A Scalable High-Performance In-Memory Key-Value Cache using a Microkernel-Based Design

Daniel G. Waddington, Juan A. Colmenares, Jilong Kuang, Reza Dorrigiv
{d.waddintong,juan.col,jilong.kuang,r.dorrigiv}@samsung.com

Abstract—Latency and costs 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 performance and scalability 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 extensive experimental results for the current prototype running on an Intel E5-based 32-core server platform and servicing request traffic with statistical properties similar to realistic workloads. Our results show that KV-Cache offers significant performance advantages over optimized Memcached on Linux for commodity x86 server hardware.

Index Terms—In-memory key-value cache, network cache, microkernel, scalability, multicore, manycore

1 Introduction

WEB-OBJECT caching temporarily stores recently or frequently accessed remote data (*e.g.,* popular photos, micro web logs, and web pages) in a nearby location to avoid "slow" remote requests when possible. *Memcached* [1] is one of the most popular web-object caching solutions. It is a key-value cache that operates purely *in-memory* and provides access to unstructured data through read and write operations, where each unique key maps to one data object.

Memcached is typically deployed in a *side-cache* configuration. Client devices send requests to the front-end web servers, which then attempt to resolve each request from local Memcached servers by issuing a GET request (using the Memcache protocol). If a cache miss occurs, the front-end server handling the request forwards it to the back-end servers for fulfillment (*e.g.,* performing the database transaction, retrieving data from disk). On receiving the result, the front-end server both sends the response to the client and updates the cache by issuing a SET request to the cache.

In earlier work, we introduced KV-Cache [2], a web-object caching solution conforming to the Memcache protocol. KV-Cache is an in-memory key-value cache that exploits a software *absolute zero-copy* approach and aggressive customization to deliver significant performance improvements over existing Memcached-based solutions. Our previous results show that KV-Cache is able to provide more than $2\times$ the performance over Intel's optimized version of Memcached [3] and more than $24\times$ the performance of off-the-shelf Memcached.

The present paper discusses the design of KV-Cache (Section 4), and reports new experimental results from an extensive comparative evaluation of KV-Cache and Intel's Bag-LRU Memcached (Section 6). In our experiments the cache systems are subject to traffic with statistical properties (*e.g.,* key popularity, key and value sizes, and request inter-arrival time) that resemble a real-life workload at Facebook [4]. We use request mixes with increasingly diverse composition carrying read, update, and delete commands. The experimental setup and our workload generator system are described in Section 5.

Our evaluation starts by examining the peak throughput that KV-Cache and Bag-LRU Memcached can sustain while meeting a target round-trip time (RTT) (Section 6.1). In addition to throughput we examine the response latency (Section 6.2). Next, we assess the scalability of both cache systems when servicing requests through an increasing number of network devices (Section 6.3). Finally, we compare the performance of both systems while using request profiles representative of realistic workloads [4] (Section 6.4).

2 KV-Cache Design Overview

In the development of KV-Cache [2] 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

• *D. G. Waddington, J. A. Colmenares, J. Kuang, and R. Dorrigiv are members of the Computer Science Laboratory (CSL) at Samsung Research America – Silicon Valley (SRA-SV).*

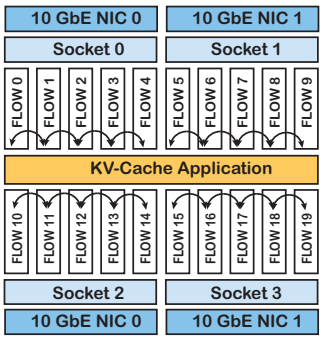


Fig. 1. KV-Cache column architecture.

to monolithic kernels (*e.g.,* Linux) and allows aggressive customization of the memory and network subsystems.

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 spinlocks.
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 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*: The design leverages 10Gbps NICs with support for 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 balance the load among work queues when requests are not evenly distributed across the key space.

KV-Cache's architecture is based on a column-centric design whereby the execution "fast path" flows up and down a single column (see Fig. 1). A "slow path" is executed when work stealing occurs as a result of request patterns with non-uniform access to the key space. This design not only improves performance by minimizing cross-core cache pollution and remote NUMA memory accesses, but also provides an effective framework for performance scaling based on dynamically adjusting the total number of active columns.

Memcached defines both text and binary protocols. To maximize performance, KV-Cache uses the binary protocol [5]. KV-Cache implements five key Memcache commands: GETK, SET, ADD, REPLACE, and DELETE.

3 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.

3.1 L4 Fiasco.OC Microkernel

We chose to implement our solution using a microkernel operating system because of the 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 (*e.g.,* [6], [7], [8], [9]).

The base of our system is the Fiasco.OC L4 microkernel from TU Dresden [10]. Fiasco.OC is a third generation microkernel that provides real-time scheduling and object capabilities [11]. For this work we have kept the kernel largely as-is. The kernel is well-designed for multicore scalability. Per-core data structures, static CPU binding, and local manipulation of remote threads [12] have been integrated into the design.

3.2 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, interprocess 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 [13] as the basis for the personality. The Genode personality is principally aimed at multi-L4 kernel portability. However, what is of more interest in the context of KV-Cache is the explicit partitioning and allocation of resources [14]. 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, as well as partitioning end-system QoS.

In this work, we made some enhancements to the Genode OS Framework to support improved multicore scalability. Two important improvements include the introduction of a NUMA-aware memory management and direct MSI-X interrupt handling to device drivers. Further detail is given in [2].

4 KV-Cache Architecture

KV-Cache comprises two subsystems, the network subsystem and the caching application subsystem. Each subsystem is implemented as a different multi-threaded process. They communicate via shared-memory non-blocking channels (see Fig. 2).

The inter-subsystem channels are lock-free circular buffers. They support efficient communication between a single producer and multiple consumers, enabling work stealing in the application subsystem (refer

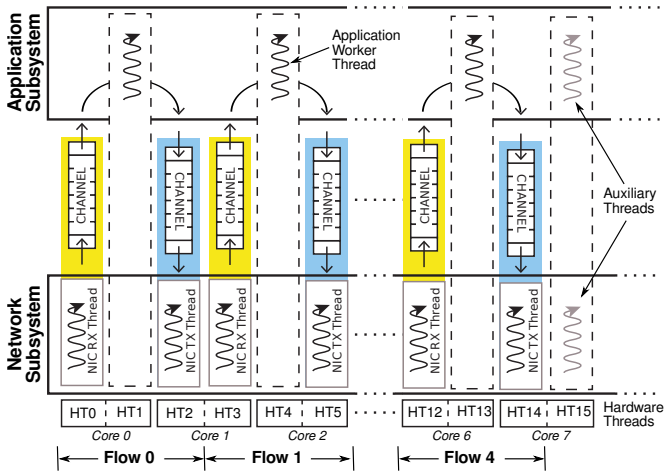


Fig. 2. KV-Cache's architecture and thread mapping.

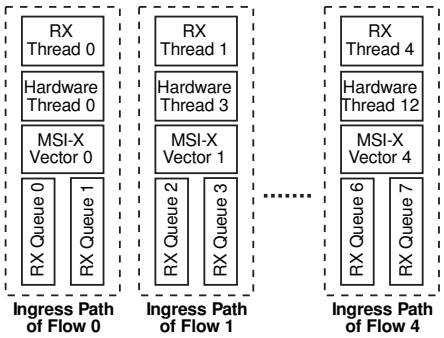


Fig. 3. NIC interrupt handling.

to Section 4.2.1). Through the channels, the subsystems exchange pointers to *job descriptors* that represent application-level (Memcache protocol) commands and results.

