approach is to aggressively customize the operating system (OS) and other software in the system stack. Our solution is designed as an "appliance" with a specific task in hand.

A. L4 Fiasco.OC Microkernel

We implement our solution using a microkernel operating system because of its inherent ability to decentralize and decouple system services – this is a fundamental requirement for achieving scalability. A number of other research efforts to address multicore and manycore scalability have also taken the microkernel route [7], [8], [9], [10].

The base of our system is the Fiasco.OC L4 microkernel from TU Dresden [11]. Fiasco.OC is a third generation microkernel that provides real-time scheduling and object capabilities [12]. Out-of-the box Fiasco.OC supports both x86 (32 and 64 bit) as well as ARM platforms. In this work we keep the kernel largely as-is. The kernel itself is reasonably well-designed for multicore scalability. For instance, per-core data structures, static CPU binding and local manipulation of remote threads [13] are integrated into the design.

We make minor enhancements to the kernel to provide multicore features required by our solution. These include: 1) deterministic mapping of logical-to-physical core identifiers, 2) support for socket, core and hyper threading topology discovery, and 3) new system calls for ACPI and CPU information retrieval.

B. Genode OS Microkernel User-land

In a microkernel solution the bulk of the operating system exists outside of the kernel in user-land, in what is termed the *personality*. Memory management, inter-process communication (IPC), network protocol stacks, interrupt-request (IRQ) handling, and I/O mapping are all implemented as user-level processes with normal privileges (*e.g.,* x86 ring 3).

Our solution uses the Genode OS framework [14] as the basis for the personality. The Genode personality is principally aimed at multi-L4 kernel portability. However, what is of more interest to our broader vision is the explicit partitioning and allocation of resources [15]. The Genode architecture ensures tight admission control of memory, CPU and other resources in the system as the basis for limiting failure and attack propagation. Quota and admission control is hierarchically managed by each process's parent. This hierarchical design allows each application's Trusted Computing Base (TCB) to be constrained in scope, thus furthering security and robustness.

C. S-Core: Genode Scalability Enhancements

Development of the Genode OS framework by Genode Labs is still an on-going effort. To date the group has predominantly focused on getting the fundamentals right in the context of embedded and PC-based uniprocessor platforms. Consequently, for clean design the Genode core process, which is single-threaded, becomes a serialization bottleneck for services and page-fault handling when multi-threaded processes execute.

Because the KV-Cache application requires high-performance multiprocessing server hardware, it was necessary to enhance the basic Genode OS framework. Most of the improvements have been realized through an additional core-like process, called *S-Core*, that effectively forms the root of a scalable subsystem on which KV-Cache rests. An important aspect of S-Core is that it is given the same base

capabilities as the Genode core process so that it can interact directly with the kernel (as opposed to via the Genode core process).

The S-Core server provides the following key features:

1) *Direct IRQ/MSI Handling* - allowing applications to directly handle IRQ signals from the kernel (Genode conventionally handles IRQs via the core process). Support for Message Signaled Interrupts (MSIs) is also included.
2) *System Topology Discovery* - discovery and query of socket, core, hyper-threading and memory topologies (through ACPI and `cpuid` methods).
3) *NUMA-aware Memory Management* - allocation of physical memory according to NUMA latency costs.
4) *Eager Memory Mapping* - support for pre-mapping of physical-to-virtual memory to avoid non-scalable page-fault handling.
5) *Shared Memory Management* - allocation of virtual memory space that is reserved for shared memory allocations.

These features are key to providing effective scalability of the KV-Cache appliance.

IV. KV-CACHE ARCHITECTURE

Our architecture uses a novel zero-copy approach that significantly improves the performance of the appliance. Unlike other systems that employ zero-copy memory strategies that still incur copies within the kernel, our solution provides truly "absolute zero" copy by customizing the IP protocol stack for the application's needs. This aggressive application-specific tailoring is a fundamental tenet of our approach and is one of the main reasons for choosing a microkernel-based solution.

In the context of KV-Cache, zero-copy means that:

- Network packets are transferred directly via DMA from the NIC device to user-space memory, and vice versa.
- Ingress Layer-2 packets (*i.e.,* Ethernet frames) are not assembled (defragmented) into the larger IP frames.
- Each cached object (key-value pair) is maintained in memory as a linked-list of packet buffers.
- `GET` responses are assembled from packet fragments stored in memory by transferring directly out of user-space (via DMA).

