HAL
open science

BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing

Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, Gilles Muller

To cite this version:

Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. NSDI'21 - 18th USENIX Symposium on Networked Systems Design and Implementation, Apr 2021, Virtual event, United States. pp.487-501. hal-03361644

HAL Id: hal-03361644

https://inria.hal.science/hal-03361644v1

Submitted on 1 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing

Yoann Ghigoff
Orange Labs
Sorbonne Université, LIP6, Inria

Julien Sopena
Sorbonne Université, LIP6

Kahina Lazri
Orange Labs

Antoine Blin
Gandi

Gilles Muller
Inria

Abstract

In-memory key-value stores are critical components that help scale large internet services by providing low-latency access to popular data. Memcached, one of the most popular key-value stores, suffers from performance limitations inherent to the Linux networking stack and fails to achieve high performance when using high-speed network interfaces. While the Linux network stack can be bypassed using DPDK based solutions, such approaches require a complete redesign of the software stack and induce high CPU utilization even when client load is low.

To overcome these limitations, we present BMC, an in-kernel cache for Memcached that serves requests before the execution of the standard network stack. Requests to the BMC cache are treated as part of the NIC interrupts, which allows performance to scale with the number of cores serving the NIC queues. To ensure safety, BMC is implemented using eBPF. Despite the safety constraints of eBPF, we show that it is possible to implement a complex cache service. Because BMC runs on commodity hardware and requires modification of neither the Linux kernel nor the Memcached application, it can be widely deployed on existing systems. BMC optimizes the processing time of Facebook-like small-size requests. On this target workload, our evaluations show that BMC improves throughput by up to 18x compared to the vanilla Memcached application and up to 6x compared to an optimized version of Memcached that uses the SO_REUSEPORT socket flag. In addition, our results also show that BMC has negligible overhead and does not deteriorate throughput when treating non-target workloads.

1 Introduction

Memcached [22] is a high-performance in-memory key-value store used as a caching-service solution by cloud providers [1] and large-scale web services [3, 38]. Memcached allows such services to reduce web request latency and alleviate the load on backend databases by using main memory to store and serve popular data over the network.

Memcached, however, is prone to bottlenecks introduced by the underlying operating system's network stack, including Linux's [14, 34], since the main goal of general purpose operating systems is to provide applications with flexible abstractions and interfaces. To achieve high throughput and low latency, user applications can give up using the standard kernel interfaces by using kernel-bypass technologies such as DPDK [4] which allow an application to program network hardware and perform packet I/O from userspace. The application that has control of the network hardware is then responsible for implementing a network stack that fits its specific needs [12, 26]. However, kernel-bypass comes with drawbacks. First, it eliminates security policies enforced by the kernel, such as memory isolation or firewalling. Specific hardware extensions, i.e. an IOMMU and SR-IOV [13, 18], or software-based isolation are then required to maintain standard security levels. Second, kernel-bypass relies on dedicating CPU cores to poll incoming packets, trading off CPU resources for low latency. This prevents the cores from being shared with other applications even when the client load is low. Third, kernel-bypass requires an extensive re-engineering of the existing application in order to achieve high performance with a dedicated network stack.

In this paper, we propose BPF Memcached Cache (BMC) to address the kernel bottlenecks impacting Memcached. BMC focuses on accelerating the processing of small GET requests over UDP to achieve high throughput as previous work from Facebook [11] has shown that these requests make up a significant portion of Memcached traffic. Contrary to hardware-specific accelerators, BMC runs on standard hardware and thus can be deployed on infrastructure with heterogeneous hardware. BMC relies on a *pre-stack processing* approach that consists in intercepting requests directly from the network driver, before they are delivered to the standard network stack, and processing them using an in-kernel cache. This provides the ability to serve requests with low latency and to fall back to the Memcached application when a request cannot be treated by BMC. BMC can leverage modern network card features such as multi-queues to process multi-

ple requests in parallel (see Figure 1). In addition, the BMC cache uses a separate lock for each entry to introduce minimal overhead and allow performance to scale with the number of cores.

Running BMC at the kernel level raises safety issues as a bug in its implementation could put the entire system at risk. To address this issue, BMC is implemented using eBPF. The Berkeley Packet Filter (BPF) [35], and its extended version, eBPF, is a bytecode and a safe runtime environment offered by the Linux kernel to provide userspace an approach to inject code inside the kernel. The Linux kernel includes a static analyzer to check that the injected code is safe before it can be executed, which limits the expressiveness of the injected code. We show how to circumvent this limitation by partitioning complex functionality into small eBPF programs and by bounding the data that BMC processes. Using eBPF also allows BMC to be run without requiring any modification to the Linux kernel or to the Memcached application, making it easy to deploy on existing systems. The eBPF bytecode of BMC is compiled from 513 lines of C code and is JIT compiled by the Linux kernel. This results in BMC introducing a very low overhead into the OS network stack.

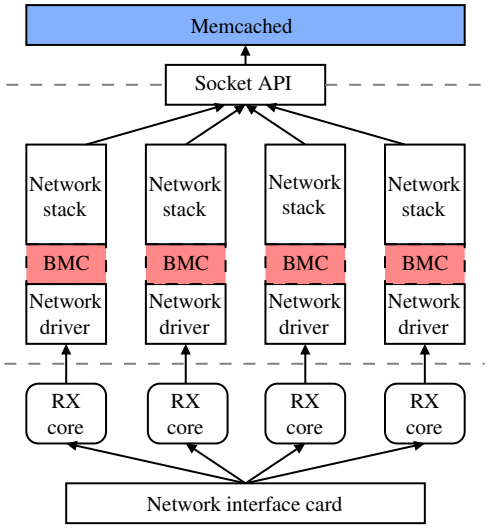


Figure 1: General architecture

The main results of this paper include:

- The identification of the bottlenecks of Memcached when processing requests over UDP. We propose MemcachedSR, a modified version of Memcached that uses the SO_REUSEPORT socket option to scale with the number of threads, improving throughput by 3x compared to the vanilla Memcached.
- The evaluation of BMC under our target workload consisting of small requests. In this setting, BMC improves the throughput by up to 6x with respect to MemcachedSR and by up to 18x with respect to vanilla Memcached.
- The evaluation of BMC under a non-target workload that consists of large requests not processed by BMC. In this setting, BMC has negligible overhead and does not deteriorate throughput with respect to MemcachedSR.
- The comparison of BMC with a dummy cache that shows that BMC's design is well suited for high throughput performance as it does not introduce unnecessary complexity.
- The comparison of Memcached running with BMC against a Memcached implementation based on Seastar, a networking stack for DPDK [4]. Our results show that Memcached with BMC achieves similar throughput to Seastar but uses 3x less CPU resources.

The rest of this paper is organized as follows. Section 2 provides background on Memcached and the OS network stack bottlenecks it suffers from. Section 3 describes BMC and its design. Section 4 discusses implementation details. Section 5 presents the experimental results. Section 6 discusses the generalization of BMC and its memory allocation challenges. Section 7 presents related work. Finally, Section 8 concludes the paper.

2 Background and motivation

This section describes the limitations of Memcached that motivate our work, and describes the eBPF runtime used to implement BMC.

2.1 Memcached

Memcached [8] is a mature in-memory key-value store traditionally used as a cache by web applications in a datacenter environment to speed up request processing and reduce the load on back-end databases. Because of its popularity, a lot of work has been put into optimizing it [33, 42].