Our architecture uses a novel *zero-copy* approach that significantly improves the performance of the KV-Cache 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 UDP/IP protocol stack for the application's needs. This aggressive application-specific tailoring is a fundamental tenet of our approach and 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 to user-space memory, and vice versa.
- Ingress Layer 2 packets (*i.e.,* Ethernet frames) are not assembled (de-fragmented) into the larger IP frames.
- Each cached object (key-value pair) is maintained in memory as a linked list of packet buffers.
- Responses to GET requests 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 is key to KV-Cache's improved system performance.

4.1 Network Subsystem

Web-object caching is an I/O-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 [15] 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.

The network subsystem is a multi-threaded process that contains receive (Rx) and transmit (Tx) software threads. Each thread is allocated and pinned to a separate hardware thread (logical core). As depicted in Fig. 2, each flow consists of three threads (including an application worker thread) executing on three hardware threads. Thus, in a system with 16 hardware threads per CPU,[2] each CPU can accommodate 5 flows. Moreover, excess hardware threads (*e.g.,* HT 15 in Fig. 2) are used to run auxiliary tasks *not* involved in the IO flows, such as periodic removal of deleted and expired key-value pairs (*i.e.,* garbage collection).

4.1.1 NIC Interrupt Handling

The Intel X540 NIC uses *message signaled interrupts* (MSI-X). Each MSI-X vector is assigned to two Rx queues (see Fig. 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 interrupt handling is directed to specific hardware threads.

4.1.2 Flow Management

The Intel X540 NIC supports *flow director filters* [17] that allow ingress packets to be routed to specific NIC queues according to some fields in the packets (*e.g.,* protocol, source IP address, and destination port). In our architecture each flow is assigned at least an Rx queue and a Tx queue. Hence, by setting designated fields in each request packet, a client can control which server's flow will receive the request.

For simplicity, in our current KV-Cache prototype flows are identified using two bytes (of a reserved field) in the application packet frame, which are set by the client. We believe that this small modification on the client side is a reasonable enhancement for a production deployment; however, flow director filters may be configured in a different mode (*e.g.,* perfect match mode [17]) and use standard fields of the protocol headers. Moreover, our prototype uses two Rx queues and two Tx queue per flow as the NIC automatically enables them for each MSI-X vector.

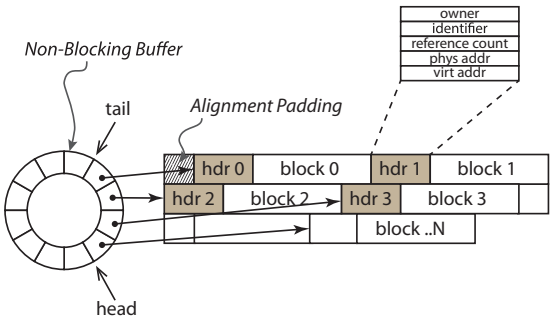


Fig. 4. Slab allocator design.

4.1.3 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 provided by a custom subsystem. 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 Fig. 4). This approach minimizes contention between allocation and free operations.

For each request, the network driver constructs a *job descriptor* that represents the application-level (Memcache protocol) command. For multi-packet requests (*e.g.,* SET, ADD, and 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.

Each memory block is prefixed by a header that includes a reference count as well as physical and virtual addresses for the block. The physical address is needed for DMA transmission of packets. 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. 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.

4.2 Caching Application Subsystem

The objective of KV-Cache is to store 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 up to 1 MB.

The crux of the cache application is a shared-memory hash table for storage of values, together with an eviction policy (*i.e.,* replacement algorithm) and its supporting data structure. Cached values are also evicted according to their expiration time, which is defined when each key-value pair is set.

4.2.1 Key Space Partitioning

To maximize performance, KV-Cache partitions the handling of requests according to the key space. A NIC, NUMA memory zone, CPU, and their associated IO paths are vertically aligned to handle requests for a specific range of keys.

For example, CPU 0 only services requests coming through NIC 0 related to the first quarter of the key space, CPU 1 and NIC 1 handle 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 4.1.2, the requests entering each NIC are further distributed across the flows by the flow director filters. Thus, the system's *fast path* is where a request enters a given NIC, is processed on the corresponding CPU accessing only local NUMA memory, and the result sent out of the same NIC.

However, for requests with keys non-uniformly distributed in the key space the workload may become unbalanced. In this case KV-Cache uses *work stealing* across application threads to try to balance the workload (see Fig. 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 stealing 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). 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.

4.2.2 Hash Table Design

The hash table consists of an array of 2^n *buckets* (see Fig. 5). Each bucket is a pointer to a doubly-linked list of *hash descriptors*. Because the hash table is shared across multiple threads that can (in the case of load balancing) access the same buckets, locking is applied to serialize their access. To avoid coarse-grained locking, the elements in the hash table are 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.

1. Rather than a wrapped iPXE device driver in the Genode DDE environment.

2. like our test server platform described in Section 5.1

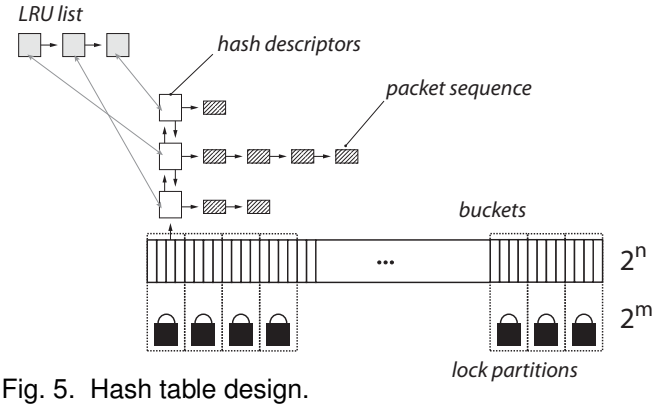


Fig. 5. Hash table design.

Processors	4 Intel E5-4640 2.4GHz CPUs
	8 cores (16 hardware threads) per CPU
	QPI interconnect: 8GT/s
Motherboard	Supermicro® SuperSaver 8047R-7JRFT
	4 sockets with x16 PCIe v3.0 to each socket
NICs	4 Intel X540 10Gbps Ethernet adapters (x8 PCIe v2.1, 5GT/s)
	Direct IO support (DMA to L3 cache)
	Per-CPU configuration (independent PCI path)

TABLE 1
Specification of the test server platform.

Each hash descriptor represents a 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 a node in the eviction-support data structure. 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 4.1).

The hash descriptor includes a back pointer to its location in the eviction-support data structure (described next in Section 4.2.3). This is to ease the update and removal of key-value pairs from the data structure.

In order to maximize performance, all the memory needed for the hash table and the eviction-support data structure is allocated according to NUMA locality.