We believe that our absolute zero-copy approach allows us to effectively double the performance of the system.

A. Network Subsystem

Web-object caching is an IO-bound application; therefore, performance of the network protocol stack and device driver is critical. KV-Cache does not use the LWIP (Lightweight IP) protocol stack or any network device driver bundled with Genode. Instead, our current prototype implements an optimized UDP/IP protocol stack with a native[1] Intel X540 device driver. Both the protocol stack and device driver are integrated into the same process as an early-stage prototype. Typical memcached deployments use UDP/IP for `GET` requests and TCP/IP for `SET`,`ADD`,`DELETE`,`REPLACE` requests.

[1]As opposed to using a wrapped iPXE device driver in the Genode DDE environment.

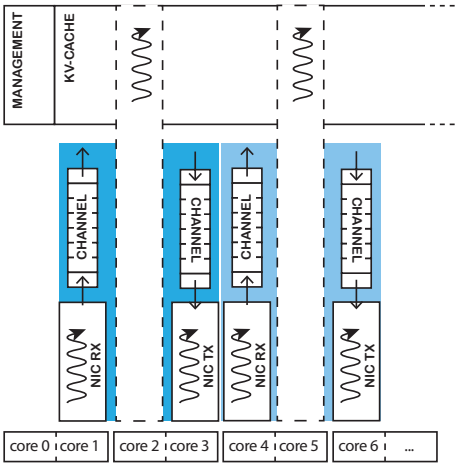

Fig. 2. Thread mapping.

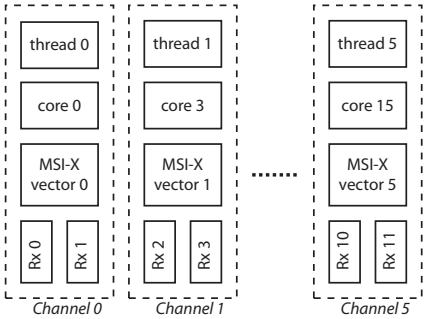

Fig. 3. Network device interrupt handling.

The network device driver is a multi-threaded process that contains receive (Rx) and transmit (Tx) worker threads. Each thread is allocated and pinned to a separate logical core (see Figure 2). Thus, each flow consists of three threads (including an application worker thread) executing on three logical cores; each CPU-socket (16 logical cores) provides five flows on a single NIC device.

Interrupt Handling: The Intel X540 NIC uses MSI-X (Message Signaled Interrupts) to interrupt the device driver. Each MSI-X vector is assigned to two Rx queues (see Figure 3). Interrupts are handled by the Fiasco.OC kernel, which signals the interrupt-handling thread via IPC. There is no cross-core IPC for normal packet delivery. The MSI Address Register's Destination ID [16] is configured so that MSIs are handled by specific cores. There are six channels for each NIC card. Channel 0 is reserved for handling of ARP and other packets. All other channels process KV-Cache packets.

Flow Management: The Intel X540 NIC supports *flow directors* that allow packets to be distributed across multiple NIC queues [17]. In our architecture, each flow maps to a Tx/Rx queue pair. Flows are identified using 2 bytes (of a reserved field) in the application packet frame, which is set by the client. We believe that this small modification on the client side is a reasonable enhancement for a production deployment.

Memory Management: The network subsystem is responsible for all memory allocations and de-allocations in the appliance. Memory is managed using the NUMA-aware allocators

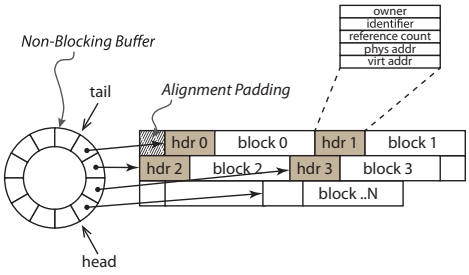

Fig. 4. Slab allocator design.

provided by S-Core. The allocators are instantiated on a per-core basis so that explicit synchronization can be avoided. Each allocator is based on a slab-allocator design that uses a non-blocking buffer scheme [18] to manage pointers to fixed-sized blocks within the slab (see Figure 4). This approach minimizes contention between allocation and free operations. Each memory block is prefixed by a header that includes the physical and virtual addresses for the block (physical address is needed for DMA transmission).