A Memcached server operates on *items*, which are objects used to store a key and its associated value and metadata. Clients send requests to a Memcached server using a basic command-based protocol, of which GET and SET are the most important commands. A GET *key* command retrieves the value associated with the specified key if it is stored by Memcached and a SET *key value* command stores the specified key-value pair. A GET command can also be used as a multiget request when a client needs to retrieve multiple values at once. Requests can be sent using either the TCP or the UDP transport protocol.

The data management of Memcached has been well optimized and relies on slab allocation, a LRU algorithm and a hash table to allocate, remove and retrieve items stored in memory. Previous studies [14, 28] have shown that Memcached performance and scalability are heavily impaired by OS network stack bottlenecks, especially when receiving a large number of small requests. Since Facebook's production

workloads show a 30:1 distribution between GET and SET commands, Nishtala et al. [38] proposed using UDP instead of TCP for GET commands to avoid the cost of TCP processing.

To gain additional insight into the performance of a Memcached server using UDP to receive GET requests, we profiled the CPU consumption while trying to achieve maximum throughput (experimental setup is described in Section 5). As shown in Figure 2, more than 50% of Memcached's runtime is spent executing system calls. Moreover, the CPU usage of both sys_recvfrom and sys_sendmsg increases as more threads are allocated to Memcached. When eight threads are used by Memcached, the total CPU usage of these three system calls reaches 80%.

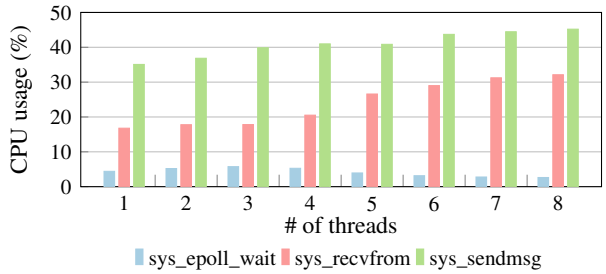


Figure 2: CPU usage of the three most used system calls by Memcached

Figure 3 shows the throughput of the vanilla Memcached application when varying the number of threads (and cores). The results show that vanilla Memcached does not scale and that its performance even deteriorates when more than four threads are used. Table 1 shows the top ten most time consuming functions measured by the *perf* tool while running Memcached with eight threads, all of them are kernel functions. The *native_queued_spin_lock_slowpath* and *__udp_enqueue_schedule_skb* functions account for a total of 28.63% of the processing time of our machine under test and are used to push packets to the UDP socket queue. The kernel's socket queues are data structures shared between the Memcached threads and the kernel threads responsible for the execution of the network stack, and therefore require lock protection. In case of Memcached, a single UDP socket is used and its queue is shared between the cores receiving packets from the NIC and the cores running the application, leading to lock contention. This lock contention is then responsible for the decrease in Memcached throughput.

To allow Memcached to scale with the number of threads, we have modified the version 1.5.19 of Memcached to use the SO_REUSEPORT socket option. The SO_REUSEPORT option allows multiple UDP sockets to bind to the same port. We refer to this modified Memcached as *MemcachedSR* in the rest of the paper. We use this option to allocate a UDP socket per Memcached thread and bind each socket to the same port. Received packets are then equitably distributed between each socket queue by the Linux kernel which reduces lock contention. As shown in Figure 3, MemcachedSR scales with the number of threads and achieves a throughput that is up to 3 times higher than the vanilla version of Memcached.

Despite the scalability of MemcachedSR, there is still room for improvement as Memcached requests still have to go through the whole network stack before they can be processed by the application.

Function	% CPU utilization
native_queued_spin_lock_slowpath	17.68%
__udp_enqueue_schedule_skb	10.95%
clear_page_erms	5.04%
udp4_lib_lookup2	3.23%
_raw_spin_lock	3.04%
fib_table_lookup	2.90%
napi_gro_receive	2.27%
nfp_net_rx	1.97%
i40e_napi_poll	1.32%
udp_queue_rcv_one_skb	1.14%

Table 1: Top ten most CPU-consuming functions on a Memcached server under network load

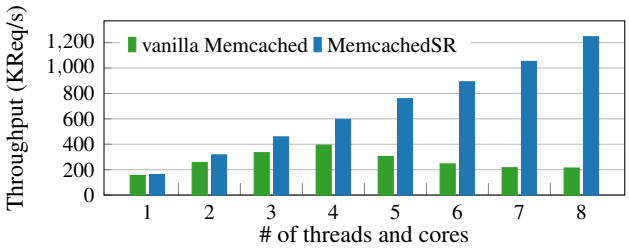


Figure 3: Vanilla Memcached vs. MemcachedSR

2.2 BPF

The Berkeley Packet Filter (BPF) [35] is an in-kernel interpreter originally designed to run packet filters from userspace using a reduced instruction set. BPF has evolved into the extended BPF (eBPF), which introduces a new bytecode and just-in-time compilation for improved performance. An eBPF program can be loaded from userspace by the Linux kernel and triggered by a specific kernel event. The eBPF program is then run whenever the event is triggered.

eBPF programs can maintain and access persistent memory thanks to kernel data structures called *BPF maps*. Maps are designed to store arbitrary data structures whose size must be specified by the user application at creation time. They can be used for communicating between different eBPF programs or between eBPF programs and user applications. Furthermore, eBPF programs can call a restricted set of kernel functions, called helpers, allowing eBPF programs to interact with the system and access specific kernel data (e.g. map data, time

since boot up). *Tail calls* allow an eBPF program to call another eBPF program in a continuation-like manner. The eBPF bytecode backend is supported by the Clang/LLVM compiler toolchain, which allows using the C language to write eBPF programs in a high-level language.

Because running user-space code inside the kernel can impact the system's security and stability, the Linux kernel calls the in-kernel eBPF verifier every time it loads an eBPF program to check if the program can be safely attached and executed. The goal of the verifier is to guarantee that the program meets two properties: safety, i.e., the program neither accesses unauthorized memory, nor leaks kernel information, nor executes arbitrary code, and liveness, i.e., the execution of the program will always terminate.

To analyze an eBPF program, the verifier creates an abstract state of the eBPF virtual machine [7]. The verifier updates its current state for each instruction in the eBPF program, checking for possible out of bounds memory accesses or jumps. All conditional branches are analyzed to explore all possible execution paths of the program. A particular path is valid if the verifier reaches a *bpf exit* instruction and the verifier's state contains a valid return value or if the verifier reaches a state that is equivalent to one that is known to be valid. The verifier then backtracks to an unexplored branch state and continues this process until all paths are checked.

Because this verification process must be guaranteed to terminate, a complexity limit is enforced by the kernel and an eBPF program is rejected whenever the number of explored instructions reaches this limit. Thus, the verifier incurs false positives, i.e. it can reject eBPF programs that are safe. In Linux 5.3, the kernel version used to implement BMC, this limit is set to 1 million instructions. Other parameters, such as the number of successive branch states, are also used to limit path explosion and the amount of memory used by the verifier. Since Linux 5.3, the verifier supports bounded loops in eBPF programs by analyzing the state of every iteration of a loop. Hence, the verifier must be able to check every loop iteration before hitting the previously mentioned instruction complexity limit. This limits the number of loop iterations as well as the complexity of the instructions in the loop body. Moving data of variable lengths between legitimate memory locations requires a bounded loop and conditional instructions to provide memory bounds checking, which in turn increase the complexity of an eBPF program. Finally, eBPF does not support dynamic memory allocation, instead eBPF programs have to rely on eBPF maps (array, hashmap) to hold a fixed number of specific data structures.