4.2.3 Replacement Policy

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

Off-the-shelf Memcached implements the *Least Recently Used* (LRU) replacement algorithm with a doubly-linked list synchronized via a global lock. A more effective approach is the Generalized CLOCK algorithm (GCLOCK) [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" sweeps through the buffer until it finds a zero reference bit. Set reference bits are cleared as the hand sweeps past them. GCLOCK's enhancement is the use of a reference counter as opposed to a single bit. Its effect is that not only recency, but access frequency can be captured and considered in determining which entry to evict.

Our solution uses a novel variant of GCLOCK design, called *Non-blocking Queue-based CLOCK* (NbQ-CLOCK) [21]. NbQ-CLOCK replaces the fixed-size array of CLOCK with a lock-free queue [22] that provides thread-safe push and pop operations through atomic instructions. This variant 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 (whereas GCLOCK was previously aimed at a fixed set of memory pages).

Eviction is performed by popping items from the queue and checking their 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.

5 Experimental Setup

This section describes the experimental setup and workload configuration parameters used to evaluate our KV-Cache prototype and Intel's Bag-LRU Memcached [3].

5.1 Test Server Platform

KV-Cache and Bag-LRU Memcached are 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). It supports advanced features such as Direct Data IO and on-chip PCIe 3.0 controllers. The server system is equipped with four Intel X540 10Gbps NICs, each installed in a PCIe slot with a direct path to a different CPU.

Table 1 gives a detailed hardware specification of our test server platform. Turbo Boost and SpeedStep features are disabled for all the experiments.

5.2 Workload Generator System

Simple multi-threaded load-generator programs, such as `mcblaster` [23] and `memslap` [24], are popular and have been used in studies of Memcached (*e.g.*, [3]). Unfortunately, their traffic generation rates are not sufficient to saturate our server. More importantly, they do not attempt to reproduce statistical characteristics of live traffic; they use fixed or uniformly distributed value sizes and keys with equal popularity. Prior work [25] has shown that workloads can drastically alter the behavior of an in-memory key-value cache (*e.g.*, Memcached), and unrealistic workloads can yield misleading conclusions.

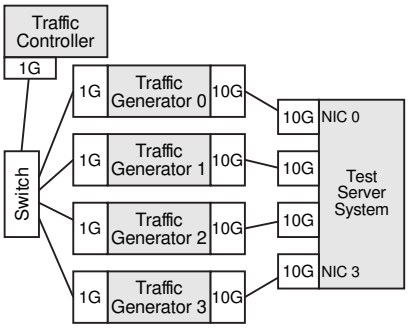


Fig. 6. Traffic generation arrangement.

To avoid these shortcomings, we developed a workload generator system, called KV-Blaster. Based on our optimized Genode/Fiasco.OC system stack (Section 3), KV-Blaster is able to generate over $4{\times}10^6$ RPS on each of our client machines and saturate our server. Furthermore, it can create traffic with similar statistical properties of workloads in production environments, such as those reported in [4].

The KV-Blaster system consists of two types of nodes: 1) one or more *traffic-generator* (TG) nodes, and 2) a *traffic-controller* (TC) node. The TGs produce requests for the in-memory key-value cache running on the server system and measure the cache's performance (*e.g.,* response time and throughput). The TC communicates with the TGs through the network. It controls the way TGs produce the traffic and continuously obtains from them measured data to be displayed on screen.

In this work we use four TG nodes, each connected to a separate NIC of the server via a point-to-point 10Gbps Ethernet link (see Fig. 6). 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 Tx threads and four Rx threads, each pinned to a dedicated hardware thread. Rx threads run on hardware threads $\{0, 2, 4, 6\}$, and Tx threads on $\{1, 3, 5, 7\}$; thus, each Tx/Rx-thread pair is given a CPU core. As for KV-Cache, each Rx thread is vertically aligned with a NIC Rx queue, and so is each Tx thread with a Tx queue. Each Tx thread sends requests to Rx queues (or flows) on the server's NIC in a round-robin fashion by setting fields in the packets, as described in Section 4.1.2. Each Tx thread also sets the UDP source port of the requests, so that the responses are directed to its sibling Rx thread. By exploiting multiple queues on the NIC, we prevent the TGs from becoming a performance bottleneck.

The TGs generate performance reports (when requested by the TC node) containing: 1) the number of sent requests, 2) the number of successful and error responses, 3) the Tx and Rx throughput in RPS, 4) the minimum, maximum, and average values for round-trip time (RTT) in μs, and 5) the number of late responses, which are those arriving after a predetermined target RTT (*e.g.,* 1 ms). Each report contains measured data, separated per request type, obtained from the instant the previous report was generated or the instant the reporting TG started collecting performance data.

The TC program runs on Linux and is implemented in Python for portability. The TC node is connected to the TGs through a 1Gbps switch on a separate LAN.

5.2.1 Statistical Workload Generation

TG nodes can be configured to produce traffic with different statistical properties, including those similar to realistic workloads. Configurable attributes on the TGs are: 1) key-appearance frequency (*i.e.,* popularity), 2) request type, 3) request inter-departure time, and 4) value size. Each attribute is configured by a histogram that defines a desired distribution. Additionally, a TG receives a set of pre-generated keys associated with the key-appearance histogram. *Inverse transform sampling*[3] [26] is used to derive each request's key, command type, time interval (from the previous request), and the value's size (for ADD, SET, or REPLACE).

Inverse transform sampling, together with the use of histograms to define behavior, allow flexibility in traffic generation according to a variety of distributions. Users can create histograms with the tools of their choice (*e.g.,* MATLAB and R), write the histograms to files, and configure the TGs with those files via the TC node. In this work, we use Python and the SciPy library [27] to generate the traffic histograms and a set of unique keys.

Table 2 summarizes the workload configuration parameters used in our experiments of Section 6. We use 10^6 unique keys, distributed among the four TG nodes in equally-sized disjoint subsets. Key sizes are randomly generated from 1 to 250 bytes (as required to conform to the Memcache protocol) using a generalized extreme value distribution from [4].

Atikoglu *et al.* [4] do not define a concrete functional model for the distribution of key repetitions in requests (key appearances) – they only show that the studied workloads exhibit the expected long-tail distributions (*i.e.,* a small percentage of keys appear in most requests, and most keys repeat only a handful of times). Atikoglu *et al.* concur with previous studies of client request traces [28], [29] in that many characteristics, including data-item popularity, can be modeled as power law distributions. For that reason, we adopt a distribution of this type to model the popularity of the keys. The popularity rank of the $250{\times}10^3$ keys of each TG is determined at random by shuffling the elements of the array $[0, 1, \cdots, 249999]$, where each element is an index in the array of keys.

We observe that, for this key count, selecting a key for every request using a histogram directly derived from the chosen distribution is prohibitively expensive. The

3. A method for generating sample numbers at random from any probability distribution, given its cumulative distribution function.