For each request, the network driver constructs a *job descriptor* that represents the application-level (Memcache protocol) command. For multi-packet requests (*e.g.,* for a `SET`, `ADD`, `REPLACE` operations of large objects), the job descriptor maintains a linked-list of all of the packet buffers that make up the command. However, defragmentation is not performed; packet data is left in the original DMA-location within memory – it is truly zero copy.

Memory reference counting is managed as part of the packet list (in the head item). A reference count of zero indicates to the network layer that the cached value has been deleted (or replaced) and thus the memory can be released for each of the associated packets. This chosen scheme of the network layer performing the release of application-used memory avoids the need to synchronize in order to prevent premature freeing of memory while packets are waiting for transmission.

V. CACHING APPLICATION LAYER

The objective of KV-Cache is to cache key-value pairs in RAM according to requests/commands of the Memcache protocol. Keys are strings, typically corresponding to database queries or URLs. They can be up to 250 bytes long. Values are of any form and typically up to 1MB.

The crux of the cache application is a shared-memory hash table for storage of values, together with a data structure for maintaining an LRU (Least Recently Used) eviction policy. Cached values are also evicted according to their expiration time, which is defined when each key-value pair is set.

Besides hit rate, *capacity* is a *de facto* cache performance metric. Capacity is typically defined as the maximum throughput, in requests per second (RPS), for which a service-level of an average round-trip time (RTT) less than 1 ms is met. In most evaluations dropped packets are treated as misses and requests exceeding the target RTT are not counted. Typical evaluations also use a uniform key distribution; *i.e.,* requests are made for keys generated from a random number generator, thus resulting in all keys having equal load.

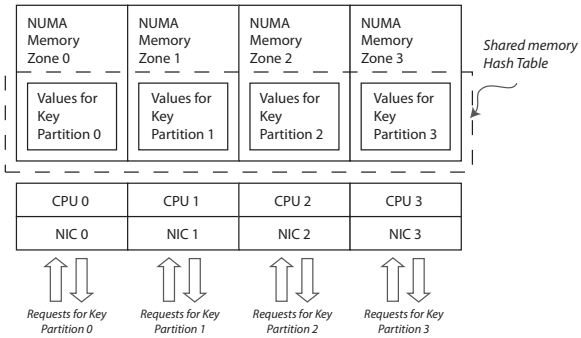

Fig. 5. Key space partitioning.

A. Key Space Partitioning

To maximize performance, KV-Cache partitions the handling of requests according to the key space. The NIC card, memory, CPU, and IO paths are vertically aligned to handle requests for a specific range of keys (see Figure 5). For example, NIC 0 handles the first quarter of the key space, NIC 1 the second, and so forth. To achieve this, the client itself hashes the request key so that it can determine in advance which flow owns the corresponding bucket, and then sets the packets' flow director bytes to the appropriate flow identifier.

As previously discussed in Section IV-A, the requests entering each NIC are further distributed across the five flows using the flow director feature. Thus, the system's *fast path* is where a request enters a given NIC, is processed on the corresponding CPU only accessing local NUMA memory, and the result sent out of the same NIC.

However, for non-uniform distributions the workload becomes unbalanced according to key-space. In this case, KV-Cache uses *work stealing* across application threads to distribute the workload (see Figure 2). When application threads no longer have job descriptors to service from their own (home) channel, they can "steal" descriptors from other (remote) channels and perform the work on their behalf. Stealing is done in a NUMA-aware fashion such that local stealing from the home socket/NIC is performed in preference to remote stealing from another socket/NIC. Transmission of results for stolen work is performed by the Tx thread associated to the stealer application thread. Memory is returned to the appropriate slab-allocator by means of an identifier embedded in the block meta-data.

While work stealing is able to distribute look-up workload and Tx workload, it cannot effectively balance Rx workload (*i.e.,* hot keys hitting the same NIC/socket). The clients can help balance the Rx workload by adjusting the hash function for the flow director (which has to be aligned with the key-value store's hash function). This is a topic of *dynamic key-space partitioning*, which is outside the scope of this paper.