Because of all these limitations, eBPF is currently mostly used to monitor a running kernel or to process low-layer protocols of network packets (i.e. L2-L4). Processing application protocols is more challenging but is required to allow the implementation of more complex network functions [36].

3 Design

In this section, we present the design of BMC, a safe in-kernel accelerator for Memcached. BMC allows the acceleration of a Memcached server by caching recently accessed Memcached data in the kernel and by relying on a *pre-stack processing* principle to serve Memcached requests as soon as possible after they have been received by the network driver. This approach allows BMC to scale to multicore architectures by leveraging modern NIC's multi-queue support to run BMC on each individual core for each received packet. The execution of BMC is transparent to Memcached, and Memcached does not need any modification to benefit from BMC. In the rest of this section, we first present the pre-stack processing approach. We then describe the BMC cache and how its coherence is insured.

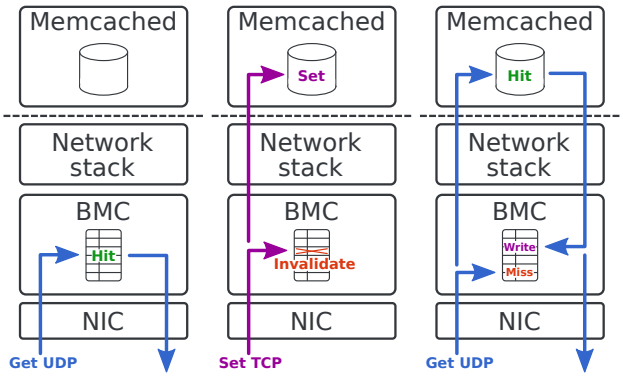
3.1 Pre-stack processing

BMC intercepts network packets at the network-driver level to process Memcached requests as soon as possible after they have been received by the NIC. BMC filters all network packets received by the network driver based on their destination port to only process Memcached network traffic. It focuses on processing GET requests using the UDP protocol and SET requests using the TCP protocol. Figure 4 illustrates how pre-stack processing allows BMC to leverage its in-kernel cache to accelerate the processing of Memcached requests.

When processing a GET request (4a), BMC checks its in-kernel cache and sends back the corresponding reply if it finds the requested data. In that case, the network packet containing the request is never processed by the standard network stack, nor the application, freeing CPU time.

SET requests are processed by BMC to invalidate the corresponding cache entries and are then delivered to the application (4b). After a cache entry has been invalidated, a subsequent GET request targeting the same data will not be served by BMC but rather by the Memcached application. BMC always lets SET requests go through the standard network stack for two reasons. First, it enables reusing the OS TCP protocol implementation, including sending acknowledgments and retransmitting segments. Second, it ensures SET requests are always processed by the Memcached application and that the application's data stays up-to-date. We choose not to update the in-kernel cache using the SET requests intercepted by BMC because TCP's congestion control might reject new segments after its execution. Moreover, updating the in-kernel cache with SET requests requires that both BMC and Memcached process SET requests in the same order to keep the BMC cache consistent, which is difficult to guarantee without a overly costly synchronization mechanism.

When a miss occurs in the BMC cache, the GET request is passed to the network stack. Then, if a hit occurs in Memcached, BMC filters the outgoing GET reply to update its cache (4c).



(a) Lookup (b) Invalidation (c) Update

Figure 4: BMC cache operations

Pre-stack processing offers the ability to run BMC on multiple cores concurrently. BMC can benefit from modern NIC features such as multi-queue and RSS to distribute processing among multiple CPU cores. The set of cores used to run BMC can also be fine-tuned in order to share a precise memory level (CPU caches, NUMA node, etc.). The performance of BMC can efficiently scale by configuring NICs to use multiple RX queues and mapping them to different cores. Pre-stack processing also enables running specialized code without having to modify existing software. Contrary to kernel-bypass, this approach does not require a whole NIC to be given to a userspace process and other applications can share the network hardware through the kernel network stack as usual.

3.2 BMC cache

The BMC cache is designed as a hash table indexed by Memcached keys. It is a direct-mapped cache, meaning that each bucket in the hash table can only store one entry at a time. BMC uses the 32-bit *FNV-1a* [19] hash function to calculate the hash value. Because this is a rolling hash function that operates on a single byte at a time, it allows BMC to compute the hash value of a key while parsing the Memcached request. The hash value is reduced to an index into the cache table by using the modulo operator. Each cache entry contains a valid bit, a hash value, a spin lock, the actual stored data and the size of the data. This cache design offers constant-time complexity for lookup, insertion and removal operations. To validate a cache hit, BMC checks that the valid bit of a cache entry is set and that the stored key is the same as that of the processed request.

The BMC cache is shared by all cores and and does not require a global locking scheme since its data structure is immutable. However, each cache entry is protected from concurrent access using a spin lock.

4 Implementation

This section explains how BMC deals with the eBPF limitations to meet the required safety guarantees.

4.1 Bounding data

The verification of a loop contained in a single program may hit the maximum number of eBPF instructions the verifier can analyze. Loop complexity depends on the number of iterations and the complexity of the body. To make the verification of loops possible, BMC bounds the data it can process. It first limits the length of Memcached keys and values. BMC uses a loop to copy keys and values from a network packet to its cache, and vice-versa. For every memory copy, BMC must guarantee that it neither overflows the packet bounds nor overflows the cache memory bounds using fixed data bounds. Bounds checking then increases the loop complexity. To ensure the complexity of a single eBPF program does not exceed the maximum number of instructions the verifier can analyze, we empirically set the maximum key length BMC can process to 250 bytes and the maximum value length to 1000 bytes. Requests containing keys or values that exceed these limits are transmitted to the Memcached application. We also limit to 1500 the number of individual bytes BMC can read from a packet's payload in order to parse the Memcached data, bounding the complexity of this process. According to Facebook's workload analysis [11], about 95% of the observed values were less than 1000 bytes. Moreover, the Memcached protocol sets the maximum length of keys to 250 bytes. Hence, bounding the BMC data size does not have a big practical impact.

4.2 Splitting complex functions

In order to avoid reaching the limits of the eBPF verifier, BMC's functional logic is separated into multiple small eBPF programs, as each eBPF program is checked for safety independently. Each program then relies on *tail calls* to jump to the next program and continue packet processing without interruption. Linux limits the maximum number of successive tail calls to 32, preventing infinite recursion. However BMC uses at most three successive tail calls.

BMC is implemented using seven eBPF programs that are written in C code and are compiled to eBPF bytecode using Clang and LLVM version 9. The processing logic of BMC is split into two chains: one chain is used to process incoming Memcached requests and the other is used to process outgoing Memcached replies. Figure 5 illustrates how BMC's eBPF programs are divided. BMC's eBPF programs consist of a total of 513 lines of C code.

4.2.1 Incoming chain

The incoming chain is composed of five eBPF programs. It is attached to the XDP [25] driver hook and is executed

Program name	# of eBPF instructions	# of analyzed instructions	analysis time (µs)	# of CPU instructions
rx_filter	87	31 503	11 130	152
hash_keys	142	787 898	290 588	218
prepare_packet	178	181	47	212
write_reply	330	398 044	132 952	414
invalidate_cache	163	518 321	246 788	224
tx_filter	61	72	43	104
update_cache	125	345 332	95 615	188

Table 2: Complexity of BMC's programs. Column 2 represents the number of eBPF bytecode instructions of the program compiled from C code. Columns 3 and 4 respectively show the number of eBPF bytecode instructions processed by the Linux verifier and the time spent for this analysis. Column 5 shows the number of CPU instructions after JIT compilation.