Aspect	Configuration
Key count	10^6 unique keys, split in $250{\times}10^3$ keys per TG node.
Key sizes (in bytes)	Random values generated in the range $[1, 250]$ using a generalized extreme value distribution with parameters $\mu = 30.7984$ (location), $\sigma = 8.20449$ (scale), $\epsilon = 0.078688$ (shape).
Key-appearance frequency	The $250{\times}10^3$ keys per TG are grouped in 10,000 "buckets," each with 25 keys. The histogram of the popularity of the buckets is generated using the power law distribution $PDF(x) = \alpha x^{\alpha-1}$, with $\alpha = 0.01$ and $0.001 \leq x \leq 1$. Keys in the same bucket have the same probability of appearing.
Mixes of request types	*Request mix A*: 100% GET.
	Request mix B: 96.77% GET and 3.23% SET (30:1 ratio of reads and updates).
	Request mix C: 70% GET, 25% SET, and 5% DELETE.
Value sizes (in bytes)	Predetermined fixed values: $\{64, 128, 256, 512, 768, 1024\}$
	Histogram generated for values in the range $[64, 1024]$ using a generalized Pareto distribution with parameters $\theta = 0$ (location), $\sigma = 0.348238$ (scale), $\epsilon = 214.476$ (shape).
Request inter-departure times (in μs)	Manually adjusted to control the transmit rates of the TG nodes.
	Histogram of value in the range $[1, 10^3]$, generated using a generalized Pareto distribution with parameters $\theta = 0$ (location), $\sigma = 0.154971$ (scale), $\epsilon = 16.0292$ (shape).

TABLE 2
Workload configuration parameters. The distributions of key sizes, value sizes, and request inter-departure times are those reported in Section 5 of the paper by Atikoglu *et al.* [4].

overhead prevents the TGs from being able to saturate the server. To reduce the computational cost of inverse transform sampling, and overcome this limitation, we group the keys of each TG into "buckets" of 25 keys (*i.e.,* 10,000 buckets per TG). The power law distribution is used to generate a histogram for the buckets, and a key in a bucket is selected uniformly at random. We believe this approach is adequate because it approximates reasonably well the desired key-appearance frequencies while maintaining high traffic generation rates.

Throughout our evaluation we consider three basic Memcache protocol commands: GET, SET, and DELETE. However, KV-Cache's implementation of GET is effectively GETK. The difference between GET and GETK is that the latter adds the key in the response to a successful request. Therefore, GETK responses carry more data than GET responses, which only favors Bag-LRU Memcached when comparing both cache systems. For brevity, hereafter we do not distinguish between GET and GETK and refer to both request types as GET.

As shown in Table 2, we use three request mixes with increasingly diverse composition. *Request mix A* with 100% GETs allows for read-only performance evaluation. *Request mix B* with 96.77% GETs and 3.23% SETs reproduces the 30:1 ratio of reads and updates observed in the most representative of large-scale, general-purpose workloads (*i.e.,* ETC) reported in [4]. Finally, *request mix C* with 70% GETs, 25% SETs, and 5% DELETEs is a mostly-read mix with a 7:3 ratio between GETs and the other request types. This mix is chosen because its low percentage of DELETEs (compared to SETs) results in a relatively stable population in the cache;[4] thus, most GET and DELETE requests find the key-value pairs in the cache resulting in a successful response.

In each experimental run, the TGs first pre-populate the cache with key-value pairs congruent with the desired traffic profile. After population, the TGs transmit requests according to the defined traffic distributions. Since we only use a few simple request mixes, the request-type histograms used to configure the TGs are created manually.

We start our performance evaluation with a predetermined set of value sizes (from 64 to 1024 bytes), and vary the request inter-departure times to control the TGs' transmit rates. Then, to experiment with a more realistic workload, we use the generalized Pareto distributions reported in [4] to model value sizes and request inter-departure times. With these distributions (shown in Table 2) we generate a histogram of value sizes from 64 to 1024 bytes and a histogram of 1000 values evenly spread in the range from 1 μs to 1 ms. Note that we do not use value sizes beyond 1024 bytes in order to avoid fragmented IP packets – such packets implicitly bypass the flow director filters [17] on the NIC and are routed to Rx queue 0 by the NIC hardware.

5.3 Bag-LRU Memcached Configuration

We use the latest version of Bag-LRU Memcached available at GitHub at the time of writing.[5] It is implemented as an engine (`bag_lru_engine.so`) provided as an alternative to the Memcached's default engine. Bag-LRU Memcached runs on our test server platform with Ubuntu Server 12.04.3 LTS (GNU/Linux 3.2.0-55-generic x86_64) and the Intel's NIC driver ixgbe 3.19.1, the latest at the time of writing. The NICs and Linux network stack are configured to obtain optimal performance for UDP.

4. As opposed to a request mix with 66.0% GETs, 2.2% SETs, and 31.8% DELETEs, similar to that of the ETC workload in [4].

5. https://github.com/rajiv-kapoor/memcached/tree/bagLRU, commit 7f3963d9f9cc82a57fad2a6b9a2cfb6d34a984e5.

Bag-LRU Memcached is run with 64 worker threads. Each worker thread is dedicated (pinned) to a separate hardware thread. Performance isolation on the worker threads is not enforced any further since no other process in the system noticeably influences Bag-LRU Memcached's performance in our experimental setup. Our test server is run in isolation and is not subject to any traffic other than the test traffic.

To optimize IO and try to replicate KV-Cache's column architecture (see Fig. 1), the NICs, the CPUs, and the PCI paths among them are vertically aligned; *i.e.,* CPU i, with $i = 0, 1, 2, 3$, only serves requests received through NIC i, which has a direct PCI channel to the respective CPU.

Further, for each aligned NIC-CPU pair, the NIC Rx and Tx queues, NIC interrupts, and worker threads are also vertically aligned. On our test server, each Intel X540 NIC supports 64 Rx queues and 64 Tx queues (*i.e.,* one Rx queue and one Tx queue per hardware thread in the system). Each NIC is configured so that IRQs from Rx queue j and Tx queue j are handled by hardware thread j, where $j = 0, \cdots, 63$. This is achieved by routing the MSI interrupts to specific cores accordingly (configured through the */proc/irq* file system). Moreover, we make each of the 16 worker threads on each CPU listen to a different UDP destination port (by using Bag-LRU Memcached's command-line option `-l ip:port`). By configuring the flow director filters [17] on each NIC (with the `ethtool` utility) we ensure that single-packet requests destined to a specific UDP port are routed to a predetermined Rx queue and thus served by the *aligned* worker thread. Therefore, a Tx thread on TG i can control which of the 16 worker threads on the server's CPU i will serve a request by setting the UDP destination port.

We use Linux's *transmit packet steering* (XPS) [30] to increase parallelism and improve performance of the egress traffic (*i.e.,* responses). We map hardware thread j onto each NIC's Tx queue j, as suggested in [30].

From our experimentation with different network configuration parameters, we observe that Rx interrupt coalescing has strong influence on Bag-LRU Memcached's latency and throughput. For each NIC, we set `rx-usecs` to 200, which gives the best performance. We also observe that modifying NIC parameters (*e.g.,* sizes of Tx and Rx ring buffers and generic Rx offload) and network-stack parameters (*e.g.,* core and UDP buffer limits in `/etc/sysctl.conf`) either worsens or has no significant influence on Bag-LRU Memcached's performance. For that reason, we use their default values.

6 Performance Evaluation

The current *de facto* service-level objective (SLO) for Memcached requires request-response pairs to have an *average* RTT below 1 ms. This SLO has been shown to offer an acceptable quality of service (QoS) for Memcached clients [31], [3], and the system's *throughput capacity* has been defined as the maximum number of RPS the system can sustain without violating this de facto SLO.