B. Hash Table Design

The hash table consists of an array of 2^n *buckets* (see Figure 6). Each bucket is a pointer to a double-linked list of *hash descriptors*. Because the hash table is shared across multiple threads that could (in the case of load balancing) access the same buckets, locking is applied to serialize their access. To avoid coarse-grained locking, the hash table elements are

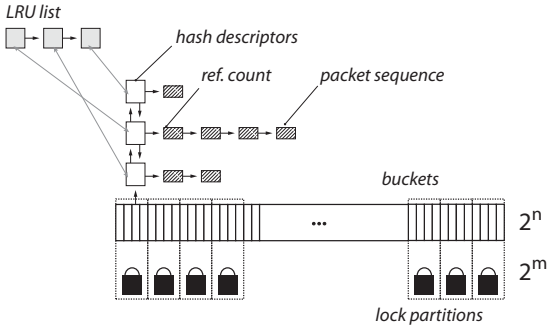

Fig. 6. Hash table design.

protected by 2^m locks where $m < n$. In the current prototype, $n = 20$ and $m = 18$; thus, each lock protects 4 buckets in the hash table.

Each hash descriptor represents a logical key-value pair. Fields in the descriptor include key, key size, value, value size, last time of access, expiration time, originating memory slab identifier, and pointer to LRU node. As part of the zero-copy architecture, the value data is stored as an ordered sequence of network frames. An indirect advantage of this scheme is that a slab allocator (as opposed to a heap allocator) can be used to manage the value memory (refer to Section IV-A). The hash descriptor also includes a back pointer to its location in the LRU data structure. This is to ease the update and removal of key-value pairs from the LRU.

In order to maximize performance, all of the memory needed for the hash-table and LRU data structures is allocated to according to NUMA locality.

C. Replacement Policy

Both Memcached and KV-Cache use an LRU replacement policy. Eviction is necessary when the memory is exhausted or a cached value expires with respect to time.

Memcached's "vanilla" implementation of LRU uses a doubly-linked list synchronized via a global lock. A more effective approach is the Generalized CLOCK (GCLOCK) algorithm [19], which is a variant of the second-chance CLOCK page replacement algorithm [20]. The essence of CLOCK is to maintain a circular buffer of reference bits, one for each page of memory. When a page is accessed, its bit is set. For page eviction, the "clock hand" it sweeps through the buffer until it finds a zero reference bit. Set reference bits are cleared as the hand sweeps past them. The generalized enhancement of CLOCK is the use of a reference counter as opposed to a single bit. The effect of this is that access frequency can be captured and considered in determining which entry to evict.

Our solution uses a novel extension of the GCLOCK design, which we term NbQ-CLOCK (short for *Non-blocking Queue-based CLOCK*). NbQ-CLOCK replaces the fixed clock array with a lock-free queue [21], which provides thread-safe push and pop operations through atomic operations. This variation eliminates the need for an explicit "hand" marker as this is implicitly replaced by push and pop operations on the queue. The use of a dynamic queue also allows the total set of cached items to be dynamically modified (where as the GCLOCK design was previously aimed at a fixed set of memory pages). Eviction is performed by popping items from the queue and checking the reference counters. If the reference counter is zero, the item can be evicted; otherwise, the reference counter is decremented. NbQ-CLOCK also maintains a *delete marker* that is set when values must be deleted.

TABLE II. SPECIFICATION OF THE TEST HARDWARE PLATFORM

Processors	Intel E5-4640 2.4GHz CPU (2.7 GHz Turbo)
	8 cores, 16 hardware threads per socket
	20MB L3 shared cache Per-core 256KB L2 cache and 32KB L1 cache
	QPI interconnect: 8 GT/s
	Max TDP: 95W
	4 DDR3 memory channels with 51.2 GB/s
Motherboard	Super Micro Computer Inc. SuperSaver 8047R-7JRFT
	4 sockets with x16 PCIe to each socket
	Intel C602 chipset
NIC	4 Intel X540 10G PCIe Ethernet adapters
	Direct IO support (DMA to L3 cache)
	Per-CPU Configuration (independent PCI path)

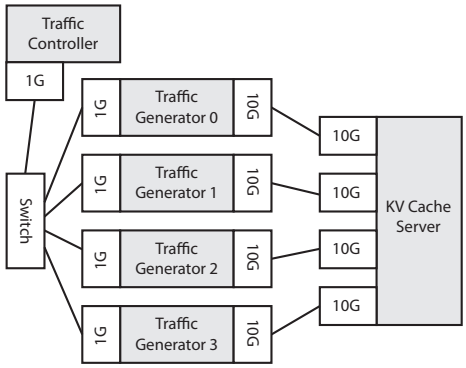

Fig. 7. Traffic generator arrangement.