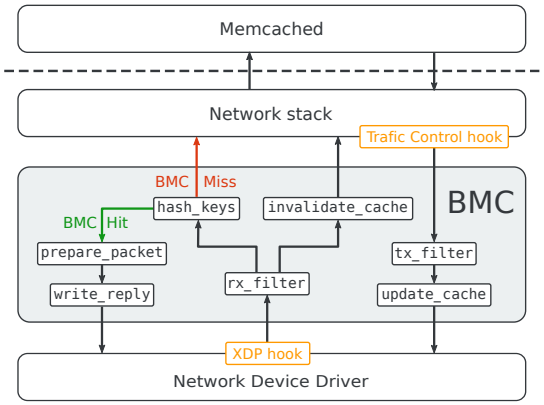


Figure 5: Division of BMC into seven eBPF programs

whenever a new packet is processed by the network driver. This hook is the earliest point in the network stack at which an eBPF program can be attached and allows BMC to use pre-stack processing to save the most CPU cycles by responding to Memcached requests as soon as possible.

rx_filter. The goal of this first eBPF program is to filter packets corresponding to the Memcached traffic using two rules. The first rule matches UDP packets whose destination port corresponds to Memcached's and whose payload contains a GET request. The second rule matches TCP traffic whose destination port also corresponds to Memcached's. The incoming chain branches based on which rule matches. If neither rule matches, the packet is processed by the network stack as usual.

hash_keys. This program computes hashes for every Memcached GET key contained in the packet. It then checks the corresponding cache entries for any cache hit and saves the key hashes that have been hit in a per-cpu array used to store context data for the execution of the chain.

prepare_packet. This eBPF program increases the size of the received packet and modifies its protocol headers to prepare the response packet, swapping the source and destination Ethernet addresses, IP addresses and UDP ports. It then calls the last eBPF program of this branch of the chain. The maximum number of bytes BMC can add to the packet is limited by the network driver implementation. In our current implementation of BMC, this value is set to 128 bytes based on the different network drivers BMC attaches to, and it can be increased to a higher value to match other network driver implementations.

write_reply. This eBPF program retrieves a key hash saved in the per-cpu array to copy the corresponding cache entry to the packet's payload. If the table contains multiple key hashes, this eBPF program can call itself up to 30 times to copy as many items as possible in the response packet. Finally, this branch of the incoming chain ends by sending the packet back to the network.

invalidate_cache. The second branch of the incoming chain handles Memcached TCP traffic and contains a single eBPF program. This program looks for a SET request in the packet's payload and computes the key hash when it finds one to invalidate the corresponding cache entry. Packets processed by this branch of the incoming chain are always transmitted to the network stack so that Memcached can receive SET requests and update its own data accordingly.

4.2.2 Outgoing chain

The outgoing chain is composed of two eBPF programs to process Memcached responses. It is attached to the Traffic Control (TC) egress hook and is executed before a packet is sent to the network.

tx_filter. The first eBPF program of this chain serves as a packet filter and applies a single rule on outgoing packets. The rule matches UDP packets whose source port corresponds to Memcached's. In this case the second eBPF program of the chain is called, otherwise the packet is sent to the network as usual.

update_cache. The second eBPF program checks if the packet's payload contains a Memcached GET response. If positive, its key is used to index the BMC cache and the response data is copied in the corresponding cache entry. The network stack then carries on its execution and the Memcached response is sent back to the network.

Table 2 provides complexity metrics for each eBPF program. For the most complex ones, the number of eBPF instructions the Linux verifier has to analyze to ensure their safety is a thousand times higher than their actual number of instructions. The table also shows that it is necessary to divide BMC's logic into multiple eBPF programs to avoid reaching the limit of 1,000,000 instructions that can be analyzed by the Linux verifier.

5 Evaluation

In this section, we evaluate the performance of MemcachedSR running with BMC. We aim to evaluate the throughput gain offered by BMC and how performance scales with the number of cores when processing a target workload that consists of small UDP requests. We also evaluate MemcachedSR with BMC on a non-target workload to study the overhead and impact of BMC on throughput when it intercepts Memcached requests but does not cache them. We show that the increase in throughput can be obtained without allocating additional memory, and that the cache memory can be partitioned between the Memcached application and BMC. We compare BMC with a dummy cache implementation and show that its design is efficient for high performance. Finally, we compare MemcachedSR running with BMC to Seastar, an optimized networking stack based on DPDK. We study their performance using our target workload and a workload that uses both TCP and UDP requests. We also measure their CPU resource consumption for an equivalent client load and show that BMC allows saving CPU resources.

5.1 Methodology

Platform. Our testbed consists of three machines: one acting as the Memcached server under test, and two as the clients. The server machine is equipped with a dual socket motherboard and two 8-core CPUs (Intel Xeon E5-2650 v2 @ 2.60 GHz) with HyperThreading disabled, 48 GB of total memory and two NICs (one Intel XL710 2x40GbE and one Netronome Agilio CX 2x40GbE). The other two machines are used as clients to send traffic and are equipped with the same Intel Xeon CPU and an Intel XL710 2x40GbE NIC. One client is connected back to back to the server using its two network ports while the other client is connected using a single port. In total, the server machine uses three network ports to receive traffic from the clients. In all experiments, the server machine runs Linux 5.3.0.

Target workload and Method. Our target workload is the following: the client applications generate skewed workloads based on established Memcached traffic patterns [11]. Clients use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same used in Yahoo Cloud Serving Benchmark (YCSB) [17]. MemC3 [21] is an in-memory key-value store that brings carefully designed algorithms and data structures to Memcached to improve both its memory efficiency and scalability for read-mostly workloads. Similarly to the evaluations performed in the MemC3 paper, our workload consist of a population of 100 million distinct 16-byte keys and 32-byte values. By default, we allocate 10 GB of memory for the Memcached cache and 2.5 GB for the BMC cache. With this amount of memory, the Memcached cache can hold about 89 million items while the BMC cache can hold 6.3 million. Hence, some items can be missing from both the BMC and the Memcached cache. The memory allocated to both Memcached and BMC is not only used for keys and values but also stores metadata. For each cache entry in BMC, 17 bytes are used as metadata. Before each experiment, the clients populate Memcached's cache by sending a SET request for every key in the population. Note that this does not populate BMC's cache as it is only updated when the application replies to GET requests.

The client applications send requests at the rate of 12 million requests per second (Req/s) in an open-loop manner in order to achieve the highest possible throughput and highlight bottlenecks. A total of 340 clients are simulated to efficiently distribute requests among multiple cores on the server by leveraging the NICs' multi-queue and RSS features. We further refer to these cores as *RX cores*. We limit our evaluations to a maximum of 8 RX cores on a single CPU to enforce NUMA locality with the NICs.

Table 3 summarizes this target workload as well as other default evaluation settings that are used in the following experiments unless otherwise specified.