We believe, however, that the percentage of late responses (*i.e.,* those arriving after a target RTT) is an important performance attribute for web-object caching solutions, since even a few late responses can significantly impact the end user experience [32]. Consequently, we are also interested in the peak throughput that KV-Cache and Bag-LRU Memcache can attain without exceeding a *maximum* RTT of 1 ms (*i.e.,* no late responses).

We introduce the following nomenclature: SLO_{avg} is the de facto SLO with a target *average* RTT of 1 ms, Cap_{avg} is the system's throughput capacity while conforming to SLO_{avg}, SLO_{max} is a stricter SLO with a target *maximum* RTT of 1 ms, and Cap_{max} is the system's throughput capacity while conforming to SLO_{max}.

In the rest of this section, we compare the performance of KV-Cache and Intel's Bag-LRU Memcached [3]. We first evaluate their capacity while serving traffic through a single NIC. Next, the response latency of the systems is examined under the same single-NIC scenario. We then assess scalability across multiple NICs, and finally report performance under a realistic workload.

6.1 Single-NIC Throughput Capacity

In this part of the evaluation traffic is generated solely from TG 1, and hence the server only receives requests on NIC 1 (see Fig. 6) and processes them on socket/CPU 1. CPU 1 is chosen (instead of CPU 0) for the single-NIC experiments in order to avoid possible implications of using hardware thread 0 (on CPU 0) that is where Genode system threads reside by default.

The cache system is subject to traffic load carrying different request mixes (see Table 2). Using a predefined set of value sizes (64 to 1024 bytes) we "search" for the capacity of the system under test by varying the request inter-departure time that controls the TG node's Tx throughput. Searching for a capacity value involves conducting a number of runs (5 and often more per value size and system). In every run, TG 1 first pre-populates the cache using the assigned keys. Once pre-population is complete, the configured workload is generated.

We attempt to report Cap_{avg} and Cap_{max} for both cache systems from the collected data. But, in the case of KV-Cache we focus on Cap_{max} (*i.e.,* peak throughput with no late responses). Capacity values are derived from six consecutive performance reports with a period of 10 seconds between them, for a total of a 1 minute observation. The six consecutive reports are selected after the system's behavior stabilizes (typically 10 reports after the population phase ends). The throughput values are given as the average across all reports, while the maximum (average) RTT is the maximum (average) across reports.

6.1.1 Capacity for Request Mix A

In this experiment the cache systems are subject to a *read-only* workload (*i.e.,* 100% GETs). Fig. 7 summarizes our

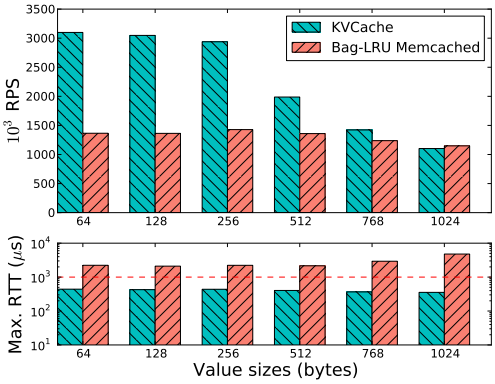


Fig. 7. Capacities and maximum observed RTTs for 100% GET requests. KV-Cache's capacities are Cap_{max}, whereas Bag-LRU Memcached's are Cap_{avg}.

comparative results. The upper plot shows Cap_{max} values for KV-Cache and Cap_{avg} values for Bag-LRU Memcached, for different value sizes. The lower plot of Fig. 7 shows the maximum observed RTTs. Note that while the capacities for 1024-byte values seem comparable (and in fact that Bag-LRU Memcached might be better), the shown value for Bag-LRU Memcached is Cap_{avg} due to a significant number of responses exceeding the target RTT of 1 ms. In contrast, KV-Cache consistently maintained a maximum RTT of less than 1 ms for all value sizes.

The data in Fig. 7 also shows that KV-Cache's capacity decreases with the value sizes (this is expected due to increasing packet size and thus transmission time). Nevertheless, although we observe some reduction in Bag-LRU Memcached's capacity for larger values, it is not as significant as we expected. Since the test workload includes solely GET requests, the invariability of Bag-LRU Memcached's capacity is attributed to Linux's UDP/IP network stack and NIC driver. This conclusion is supported by kernel threads reaching a CPU utilization close to 100% on all the hardware threads of CPU 1.

6.1.2 Capacity for Request Mix B

The traffic generated for this experiment comprises 96.77% GETs and 3.23% SETs; *i.e.,* a request mix with a 30:1 ratio of reads and updates similar to that of a realistic workload reported in [4]. The results are summarized in Fig. 8 and separated by request type.

As in Section 6.1.1, we report Cap_{max} values of KV-Cache and Cap_{avg} values of Bag-LRU Memcached. We report Cap_{avg} values for Bag-LRU Memcached because the respective maximum RTT data is above 1 ms.

Again, KV-Cache's capacity decreases with the value sizes, whereas Bag-LRU Memcached's capacity remains relatively constant. For 1024-byte values, both cache systems offer comparable capacities, with KV-Cache's slightly higher.

6.1.3 Capacity for Request Mix C

In this experiment the request mix is 70% GET, 25% SET, and 5% DELETE. With this profile there is a possibility that GET and DELETE requests do not have an entry in the cache and therefore an error response may be given. To ensure an unbiased comparison, we pay careful attention to the percentage of error responses produced by each solution during the experimental runs. We observe stable and similar percentages (between 16% and 17%) of error responses to GET and DELETE requests across all runs with both cache systems.

Fig. 9 summarizes our results for each request type. As in the previous experiments, we present Cap_{max} values of KV-Cache and Cap_{avg} values of Bag-LRU Memcached. We can only report Bag-LRU Memcached's Cap_{avg} values because once again Bag-LRU Memcached was not able to deliver maximum RTTs below 1 ms in any experimental run.

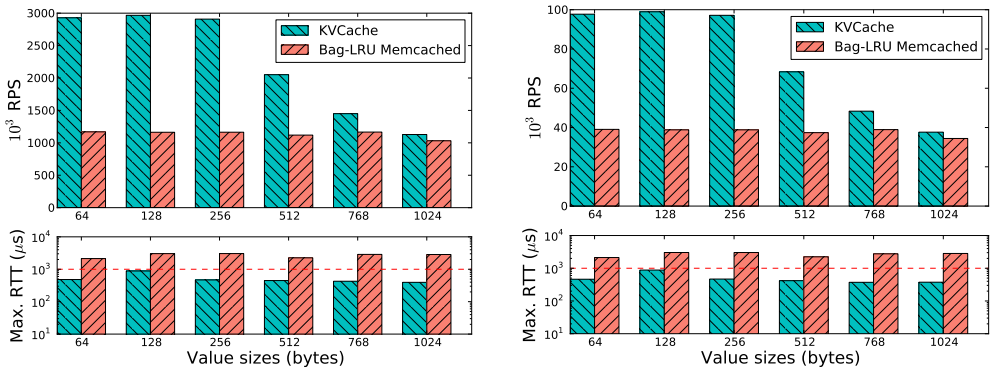
Our results show that while Bag-LRU Memcached experiences severe performance degradation with this request mix, KV-Cache is able to maintain high capacity levels. As before, KV-Cache's capacity decreases with the value sizes.