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate the total system performance of our KV-Cache prototype. We measure *peak throughput* that KV-Cache can achieve without violating the de facto service-level agreement (SLA) [4], [3], requiring an average round-trip time (RTT) under 1 ms (Section VI-A). We argue that this limited SLA is insufficient to ensure effectiveness of web-object caching systems in real-life deployments (Section VI-B). Therefore, we propose a stricter SLA that we believe is more practical. We show that KV-Cache is able to satisfy this new SLA while sustaining near-peak throughput. Finally, we evaluate KV-Cache's scalability and show that it scales linearly with the number of NICs (Section VI-C).

Our system prototype is evaluated on a quad-socket Intel E5-4640 server system. This platform provides 32 cores with fully cache-coherent memory and hyper-threading support, offering a total of 64 hardware threads (logical cores). In addition, it supports advanced features such as Direct Data IO, on-chip PCIe 3.0 controllers, and CPU frequency Turbo Boost. Table II gives a detailed specification of the test hardware platform.

Four separate Traffic-Generator (TG) nodes send requests to the KV-Cache server, and each TG is connected to the server

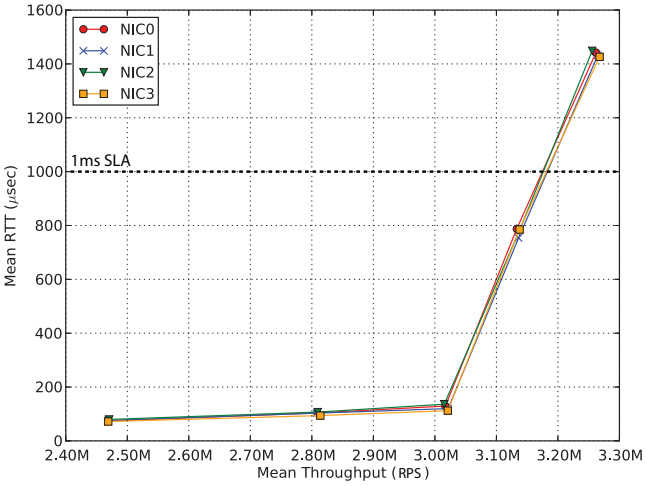

Fig. 8. KV-Cache's throughput and RTT performance for 4 NICs.

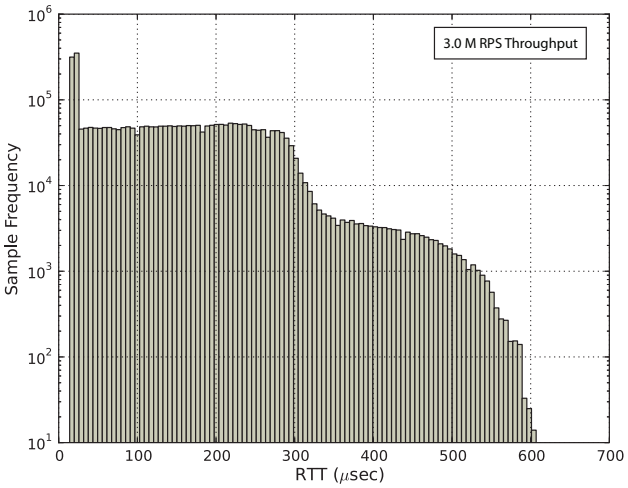

Fig. 9. RTT distribution at 3 million RPS for a single NIC.

via a point-to-point 10Gbps Ethernet link (see Figure 7). The TGs also collect data about throughput, RTT (*i.e.,* latency), late responses, and dropped packets. They are coordinated by a single Traffic-Controller (TC) node, which also collects real-time performance statistics from them.