Key distribution	Zipf (0.99)
Key size	16 bytes
Value size	32 bytes
Key population	100 million
BMC to Memcached cache size ratio	25%
Number of Memcached application threads	8
Number of RX cores	8

Table 3: MemC3-like evaluation settings

5.2 Throughput

Target workload. We evaluate the throughput of Memcached under three configurations: vanilla Memcached alone, MemcachedSR alone and MemcachedSR with BMC. We also evaluate how these three configurations scale with the number of cores. We allocate the same CPU resources for all three configurations. For all configurations, we vary the number of threads the application uses to process requests simultaneously and dedicate cores to the application by pinning its threads. For MemcachedSR with BMC, we also vary the number of queues configured on each of the server's NICs and use the same cores to handle interrupts. This allows BMC to be executed by each core serving interrupts in parallel. For

the vanilla Memcached application alone and MemcachedSR alone, 8 cores are used to execute the network stack.

Figure 6 shows the throughput achieved by these three configurations. As mentioned in Section 2.1, the vanilla Memcached application does not scale due to the socket lock contention; at best it achieves 393K requests per second using 4 cores. MemcachedSR offers better scalability and achieves 1.2M requests per second when using 8 cores. For MemcachedSR with BMC, the overall system throughput is split between requests answered using the BMC cache and requests handled by the application. When running on a single core, MemcachedSR with BMC achieves 1M requests per second, which is 6x the throughput of both vanilla Memcached and MemcachedSR. When BMC runs on 8 cores, the server achieves a throughput of 7.2M requests per second, 6.3 million being processed by BMC, the rest being processed by Memcached. This is 18x better performance with respect to vanilla Memcached and 6x with respect to MemcachedSR.

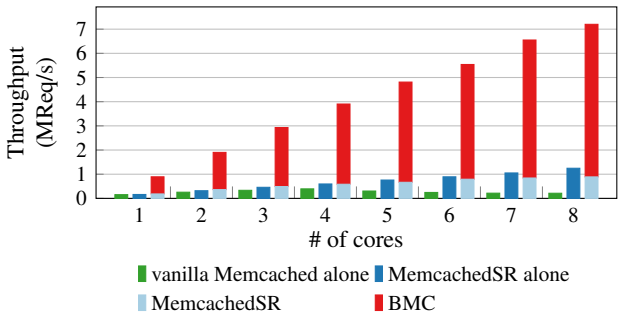


Figure 6: Throughput of BMC

Worst-case workload. We now change our workload to use 8KB Memcached values instead of 32 bytes. This is BMC's worst-case workload since the Memcached requests are still analyzed by BMC but the values are too large to be stored in its cache, thus the Memcached application is always used to serve the requests. To study the impact of the additional processing of BMC, we compare the throughput of the MemcachedSR application alone and that of the MemcachedSR application running with BMC. Figure 7 shows that BMC's additional processing has negligible overhead and does not significantly deteriorate the application throughput.

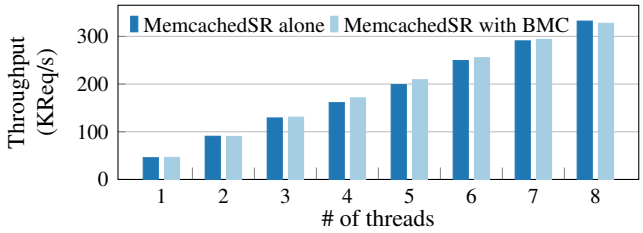


Figure 7: Throughput under a worst-case workload

We now modify the workload so that part of it are requests that BMC targets while the rest is for 8KB values. We then and evaluate how varying the ratio of the target requests affects throughput. As shown in Figure 8, BMC improves throughput by 4x compared to MemcachedSR alone when the workload consists of 25% of targeted requests even though it does not speed up the majority of requests received. This shows that BMC is valuable even when the workload contains few target requests and that BMC further improves throughput as the ratio of target requests increases.

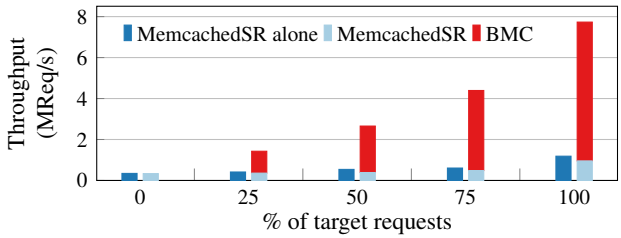


Figure 8: MemcachedSR vs. MemcachedSR with BMC throughput for varying request size distributions.

5.3 Cache size

We then evaluate the impact of BMC's cache size on throughput. In this experiment, we use a total of 10 GB of memory and split it between the Memcached cache and the BMC cache, varying the distribution from 0.1% of memory allocated to BMC to a maximum of 40%. The latter corresponds to a size of 4 GB for the BMC cache, which is the maximum size accepted by the Linux kernel for the allocation of a single eBPF map [5]. The results are shown in Figure 9 where the total system throughput is broken down into hits in the BMC cache, and hits and misses in the application cache. For all distribution schemes tested, there is an increase in performance compared to running MemcachedSR alone. The best throughput is achieved when BMC uses 25% of the total memory. In this case, the BMC cache size is well-suited to store the hottest items of the Zipf distribution. Throughput decreases from 25% to 40% because the Memcached cache hit rate shrinks from 89% to 43% as its cache gets smaller. This causes the BMC cache hit rate to diminish as well because only responses from Memcached cache hits allow entries of the BMC cache to be updated. When only 0.1% (10 MB) of the memory is used by the BMC cache, throughput is multiplied by 2.3 compared to the best throughput achieved by MemcachedSR alone, showing that BMC offers good performance even with minimal memory resources.

5.4 BMC processing latency

We now evaluate the overhead induced by a BMC cache miss, which characterizes the worst-case scenario since BMC processes a request and executes additional code that does not lead to any performance benefit. To characterize this overhead,

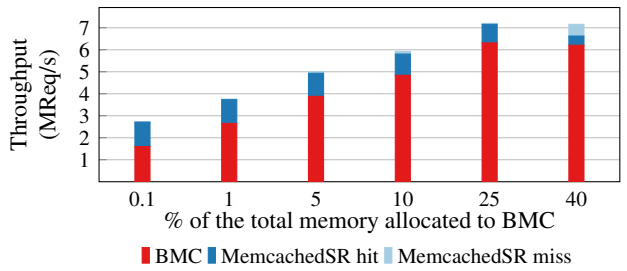


Figure 9: System throughput under various memory partition schemes

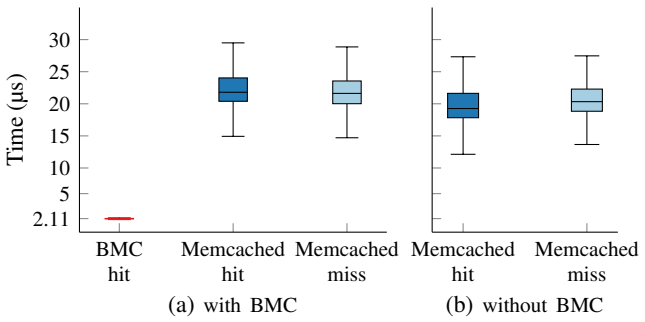


Figure 10: Time to receive, process and reply to a request.

we use *kprobes* to measure the total time required to receive, process and reply to a Memcached request. The *nfp_net_rx* and *nfp_net_tx* driver functions are respectively instrumented to record the time at which a request is received and the corresponding reply is sent back to the network. In this experiment, a single client machine is used to send requests to the server and a single Memcached key is used to ensure that a cache result is always either a hit or a miss. After sending a request, the client always waits for the server's reply to make sure the server does not process more than one request at a time.