To investigate the reason for Bag-LRU Memcached's performance degradation, we experimented with other request compositions. We observed that its performance significantly degraded as we increased the percentage of SET requests. For example, for traffic with 70% GETs and 30% SETs and 64-byte values, Bag-LRU Memcached was only able to sustain about 300×10^3 RPS for GETs and 125×10^3 RPS for SETs with an average RTT below 1 ms. The same observation can be made for a request mix of 70% GETs, 25% SETs, and 5% DELETEs, for which we present comprehensive experimental results in this section. We also observed similar performance degradation with the Memcached's default engine (as opposed to Bags-LRU).

6.1.4 Discussion

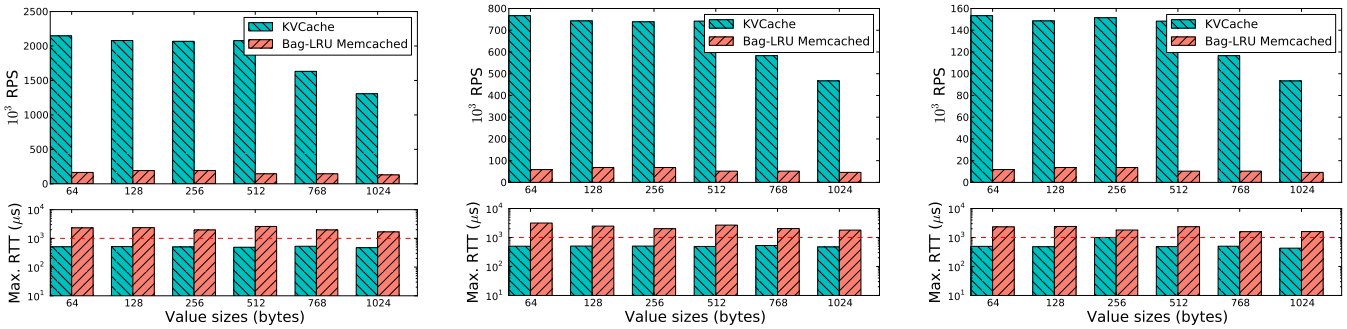
Our experiments show that, in terms of throughput capacity, KV-Cache clearly outperforms Bag-LRU Memcached when servicing traffic with the considered request mixes and value sizes less than 512 bytes. With larger value sizes (*i.e.,* 768 and 1024 bytes), both in-memory cache systems offer similar capacities when the traffic contains zero or a very small percentage (*e.g.,* 3%) of SET requests. Note that these results are obtained even when KV-Cache is forced to the meet the strict requirement of no RTTs above 1 ms (*i.e.,* zero late responses) and Bag-LRU Memcached is not. Moreover, KV-Cache is able to deliver maximum RTTs substantially smaller than Bag-LRU Memcached's, even when capacities are similar (note the log scale on the y axis of the RTT plots).

During our comparative evaluation, two points stand out. First, Linux's UDP/IP network stack, which is general-purpose, seems to be a major limiting factor to Bag-LRU Memcached's performance. KV-Cache overcomes this issue by exploiting aggressive customization of the UDP/IP stack and taking advantage of absolute software zero copy and other techniques described in Section 4.1. Thus, KV-Cache helps make the case for



(a) GET requests (96.77%) (b) SET requests (3.23%)

Fig. 8. Capacities and maximum observed RTTs for 96.77% GETs and 3.23% SETs. KV-Cache's capacities are Cap_{max} and Bag-LRU Memcached's are Cap_{avg}.



(a) GET requests (70%) (b) SET requests (25%) (c) DELETE requests (5%)

Fig. 9. Capacities and maximum observed RTTs for 70% GETs, 25% SETs, and 5% DELETEs. KV-Cache's capacities are defined as Cap_{max}, whereas Bag-LRU Memcached's are defined as Cap_{avg}.

customizable protocol stacks that can support specific, yet important systems and applications.

Second, the presence of SET (*i.e.*, update) requests severely affects the performance of Bag-LRU Memcached. KV-Cache, on the other hand, experiences negligible degradation in total system capacity due to SET requests. Besides storing and replacing data items, SET requests trigger KV-Cache's eviction logic once the amount of memory used to store the key-value pairs exceeds a predetermined threshold. This result is a strong indication of the effectiveness of NbQ-CLOCK [21] (see Section 4.2.3) and its integration into KV-Cache.

6.2 Single-NIC Response Latency

In this section we compare KV-Cache and Bag-LRU Memcached in terms of response latency while serving traffic through a single NIC. We report first-order RTT statistics for both systems while servicing the request mixes of Table 2. For brevity, we present latency data for only 64- and 1024-byte value lengths (*i.e.*, the smallest and largest experimental value lengths).

As in Section 6.1, the traffic is generated solely from TG 1, and hence the server only receives and processes requests on NIC 1 and CPU 1, respectively. For RTT data collection, TG 1 is configured to produce traffic at a rate that is close to, but lower than, the Cap_{avg} of the system under test. The rate is selected so that there are no packet drops, and it is also reported in our results.

Latency is measured by time stamping the requests using the processor's time stamp counter (via `rdtsc`) to first record the transmit time (on the Tx thread) and then the return receive time (on the Rx thread). The originating time stamps are carried in the `opaque` field of the Memcache binary protocol's header.

The `opaque` field in Memcache binary protocol's header is 32 bits. For KV-Cache, we modified the header and made the field 64 bits, sufficient to accommodate the time stamp. In the case of Bag-LRU Memcached, we did not change the protocol header; instead, we store bytes $\{b_5, b_4, b_3, b_2\}$ of the 64-bit time stamp in the 32-bit `opaque` field. Excluding bytes $\{b_1, b_0\}$ from the time stamp adds *at most* 18.2 μs to the RTT measurements, which is small ($\leq 2\%$) compare to the 1-ms target RTT. Excluding time stamp's bytes $\{b_7, b_6\}$ is of no concern for this time scale.

When a TG 1's Rx thread processes a response, it determines the RTT of the request-response pair as the difference of the current read of the processor's timestamp counter and the request's origination timestamp.

System	Throughput	Min.	Max.	Avg.	Std.Dev.
KV-Cache	3.18×10^6RPS	14μs	395μs	86μs	65.49μs
Bag-LRU Memcached	1.36×10^6RPS	46μs	1063μs	236μs	165.25μs

TABLE 3
RTT statistics for 100% GETs with 64-byte values.

System	Throughput	Min.	Max.	Avg.	Std.Dev.
KV-Cache	1.11×10^6RPS	15μs	357μs	46μs	23.12μs
Bag-LRU Memcached	1.10×10^6RPS	48μs	3802μs	219μs	56.72μs

TABLE 4
RTT statistics for 100% GETs with 1024-byte values.

System	Request	Throughput	Min.	Max.	Avg.	Std.Dev.
KV-Cache	GET	2.9×10^6RPS	15μs	446μs	107μs	83.23μs
	SET	96.8×10^3RPS	16μs	458μs	107μs	83.08μs
Bag-LRU Memcached	GET	1.1×10^6RPS	41μs	2340μs	214μs	82.09μs
	SET	3.8×10^3RPS	69μs	2335μs	222μs	88.09μs

TABLE 5
RTT statistics for 96.77% GETs and 3.23% SETs with 64-byte values.

RTT samples are stored in memory until the run has completed; on completion, the timestamp array is offloaded from TG 1.