Like KV-Cache, the TGs are implemented on Fiasco.OC/Genode so that they can generate sufficient traffic to saturate the server. Each TG runs on a Dell Precision T3500 workstation with a 3.6GHz single-socket quad-core Intel Xeon Processor E5-1620, hyper-threading enabled (*i.e.,* 8 hardware threads), 8GB RAM, and one Intel X540 10Gbps NIC. Each TG has four transmit threads and four receive threads, each pinned to a dedicated hardware thread. The transmit threads send requests to the NIC flows in a round-robin fashion (by manipulating the flow director field). We did not use the *mcblaster* [22], a popular Linux traffic-generator program for Memcached, primarily due to its limited generation rate (typically < 1M RPS).[2]

The request workload used in our experiments is similar to that used in the evaluation of Intel's Bag-LRU Memcached [3]. It consists of an initial round of `SET` requests, followed only by `GET` requests. Keys and values are 4 bytes and 64 bytes long, respectively. In our case, each TG uses a unique and relatively small set of keys, and the keys are generated to minimize lock contention in KV-Cache's hash table. This workload corresponds to a very favorable scenario and allows the system to achieve peak performance.

A. Throughput Capacity

The current de facto SLA for Memcached requires request-response pairs to have an *average* round-trip time (RTT) not larger than 1 ms. This SLA has been deemed by other researchers [4], [3] to offer an acceptable quality of service (QoS) for Memcached clients. Thus, the system's *throughput capacity* has been defined as the maximum number of RPS the system can sustain without violating the SLA.

Figure 8 shows the relationship between mean throughput and mean RTT per individual NIC in our system for the considered workload. Our results show that KV-Cache is able to sustain over 3.1M RPS per NIC with an average RTT less than 1 ms (indicated on figure). KV-Cache's total capacity (with all 4 NICS) is approximately 12.7 M RPS. This capacity effectively doubles an *idealistic extrapolation* of the reported capacity of Intel's Bag-LRU Memcached [3] for a 4-socket hardware platform.

[2]M = million, and RPS = requests per second.

B. Round-Trip Time Distribution

As indicated in Figure 8, when capacity is *stressed*, the system performance exhibits an increased RTT. We observe a significant increase in RTT at around 3M RPS; at this point data flow begins to "pile up" in the system. Stressing the system further results in packet dropping, which means that some requests do not get answered.

Figure 8 shows throughput rates with zero packet loss. Figure 9 provides the latency distribution of over 3 million samples (after a given warm-up period) for a throughput of 3.0M RPS. The data indicates a maximum RTT of 600 μseconds, while 95% of the samples achieve an RTT of less than 300 μseconds (note the logarithmic y scale).

We believe that the SLA-violation rate (in our case zero) is an important performance metric that is directly related to the effectiveness of a web-object caching solution, such as KV-Cache and Memcached, in a real-life deployment. Even a few late responses can significantly impact end-user performance [23] – we believe that minimizing the total number of delayed responses is as important as maximizing throughput. Consequently, we propose an extension to the SLA that includes: 1) a target RTT value that request-response pairs must observe on average (*e.g.,* 1 ms), and 2) a percentage of late responses, arriving after their target RTT, that must not be exceeded (*e.g.,* 0.01%).

C. Scalability

In this section, we evaluate KV-Cache's throughput scalability with respect to the number of channels (*i.e.,* flows) in a individual NIC as well as the number of NICs. The throughput

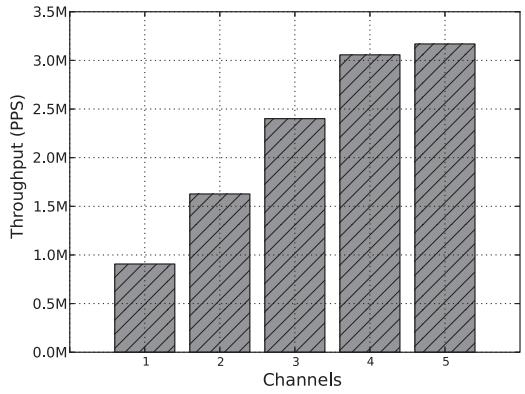

Fig. 10. Throughput scaling for 1 to 5 channels on a NIC.