Figure 10 shows the time distribution of 100,000 measurements for cache hits and misses separately. Figure 10a shows the distributions of MemcachedSR running with BMC as well as BMC hits. For Memcached hits, the valid bit of BMC's cache entries is never set to ensure BMC lookups result in a cache miss and that the request is always processed by the Memcached application. However, the BMC cache is still updated to measure additional data copies. The median of the distribution of BMC cache hits is 2.1 µs and that of Memcached cache hits and misses are respectively 21.8 and 21.6 µs. Hence, a BMC cache hit can reduce by 90% the time required to process a single request. Running the same experiment on MemcachedSR without BMC (Figure 10b) shows that the processing time of both Memcached hits and misses is lower by about 1 µs. This shows that BMC has a negligible processing overhead compared to the total time required for the execution of the Linux network stack and the Memcached application. Moreover, this additional processing time is entirely recovered by a single BMC cache hit.

Next we study the impact of the processing time of the kernel cache on its throughput. To do so we have implemented a dummy cache that always replies the same response and whose processing time can be parameterized using a empty loop. This dummy cache is implemented using a single eBPF program and is attached to the XDP network-driver hook just like BMC. Figure 11 shows the throughput of the dummy cache while varying its processing time and compares it with the actual BMC cache perfoming cache hits. This experiment demonstrates that the cache throughput is highly dependent on its processing time: increasing the processing time from 100 ns to 2000 ns decreases throughput by a factor of 4.5. The average time to perform a cache hit in BMC is fairly close to the time of the dummy cache with no additional processing time; this shows that choosing simple and fast algorithms for BMC's cache design introduces little processing overhead and contributes to its high throughput performance. Implementing overly complex algorithms may lead to a sharp drop in performance. Hence, adding new features to BMC, such as an eviction algorithm, must be well thought out to result in an improved hit rate that compensates for the additional processing time.

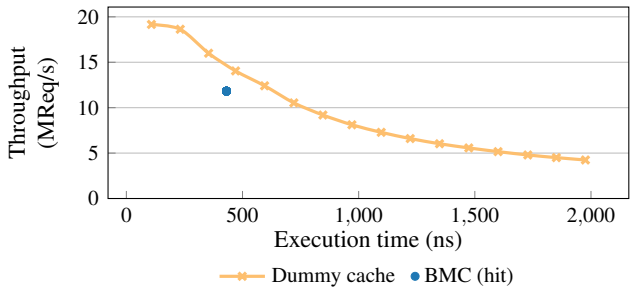


Figure 11: Execution time impact on throughput.

5.5 Impact on concurrent network applications.

As BMC intercepts every network packet before the OS network stack, its execution may have a performance impact on other networking applications running on the same host. To study this impact, we use *iperf* to transfer 10GB of data over TCP from one client machine to the server machine while serving Memcached requests, and measure the time required to complete this transfer. This experiment is conducted for MemcachedSR alone and MemcachedSR with BMC, while varying clients' request throughput. Figure 12 shows the results. When there is no load, the baseline transfer time is 4.43 seconds. When the Memcached load rises, the time required to complete the data transfers increases since the cores are busy processing incoming Memcached traffic in addition to

the *iperf* traffic, which leads to an increase in TCP retransmission rate and UDP packet drops. When using BMC, the transfer time is lowered when the server is under load as CPU resources are saved when BMC processes Memcached requests in place of the application. For a Memcached load of 5000K Req/s, BMC allows *iperf* to complete its transfer 83% faster compared to the configuration in which Memcached processes the requests alone and the OS network stack is always executed.

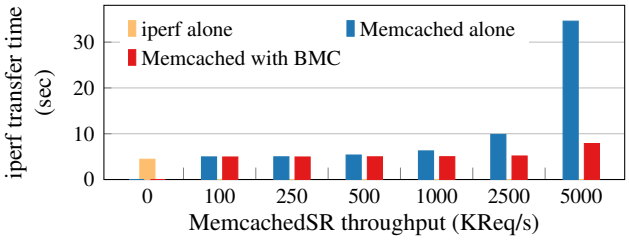


Figure 12: Time required to transfer 10GB of data using *iperf* under client load.

5.6 Kernel-bypass comparison

We now compare MemcachedSR with BMC against the Memcached implementation from Seastar [2], a framework for building event-driven applications that comes with its own network stack built on top of DPDK.

Target workload. In this experiment, we evaluate the throughput of Seastar and compare the results with MemcachedSR running with BMC. We perform the experiment using a single client to generate our target UDP workload as Seastar does not support multiple network interfaces. This single client alone generates 4.5 million requests per second. Figure 13 shows the throughput of Seastar and MemcachedSR with BMC when varying the number of cores. BMC is able to process the workload generated by a single client machine using 4 cores. Using the same number of cores, Seastar achieves 443K requests per second. Seastar's throughput increases to 946K requests per second when using 8 cores. We are not sure why Seastar's throughput drops when using 2 cores; our investigations revealed that this only happens when Seastar receives UDP packets and that Seastar performs best when it processes Memcached requests over TCP.

Workload mixing UDP and TCP requests. As our preliminary investigation shows that Seastar performs best on TCP, we change our workload to send half of the Memcached requests with TCP while the other half keeps using UDP. This workload coincides with a Memcached deployment for which the protocol used by clients cannot be anticipated. Figure 14 shows that the throughput of both configurations scales with the number of cores. Seastar's high-performance TCP stack enables its Memcached implementation to process 2.3 million requests per second when using 8 cores. Accelerating the processing of UDP requests allows MemcachedSR with BMC

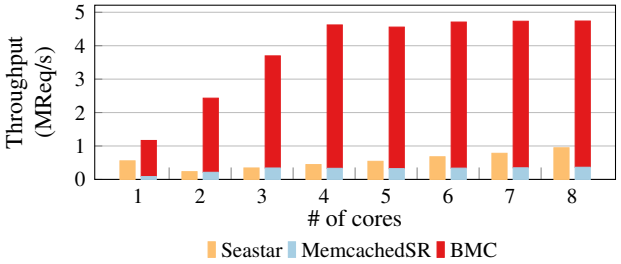


Figure 13: Seastar vs. BMC throughput

to achieve similar throughput when using 3 cores. Increasing the number of cores does not increase the throughput of MemcachedSR with BMC as the client TCP workload generation becomes the bottleneck.