TG 1 runs with all its Tx and Rx threads, but the Rx thread on hardware thread 0 is excluded from RTT data gathering (*i.e.,* only 3 Rx threads do it). The reason is that hardware thread 0 is time-sliced with other essential Genode/Fiasco.OC processes, which may skew results.

6.2.1 RTT Statistics for Request Mix A

In this experiment both cache systems receive traffic with 100% GET requests. Tables 3 and 4 contain our results, which were derived from 3×10^6 RTT samples.

The data shows that at rates close to capacity, KV-Cache's average and maximum RTTs are much lower than Bag-LRU Memcached's. In particular, KV-Cache offers a maximum RTT significantly ($10\times$) smaller than Bag-LRU Memcached's at a similar throughput of $\sim1.1\times10^6$ RPS for 1024-byte values.

6.2.2 RTT Statistics for Request Mix B

The traffic generated for this experiment comprises 96.77% GETs and 3.23% SETs. Tables 5 and 6 summarize our results, which were obtained from 3×10^6 RTT samples of GET requests and between 250×10^3 and 290×10^3 samples of SET requests.

As before, we observe that KV-Cache offers average and maximum RTTs considerably lower than those provided by Bag-LRU Memcached for both GET and SET requests. Note that for 1024-byte values, Bag-LRU Memcached's maximum RTTs are at least $5.5\times$ larger than KV-Cache's at comparable throughputs of $\sim1\times10^6$ RPS for GETs and $\sim35\times10^3$ RPS for SETs.

System	Request	Throughput	Min.	Max.	Avg.	Std.Dev.
KV-Cache	GET	1.1×10^6RPS	15μs	328μs	60μs	34.06μs
	SET	37.4×10^3RPS	17μs	338μs	62μs	33.52μs
Bag-LRU Memcached	GET	988.6×10^3RPS	50μs	2137μs	207μs	45.07μs
	SET	33.0×10^3RPS	84μs	1866μs	212μs	44.53μs

TABLE 6
RTT statistics for 96.77% GETs and 3.23% SETs with 1024-byte values.

System	Request	Throughput	Min.	Max.	Avg.	Std.Dev.
KV-Cache	GET	2.13×10^6RPS	15μs	482μs	129μs	93.10μs
	SET	763.1×10^3RPS	15μs	470μs	129μs	92.94μs
	DELETE	152.2×10^3RPS	14μs	445μs	129μs	93.07μs
Bag-LRU Memcached	GET	191.4×10^3RPS	53μs	1852μs	168μs	59.15μs
	SET	68.5×10^3RPS	65μs	1838μs	178μs	64.42μs
	DELETE	13.6×10^3RPS	63μs	1400μs	170μs	57.96μs

TABLE 7
RTT statistics for 70% GETs, 25% SETs, and 5% DELETEs with 64-byte values.

System	Request	Throughput	Min.	Max.	Avg.	Std.Dev.
KV-Cache	GET	1.1×10^6RPS	16μs	410μs	72μs	51.66μs
	SET	395.9×10^3RPS	17μs	418μs	72μs	51.05μs
	DELETE	79.2×10^3RPS	15μs	394μs	71μs	51.45μs
Bag-LRU Memcached	GET	116.1×10^3RPS	43μs	433μs	159μs	53.28μs
	SET	41.2×10^3RPS	49μs	418μs	169μs	54.46μs
	DELETE	8.2×10^3RPS	45μs	372μs	159μs	52.96μs

TABLE 8
RTT statistics for 70% GETs, 25% SETs, and 5% DELETEs with 1024-byte values.

6.2.3 RTT Statistics for Request Mix C

In this experiment both cache systems are subject to traffic carrying 70% GET, 25% SET, and 5% DELETE requests. Due to the presence of DELETE requests, it is possible that GETs and also DELETEs do not find the requested key-value pairs in the cache, thus resulting in error responses. As in Section 6.1.3, we confirm that the percentages of error responses are comparable (about 16% in average) in the runs with both cache systems.

Our results are presented in Tables 7 and 8. The results for KV-Cache are obtained from 3×10^6 RTT samples of GET requests, a similar sample count for SETs, and over 550×10^3 samples for DELETEs. Due to Bag-LRU Memcached's low throughput capacity with this request mix, we derive its RTT statistics from fewer RTT samples: 1.4×10^6 for GETs with 64-byte values, 870×10^3 for GETs with 1024-byte values, $\sim400\times10^3$ for SETs, and up to 100×10^3 for DELETEs.

System	Request	Avg. throughput	Min.	Max.	Avg.
KV-Cache	GET	15916 RPS	14μs	239μs	16.0μs
	SET	5689 RPS	15μs	237μs	17.0μs
	DELETE	1143 RPS	14μs	240μs	16.0μs
Bag-LRU Memcached	GET	15922 RPS	35μs	2630μs	72.0μs
	SET	5693 RPS	41μs	1340μs	79.0μs
	DELETE	1140 RPS	41μs	1096μs	72.5μs

TABLE 9
RTT statistics for workload with realistic statistical properties and a request composition of 70% GETs, 25% SETs, and 5% DELETEs.

From Table 7, we can see that for 64-byte values, KV-Cache's maximum RTTs are much lower than Bag-LRU Memcached's, while the average RTTs of both systems are comparable. Table 8 shows comparable results for 1024-byte values. However, one should note that the capacities are about $10\times$ different for this request mix (see Section 6.1.3).

6.3 Inter-NIC Scalability

In this section we assess the scalability of KV-Cache and Bag-LRU Memcached when servicing requests through an increasing number of NICs, from 1 to 4 (see Fig. 6).

The traffic generated in this experiment carries 96.77% GETs and 3.23% SETs (*i.e.,* request mix B). We focus on this request composition because the percentage of SET requests is sufficiently small to cause any severe degradation in Bag-LRU Memcached's performance (see Section 6.1.3). For brevity, we only report results for 1024-byte values; the other value sizes follow the same trend.

For each value size, we consider 4 different *settings*: S_0=$\{0\}$, S_1=$\{0,1\}$, S_2=$\{0,1,2\}$, and S_3=$\{0,1,2,3\}$, each indicating the indices of the TG nodes, server's NICs and CPUs in use. For example, in setting S_1=$\{0,1\}$ the cache systems handle requests from TG 0 through NIC 0 on CPU 0, and from TG 1 through NIC 1 on CPU 1. For each setting, we report KV-Cache's Cap_{max} values and Bag-LRU Memcached's Cap_{avg} values, observed by all the TGs involved in the run.

Fig. 10 shows the results. The labels on the x-axis correspond to each of the settings. The segments of each stacked bar indicate the capacities reported by the TGs, and they are sorted bottom-up according to the indices in the settings.

Our results show that KV-Cache offers linear inter-NIC capacity scaling up to 4 NICs, whereas Bag-LRU Memcached scales poorly. KV-Cache's good scalability across NICs is attributed to the zero-copy approach and its partitioned software architecture (see Section 4). The results indicating poor scaling for Bag-LRU Memcached are congruent with previous work [31], [33].