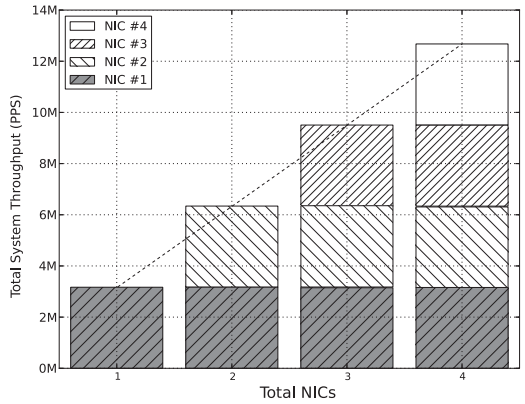

Fig. 11. Throughput scaling for 1 to 4 NICs.

values reported here were obtained while maintaining an average RTT of less than 1 ms (around 3.1M RPS).

As shown in Figure 10, the intra-NIC throughput gain diminishes as the number of channels increases. The units are given in Packets Per Second (PPS). However, in these experiments there is exactly one request per packet; we are not using the multi-get command, which combines multiple requests into a single packet.

We believe that this non-linear scaling behavior is attributed directly to NIC hardware limitations; we do not believe that the software design is the cause of this degradation. This may also be an issue with client-side scaling.

Figure 11, shows that KV-Cache is able to provide linear throughput scaling with an increasing number of NICs. KV-Cache's good scalability across NICs is a direct consequence of the zero-copy approach and its partitioned software architecture (see Section IV).

VII. RELATED WORK

Memcached has been widely studied in both industry and academia due to its simplicity and popularity. In this section, we present recent work in improving Memcached performance. Related work can be divided into three categories: improving software scalability, improving cross-node memory sharing through RDMA-capable network hardware, and improving power/performance through custom/non-commodity hardware.

Improving Software Scalability: Work in this category focuses on reducing the contention in software. Wiggins et al. [3] employ concurrent data structures and a modified LRU cache replacement strategy to overcome Memcached's thread-scaling limitations. The data structures enable concurrent lock-less item retrieval and provide striped lock capability for hash-table updates. The replacement strategy, a non-strict LRU variant, imposes a relaxed ordering of items based on relative timestamps.

In [24], the authors use virtualization to improve scale-up of Memcached. Based on experiments with VMware ESX running on AMD Opteron 6200 processors, they show that virtualization can be used to double system throughput while keeping average response times under 1 ms. Nishtala et al. [25] present important themes that emerge at different scales of Memcached deployment in Facebook, ranging from cluster-level to region-level to global-level. They also optimize single-server performance by using a more scalable hash table, multi-port server thread, adaptive slab allocator, and a transient item cache.

RDMA Integration: The focus of work in this category is to improve network and cache performance for scale-out by using RDMA transport and distributed shared memory semantics. Jose et al. [26] propose the use of non-commodity networking hardware (InfiniBand) with RDMA interfaces to improve cache performance. Their design extends the existing Memcached software with RDMA memory sharing. In a later work [27], they developed a scalable and high-performance Memcached design using hybrid InfiniBand transports that allow connections to transparently switch between Un-reliable Datagram (UD) and Reliable Connection (RC) to offer a hybrid transport to the Memcached stack. Other work proposes to improve performance using a software-based RDMA interface over commodity networking [28]. They studied the feasibility of implementing Memcached on a cluster of low-power CPUs interconnected with 10Gbps Ethernet, and showed that by leveraging one-sided operations in soft-RDMA they could improve CPU efficiency and thus overall performance.

Hardware Optimization: Berezecki et al. [4] have studied Memcached in the context of the TILERA many-core platform. They attribute increased performance to the elimination of serializing bottlenecks using on-chip core connectivity (switch interconnect), as well as resource partitioning (CPU and memory) for different functions, such as the kernel networking stack and application modules. Other work examines the use of GPUs [29] and custom hardware to accelerate Memcached. Chalamalasetti et al. [5] implemented a fully FPGA-based Memcached appliance.

VIII. CONCLUSIONS

Servers with high core counts and large coherent shared memories provide an effective platform for caching by helping to maximize cache hit rates and reduce redundancy (*i.e.,* copies of cached values). However, the challenge of multi-threaded software scaling, in both the operating system and application, can quickly degrade potential performance gains if not addressed.