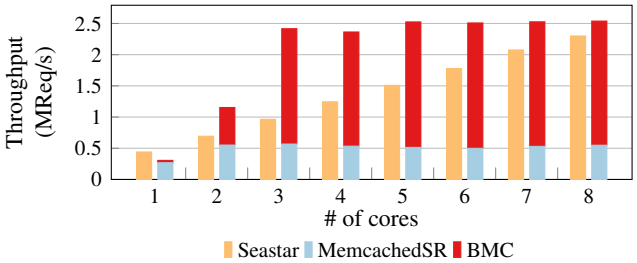


Figure 14: Seastar vs. BMC throughput when mixing TCP and UDP requests

CPU usage. We then measure the CPU usage of both MemcachedSR with BMC and Seastar for different client loads. In both configurations we use a total of 8 CPU cores to process the workload. For MemcachedSR with BMC, we use 6 RX cores and pin the Memcached threads to the two remaining cores. This configuration offers the best performance and allows us to measure the CPU usage of the Memcached application and the network stack (including BMC) separately. The CPU usage is measured on each core for 10 seconds using the *mpstat* tool. Figure 15 shows the average CPU core usage per core type (Seastar, MemcachedSR and BMC). The results show that Seastar always uses 100% of its CPU resources, even when throughput is low. This is because DPDK uses poll mode drivers to reduce the interrupt processing overhead when packets are received by the NIC. The CPU usage of MemcachedSR with BMC scales with the load thanks to the interrupt-based model of the native Linux drivers BMC builds upon. As shown in Figure 14, Seastar can process 2.3 million requests per second when using 8 cores, Figure 15 shows that MemcachedSR with BMC consumes 33% of the CPU (91% of the two Memcached cores and 13% of the six RX cores) to achieve similar throughput. Therefore, using Memcached with BMC saves CPU resources that can be used by other tasks running on the same system when the workload is low.

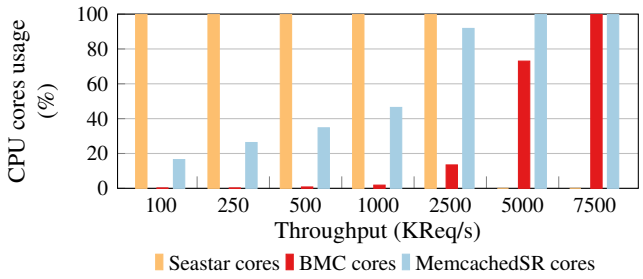


Figure 15: CPU usage of BMC compared to Seastar

6 Discussion

Although the BMC cache is fairly generic and can store any datatype, most of BMC is specialized to filtering and processing Memcached requests. Applying in-kernel caching to another key-value store like Redis [9] would then require implementing new BPF programs to process Redis's RESP protocol. The main challenge of adapting BMC to Redis is to ensure compatibility with the TCP protocol. Because Redis requests are only transmitted over TCP, it its mandatory to either send acknowledgements or reuse the existing TCP kernel implementation by intercepting packets past the TCP stack. As Redis is more focused on functionality and Memcached on performance, an in-kernel cache for Redis will need to use more BPF programs to implement the minimal subset of Redis commands required to ensure cache coherence with the application. Just like BMC, some of the functionalities can be left to the application to focus on accelerating the most frequent request types. Redis usually performs worse than Memcached when processing a large volume of requests because it is single-threaded, hence we expect the throughput speed-up to be even higher than for Memcached.

Although static memory allocation enables the verification of BMC's eBPF programs, it also wastes memory. BMC suffers from internal fragmentation because each cache entry is statically bounded by the maximum data size it can store and inserting data smaller than this bound wastes kernel memory. The simplest approach to reduce memory fragmentation would be to fine-tune the bound of the cache entries to minimize the amount of fragmented memory. A more flexible approach would be to reuse fragmented memory to cache additional data. Each cache entry would then be able to store multiple data, making BMC a set-associative cache for which the number of slots varies according to the size of the stored data to reduce memory fragmentation. The fact that BMC is a non-exclusive cache also leads to memory loss since some data is duplicated between the BMC cache and Memcached. This duplication occurs for the majority of data that is not frequently accessed. On the other hand, frequently accessed data are eventually discarded from the application by Memcached's LRU algorithm because the BMC cache is used instead. Ideally, BMC should directly access the application's cache memory to avoid any duplication, however, this requires browsing Memcached data structures which could create security vulnerabilities.

The XDP driver hook leveraged by BMC requires support in the NIC driver. With no XDP driver support, BMC can still be used with the generic Linux kernel hook but its performance will not be as high. However, this is not a critical concern as most of the drivers for high-speed network interfaces support XDP.

7 Related Work

This section discusses the most relevant related work in the field of optimization of the network stack and Memcached.

Programmable hardware switches. Recent advances in programmable hardware switches with languages like P4 [39] have raised significant interest on offloading network processing operations into the network. NetCache [27] implements an in-network key-value cache on Barefoot Tofino switches [10]. NetCache uses switch lookup tables to store, update and retrieve values. To access the key-value store, clients have to use a specific API to translate client requests into NetCache queries. Switch KV [30] and FlairKV [41] leverage programmable ASICs to implement a caching solution also acting as a load balancer. Switch KV uses an efficient routing algorithm to forward client queries to the right server. The OpenFlow protocol is used to install routes to cached objects and invalidate routes to recently modified ones. FlairKV accelerates GET queries by intercepting every SET query and the corresponding reply to detect unmodified objects. While these approaches leverage ASIC optimizations to offer high throughput and low latency, they consume switch memory, TCAM and SRAM, which is an expensive resource primary reserved for packet forwarding. Thus, using lookup table resources to store key-value data exposes the entire network to bottlenecks and failures.

FPGA and NIC offloading. Examples of key-value store applications offloaded to FPGA include TSSP [32] which implements part of the Memcached logic, i.e., the processing of GET requests over UDP, on a Xilinx Zynq SoC FPGA. A more complete FPGA Memcached implementation [15] supports processing of SET and GET requests over both TCP and UDP protocols. Similar work [16] deployed and evaluated an FPGA Memcached application within the public Amazon infrastructure. These approaches achieve high throughput with up to 13.2 million RPS with 10GbE link but they are constrained by the FPGA programming model, which requires replacing the Memcached protocol by more FPGA-compatible algorithms. KV-Direct [29] leverages Remote Direct Memory Access (RDMA) technology on NICs to update data directly on the host memory via PCIs. With this approach, KV-Direct alleviates CPU bottlenecks at the expense of PCI resources. NICA [20] introduces a new hardware-software co-designed

framework to run application-level accelerators on FPGA NICs. NICA enables accelerating Memcached by serving GETs directly from hardware using a DRAM-resident cache and achieves similar performance to BMC since it still requires host processing to handle cache misses. NICached [40] proposes to use Finite State Machines as an abstract programming model to implement key-value store applications on programmable NICs. With this abstraction, NICached can be implemented with different languages and platforms: FPGA, eBPF, P4 and NPU-based NICs. NICached is the closest work to BMC but it targets NICs and does not propose an implementation of this model.

Compared to hardware approaches, BMC offers competitive performance, does not make use of expensive hardware resources such as SRAM and does not require hardware investment and software re-engineering.

Kernel-bypass. A kernel-bypass version of Memcached has been built on top of StackMap [44], an optimized network stack that achieves low latency and high throughput by dedicating hardware NICs to userspace applications. Using StackMap improves vanilla Memcached throughput by 2x in the most favorable scenario. MICA [31] employs a full kernel-bypass approach to process all key-value store queries in user space. MICA avoids synchronization by partitioning hash-maps among cores. MICA relies on a specific protocol that requires client information to map queries to specific cores and is not compatible with Memcached. To the best of our knowledge, MICA is the fastest software key value store application with a throughput of 77 million RPS on a dual-socket server with Intel Xeon E5-2680 processors. MICA is built with the DPDK library making MICA inherit most of DPDK's constraints: dedicated CPU cores to pull incoming packets, reliance on the hardware for isolation and requiring entirely re-engineering existing applications.

Compared to MICA, BMC achieves lower throughput but keeps the standard networking stack, does not implement any modification of clients and saves CPU resources.