6.4 Round-Trip Times under Realistic Workloads

Finally, we evaluate KV-Cache and Bag-LRU Memcache with traffic profiles comparable to those of Facebook's real-life ETC workload characterized in [4]. The generated traffic has the same properties related to keys (*i.e.,* the number of keys, key-size distribution, and key-appearance frequency) as in previous experiments. But, rather than using fixed value sizes and controlling the request inter-departure time, the values of these parameters are generated at runtime according to generalized Pareto distributions presented in Table 2.

We load the cache systems with traffic from the four TG nodes. Due to space limitations, we only present results from TG 1 for the request mix C in Table 9. Similar results were obtained from the other TGs for this and the other request mixes. As in Section 6.1, the results are taken from six consecutive performance reports.

Our data shows that the workload is relatively light in terms of throughput ($\sim 22\times 10^3$ RPS) – less than 1% of KV-Cache's maximum capacity and approximately 2% of Bag-LRU Memcached's. In this experiment both cache systems deliver comparable throughputs. However, even at such low rate KV-Cache offers maximum RTTs at least $4\times$ lower than Bag-LRU Memcached's across all request mixes. The RTT for KV-Cache does not exceed 1 ms, whereas Bag-LRU Memcached's RTT does. Finally, both systems deliver similar average and minimum RTTs considering the margin of error for our measurements.

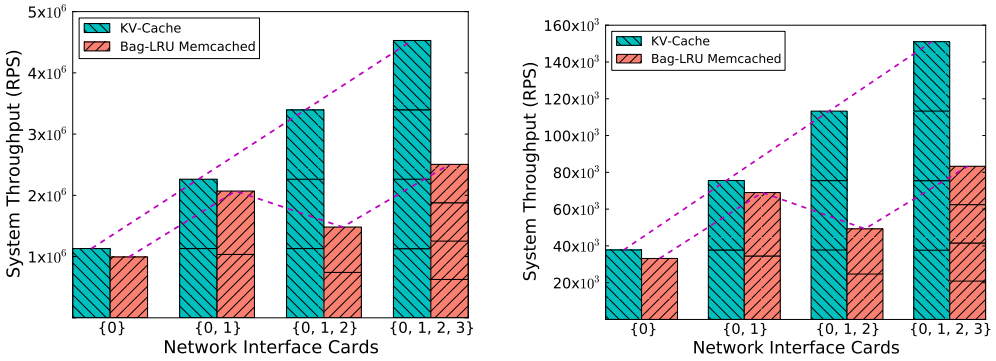
7 Related Work

Memcached has been widely studied in both industry and academia. This section presents recent work in improving Memcached performance.

Improving Software Scalability: Work in this category focuses on reducing contention in software. Wiggins *et al.* [3] employ concurrent data structures and a modified LRU replacement strategy to overcome Memcached's thread-scaling limitations. Hariharan [34] shows that virtualization can be used to double system throughput while keeping average response times under 1 ms. Nishtala *et al.* [35] 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: This body of work focuses on improving network and cache performance for scale-out by using RDMA transport and distributed shared memory semantics. Jose *et al.* [36] propose the use of non-commodity networking hardware (InfiniBand) with RDMA interfaces to improve cache performance. In [37] they developed a scalable and high-performance Memcached design that uses InfiniBand and combines reliable and unreliable transports. Other work proposes to improve performance by using a software-based RDMA interface over commodity networking [38].

Hardware Optimization: Berezecki *et al.* [31] have studied Memcached in the context of the TILERA many-core platform. They attribute increased performance to the



(a) GET requests (96.77%) (b) SET requests (3.23%)

Fig. 10. Inter-NIC scalability for 96.77% GETs and 3.23% SETs with 1024-byte values. Labels in the X axis indicate the NICs being used.

elimination of serializing bottlenecks using on-chip core connectivity (switch interconnect), as well as resource partitioning (CPU and memory). Other work examines the use of GPUs [39] and custom hardware to accelerate Memcached. Chalamalasetti *et al.* [40] implemented a fully FPGA-based Memcached appliance.

8 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 limit potential performance gains if not addressed.

In this paper we presented a radically re-designed implementation of Memcached, a popular open-source in-memory key-value cache 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 micro-kernel operating system as the basis for enabling heavy use of application-specific I/O and memory optimizations, 2) use of an absolute zero-copy architecture by combining a user-level network device driver with a custom lightweight UDP/IP protocol stack, and 3) aggressive use of fine-grained locking and lock-free data structures.

Designed around these philosophies, our current prototype (based on commodity x86 server hardware) shows a cache performance of $\sim 12 \times 10^6$ 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 (Bag-LRU Memcached) reported by Intel [3], which is 3.1×10^6 on a two-socket, one 10G-NIC commodity x86 platform. Furthermore, we have shown that our solution is extremely stable with zero late responses ($RTT \leq 1\ ms$) at $\sim 12 \times 10^6$ RPS of total system throughput.

We ran extensive experiments comparing KV-Cache and Bag-LRU Memcached under various settings. We concentrated on workloads with statistical properties conforming to real-life deployments of Memcached. Our experimental results show that KV-Cache outperforms Bag-LRU Memcached in most cases. An well-known shortcoming of Memcached is its poor scalability [33], [31], [35], [3]. Bag-LRU Memcached aimed at addressing this issue by using concurrent data structures and avoiding contention for the shared global cache lock. Although their solution scales better than "vanilla" Memcached, our results show that KV-Cache scales even better.

This work is a first step in applying scalable operating system and application design principles to an important element in cloud data center infrastructure.

9 Acknowledgment

We would like to thank Prof. Fengguang Song (IUPUI) and Gage Eads (UC Berkeley) for their contributions to the project during their stay as members of the Computer Science Lab. We also acknowledge the support from VP Moon Chan (Samsung Electronics) and Dr. Sung-Ming Lee (Samsung Electronics).

References

[1] "Memcached," http://www.memcached.org/.
[2] D. G. Waddington, J. A. Colmenares, J. Kuang, and F. Song, "KV-Cache: A scalable high-performance web-object caching for many-core," in *Proceedings of the 6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2013)*, December 2013.
[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] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 53–64, June 2012.
[5] 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
[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] "lwIP," http://savannah.nongnu.org/projects/lwip/.

[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] G. Eads, "NbQ-CLOCK: A non-blocking queue-based CLOCK algorithm for web-object caching," Master's thesis, EECS Department, University of California, Berkeley, October 2013.

[22] 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.

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

[24] "Memslap," http://docs.libmemcached.org/bin/memslap.html.

[25] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing SoC accelerators for Memcached," in *Proceedings of the 40th International Symposium on Computer Architecture(ISCA'13)*, June 2013, pp. 36–47.

[26] L. Devroye, *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[27] "SciPy Library," http://scipy.org/scipylib/.

[28] C. R. Cunha, A. Bestavros, and M. E. Crovella, "Characteristics of WWW client-based traces," Department of Computer Science, Boston University, Tech. Rep., July 1995.

[29] M. Cha, A. Mislove, and K. P. Gummadi, "A measurement-driven analysis of information propagation in the Flickr social network," in *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*, April 2009, pp. 721–730.

[30] "Scaling in the Linux networking stack," https://www.kernel.org/doc/Documentation/networking/scaling.txt, accessed: December 2013.

[31] 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.

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

[33] N. J. Gunther, S. Subramanyam, and S. Parvu., "Hidden scalability gotchas in memcached and friends," in *VELOCITY Web Performance and Operations Conference*, 2010.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.