In this paper, we presented a radically re-designed implementation of *Memcached*, the open source Memcache server

implementation. Our solution, KV-Cache, while compliant with the Memcache protocol, offers significant gains in performance and scalability. The principal philosophies behind the KV-Cache design are: 1) use of a real-time micro-kernel OS as the basis for enabling heavy use of application-specific IO and memory optimizations, 2) absolute zero-copy through a user-level network device driver and lightweight UDP/IP protocol stack, 3) aggressive use of fine-grained locking and lock-free data structures, and 4) fixed size (slab) NUMA-aware partitioned memory management.

Taking these philosophies together, our current prototype based on commodity x86 server hardware, shows a cache performance of over 12M RPS on a four-socket, four 10Gbps-NIC platform. This is a significant improvement, in both scaling and absolute performance, over the world next-best Memcache solution reported by Intel [3], which is 3.1M RPS on a two-socket, one 10G-NIC commodity x86 platform. Furthermore, we have shown that our solution is extremely stable with an observed zero-violation rate at over 12 M RPS.

This work is our first step in applying scalable operating-system and application design principles to an important element in cloud data-center infrastructure. While this initial work has focused primarily on core scaling and performance optimizations, future work will investigate energy optimizations and dynamic workload adaptation, as well as applying similar principles to other cloud-centric workloads.

REFERENCES

[1] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'91)*, 1991, pp. 181–186.

[2] "Memcached," http://www.memcached.org/.

[3] A. Wiggins and J. Langston, "Enhancing the scalability of memcached," Tech. Rep., June 2012. [Online]. Available: http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Proceedings of the 2011 International Green Computing Conference (IGCC'11)*, July 2011, pp. 1–8.

[5] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA memcached appliance," in *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*, Frebuary 2013, pp. 245–254.

[6] E. Stone and E. Norbye, "Memcache Binary Protocol," Internet Draft No. 1654, pp. 1–31, August 2008. [Online]. Available: http://code.google.com/p/memcached/wiki/MemcacheBinaryProtocol

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, 2009, pp. 29–44.

[8] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, October 2009, pp. 221–234.

[9] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, April 2009.

[10] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008, pp. 43–57.

[11] Technische Universitat Dresden, "Fiasco.OC Microkernel," http://os.inf.tu-dresden.de/fiasco/.

[12] D. Wagner, "Object capabilities for security," in *Proceedings of the ACM SIGPLAN 2006 Workshop on Programming Languages and Analysis for Security (PLAS'06)*, 2006, pp. 1–2, Invited Talk.

[13] M. Hohmuth and M. Peter, "Helping in a multiprocessor environment," 2001.

[14] Genode Labs, "Genode operating system framework, general overview," http://genode.org/documentation/general-overview/index.

[15] N. Feske and C. Helmuth, "Design of the Bastei OS Architecture," Technische Universität Dresden, Tech. Rep., December 2006.

[16] R. Budruk, D. Anderson, and E. Solari, *PCI Express System Architecture*. Pearson Education, 2003.

[17] Intel Corporation, "Intel Ethernet controller X540 datasheet," November 2012. [Online]. Available: http://www.intel.com/content/www/us/en/network-adapters/10-gigabit-network-adapters/ethernet-x540-datasheet.html

[18] K. Kim, J. Colmenares, and K.-W. Rim, "Efficient adaptations of the non-blocking buffer for event message communication between real-time threads," in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, May 2007, pp. 29–40.

[19] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Transactions on Database Systems*, vol. 3, no. 3, pp. 223–247, September 1978.

[20] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[21] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, May 1998.

[22] "MCBlaster," https://github.com/livedo/facebook-memcached.

[23] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, February 2013.

[24] R. Hariharan, "Scaling Memcached on AMD processors," AMD WhitePaper PID 52167A. [Online]. Available: http://sites.amd.com/us/Documents/Scaling_Memcached_WhitePaper_PID52167A.pdf

[25] 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 Systems Design and Implementation (NSDI'13)*, April 2013, pp. 385–398.

[26] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached design on high performance RDMA capable interconnects," in *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*, September 2011, pp. 743–752.

[27] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda, "Scalable memcached design for InfiniBand clusters using hybrid transports," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, May 2012, pp. 236–243.

[28] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10GbE: Leveraging one-sided operations in soft-RDMA to boost memcached," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*, June 2012.

[29] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*, April 2012, pp. 88–98.