Memcached optimizations Prior works [33, 42] have proposed to change the locking scheme of a former Memcached version to remove bottlenecks that impacted performance when running a large number of threads. To scale Memcached network I/O, MegaPipe [24] replaces socket I/O by a new channel-based API. Hippos [43] uses the Netfilter hook with a kernel module to serve Memcached requests from the Linux kernel, but does not ensure the safety of its kernel module and requires modifications to Memcached's source code to update its kernel cache.

eBPF verification. Gershuni et al. [23] have proposed a new verifier based on abstract interpretation in order to scale the verification of eBPF programs with loops. The authors showed that they could verify programs with small bounded loops and eliminate some false positives, however, their implementation has a time complexity about 100 times higher than the current Linux verifier, and uses from 100 to 1000 times more memory. Serval [37] introduces a general purpose and reusable approach to scale symbolic evaluation by using symbolic profiling and domain knowledge to provide symbolic optimizations. However, Serval does not consider domain specific symbolic evaluation. For example, the Linux verifier is capable of inferring register types based on the attach type of an eBPF program. Type inference then allows the Linux verifier to check the type correctness of the parameters passed to a helper function. Without this specific symbolic evaluation, Serval cannot ensure a precise analysis of eBPF programs and therefore cannot be used in place of the Linux verifier.

eBPF usage. The excitement around eBPF led the Linux community to put effort into the development of bpfilter, an eBPF alternative to iptables. Bpfilter automatically translates iptable rules into eBPF programs. Rules are written and supplied by users in C code, then translated to eBPF programs and attached to different kernel hooks including the XDP hook. eBPF is also extensively used in industry for fast packet processing. Cloudflare uses eBPF to replace their complex infrastructure filtering rules by eBPF programs. As an example, Cloudflare's DDoS mitigation solution uses XDP in L4Drop, a module that transparently translates iptable DDoS mitigation rules into eBPF programs [6]. These eBPF programs are pushed to the edge servers located in Cloudflare's Points of Presence (PoPs) for automatic packet filtering. Facebook developed Katran, an XDP based L4 Load balancer. Katran consists of a C++ library and an XDP program deployed in backend servers in Facebook's infrastructure PoPs.

8 Conclusion

We present BMC, an in-kernel cache designed to improve performance of key-value store applications. BMC intercepts application queries at the lowest point of the network stack just as they come out of the NIC to offer high throughput and low latency with negligible overhead. When compared to user space alternatives, BMC shows comparable performance while saving computing resources. Moreover, BMC retains the Linux networking stack and works in concert with the user space application for serving complex operations. We believe that the BMC design can motivate the emergence of new system designs that make it possible to maintain the standard Linux networking stack while offering high performance.

BMC focuses on the optimization of Memcached because it is a performance-oriented key-value store. As a future work, we plan to apply the design of BMC to other popular key-value store applications such as Redis.

Acknowledgments

We thank Julia Lawall, Paul Chaignon, Antonio Barbalace, and the anonymous reviewers for their valuable feedback.

References

[1] Amazon ElastiCache - In-memory data store and cache, 2020. URL: https://aws.amazon.com/elasticache.

[2] GitHub - scylladb/seastar: High performance server-side application framework, 2020. URL: https://github.com/scylladb/seastar.

[3] GitHub - twitter/twemcache: Twemcache is the Twitter Memcached, 2020. URL: https://github.com/twitter/twemcache.

[4] Home - DPDK, 2020. URL: https://www.dpdk.org/.

[5] kernel/bpf/syscall.c - Linux source code (v5.6) - Bootlin, 2020. URL: https://elixir.bootlin.com/linux/v5.6/source/kernel/bpf/syscall.c#L373.

[6] L4Drop: XDP DDoS Mitigations, 2020. URL: https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations.

[7] Linux Socket Filtering aka Berkeley Packet Filter (BPF) — The Linux Kernel documentation, 2020. URL: https://www.kernel.org/doc/html/latest/networking/filter.html#ebpf-verifier.

[8] memcached - a distributed memory object caching system, 2020. URL: https://memcached.org/.

[9] Redis, 2020. URL: https://redis.io/.

[10] Tofino Page - Barefoot Networks, 2020. URL: https://barefootnetworks.com/products/brief-tofino/.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012. doi:10.1145/2254756.2254766.

[12] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 49–65. USENIX Association, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[13] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS'06: The 2006 Ottawa Linux Symposium*, pages 71–86. Citeseer, 2006.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada*, pages 1–16. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Boyd-Wickizer.pdf.

[15] S. R. Chalamalasetti, K. T. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, pages 245–254. ACM, 2013. doi:10.1145/2435264.2435306.

[16] J. Choi, R. Lian, Z. Li, A. Canis, and J. H. Anderson. Accelerating Memcached on AWS Cloud FPGAs. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2018, Toronto, ON, Canada, June 20-22, 2018*, pages 2:1–2:8. ACM, 2018. doi:10.1145/3241793.3241795.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. doi:10.1145/1807128.1807152.

[18] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. *J. Parallel Distributed Comput.*, 72(11):1471–1480, 2012. doi:10.1016/j.jpdc.2012.01.020.

[19] D. Eastlake, T. Hansen, G. Fowler, K.-P. Vo, and L. Noll. The FNV Non-Cryptographic Hash Algorithm. 2019.

[20] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: an infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 345–362. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/eran.

[21] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the*

10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013, pages 371–384. USENIX Association, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[22] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[23] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1069–1084. ACM, 2019. doi:10.1145/3314221.3314590.

[24] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 135–148. USENIX Association, 2012. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[25] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, pages 54–66. ACM, 2018. doi:10.1145/3281411.3281443.

[26] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 489–502. USENIX Association, 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[27] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 121–136. ACM, 2017. doi:10.1145/3132747.3132764.

[28] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 4:1–4:14. ACM, 2014. doi:10.1145/2592798.2592821.

[29] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 137–152. ACM, 2017. doi:10.1145/3132747.3132756.

[30] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 31–44. USENIX Association, 2016. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-xiaozhou.

[31] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association, 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[32] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 36–47. ACM, 2013. doi:10.1145/2485922.2485926.

[33] J. Lozi, F. David, G. Thomas, J. L. Lawall, and G. Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 65–76. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

[34] I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 175–186. ACM, 2014. doi:10.1145/2619239.2626311.

[35] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 259–270. USENIX Association, 1993. URL:

https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[36] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20, 2018*, pages 1–8. IEEE, 2018. doi:10.1109/HPSR.2018.8850758.

[37] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019. doi:10.1145/3341301.3359641.

[38] 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 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[39] B. Pat, D. Dan, I. Martin, M. Nick, R. Jennifer, T. Dan, V. Amin, V. George, and W. David. Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44, 12 2013. doi:10.1145/2656877.2656890.

[40] G. Siracusano and R. Bifulco. Is it a SmartNIC or a Key-Value Store?: Both! In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 138–140. ACM, 2017. doi:10.1145/3123878.3132014.

[41] H. Takruri, I. Kettaneh, A. Alquraan, and S. Al-Kiswany. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 723–737. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/takruri.

[42] A. Wiggins and J. Langston. Enhancing the scalability of memcached. *Intel document, unpublished*, 2012.

[43] Y. Xu, E. Frachtenberg, and S. Jiang. Building a high-performance key-value cache as an energy-efficient appliance. *Perform. Evaluation*, 79:24–37, 2014. doi:10.1016/j.peva.2014.07.002.

[44] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 43–56. USENIX Association, 2016. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.