Fully Transparent Client-Side Caching for Key-Value Store Applications Using FPGAs

Sahan Bandara‡, Noah Cherry, and Martin Herbordt‡
ECE Department, Boston University ‡{sahanb, herbordt}@bu.edu

***Abstract*—Key-value stores (KVS) are a critical component of current data center infrastructure. They help address the extreme demand on data centers for high bandwidth, low latency access to large amounts of data. Due to their importance, many efforts have been made to improve their performance, which includes using FPGAs to offload some functionality. These efforts have been focused on improving the performance of the key-value store itself and reducing the load on the server running the KVS. However, with more FPGAs being deployed in the data centers by many cloud service providers, some use models that were not previously practical are becoming more realistic. In this work, we explore one such use case where we attempt to cache key-value entries at the network interface of the client server. We propose an FPGA design that is capable of caching the KVS data transparently to the KVS client application. The proposed solution is able to improve the application throughput while also reducing the network traffic generated by the KVS client. Also, as the proposed solution targets client servers that are typically shared by multiple clients, we discuss the importance of, and present our vision for, an FPGA design supporting multiple tenants.**

I. Introduction

The demand on data centers for high bandwidth access to large amounts of data is growing due to various factors such as the rise of cloud services, the Internet of Things (IoT), and the general evolution of the data center landscape. Large content providers such as Facebook, Google, Wikipedia, etc. dedicate significant compute resources to key-value databases. Key value databases cache frequently accessed content, reducing the load on back-end databases while providing low-latency and high-throughput access to content. Key-value stores (KVS) are used for tasks including storing user session data such as login information, storing configuration settings that applications need to access quickly, and numerous others.

Key-value stores, especially those geared towards caching small keys and values, pose challenges to data center servers due to the number of small network packets that must be processed. KVS clients generate a large number of requests requiring the KVS server to process all the requests and generate the response packets at a high rate to ensure high throughput. This makes KVS services susceptible to bottlenecks introduced by the networking stack of the operating system. Due to this, much prior work (such as [1], [2], [3], [4], [5]) has explored offloading key-value store functionality, fully or partially, to FPGAs using custom hardware designs. These designs tightly couple the KVS functionality with the packet processing, often with streaming architectures to provide higher throughput and lower latency compared to a standard server.

While many of the prior works have focused on using FPGAs to accelerate the KVS implementation itself, this work focuses on the client interface. We propose a lightweight FPGA design for caching the KVS entries on a network-facing FPGA attached to the client server. The FPGA implements a smaller key-value store and handles requests sent out to the remote KVS server/s. If an entry corresponding to a request is already in the local KVS, the FPGA responds to the request without sending out the packet to the network. If not, the FPGA behaves as a regular NIC and sends the packet out. When a response packet is received, the value is cached in the local KVS while forwarding the response packet to the host. This allows the FPGA to service the next request for the same entry. In this design, the FPGA acts as a secondary network interface on the client-server and the caching happens fully transparent to the KVS client application. Our FPGA design presents a Virtio [6] network device interface to the host machine. Virtio drivers are part of standard Linux distributions. Therefore, unlike most other FPGA-based designs there is no overhead of additional device drivers to communicate with the FPGA. From the point of view of the host operating system, the FPGA appears as a regular network interface. The only configuration step necessary is creating routing table entries to ensure that the packets intended for the remote KVS server are directed to the FPGA network interface instead of the primary NIC of the server.

We envision cloud applications that frequently access content from backend databases and use key-value stores as an intermediate caching layer benefiting from the proposed solution. For instance, consider a front-end web server running on a cloud server node. It needs to access the backend databases to fulfill requests from the users. In such a scenario, it is typical to have a key-value database such as Memcached [7] running on a different set of nodes to cache the frequently accessed content. The proposed solution can be deployed on the front-end server if it is equipped with an FPGA. This is becoming a realistic requirement as major cloud service providers have deployed FPGAs in their data centers [8], [9], [10], [11], [12], [13]. With caching at the network interface of the front-end server, the application should experience higher throughput due to a portion of requests being serviced without requesting data from the node running the KVS. Additionally, this will also reduce the network traffic on the data center network. In this work, we present our preliminary

implementation with a single network interface and a KVS on the FPGA. However, this work can be extended to implement multiple PCIe functions that could be assigned to different guest VMs on a server. The individual PCIe functions and the associated controllers on the FPGA could serve different applications and provide performance isolation and quality of service guarantees for different clients.

The contributions of this work are as follows:

- Propose caching key-value store data at the client-server network interface using network-facing FPGAs
- Present a lightweight FPGA design that acts as a network interface card and performs caching for the KVS transparently to the client application
- Propose a Virtio interface for the host-FPGA communication to reduce the overhead from custom device drivers for the FPGA
- Describe how the proposed solution could be extended to service multiple client applications running inside individual guest VMs
- Evaluate application performance in terms of throughput and network bandwidth utilization with and without caching KVS entries at the network interface

The remainder of this paper is organized as follows. We discuss background and related work in Section II. The proposed solution and the FPGA design are presented in Section III. In Section IV, we explore the improvement in application performance with the proposed solution. Finally, in Section V we discuss future research directions and conclude the paper.

II. Background and Related Work

A. Key-value Stores

Key-value stores (KVS) are non-relational databases that provide the functionality of an associative array by storing data as key-value pairs. KVS are suitable for storing a large variety of structured and unstructured data types. They can handle large amounts of data at high throughput and support quick read and write operations, often in constant time $O(1)$. With the ever-increasing demand on data centers for high bandwidth access to large quantities of data, distributed in-memory key-value stores such as Memcached[7] and Redis[14] have become integral middleware applications in the current datacenter infrastructure. Many web service providers use KVS [15], [16], [17], [18] and offer them to the users as a service [19], [20], [21], [22].

1) Memcached: Memcached [7] is a free and open-source, high-performance, in-memory key-value store. It is typically used as a distributed caching system for "small chunks of arbitrary data". Memcached provides a number of operators such as `Get`, `Set`, and `Delete` to manipulate the data objects.

B. Virtio

Virtio [6] is an industry standard for I/O virtualization. Guest operating systems running inside virtual machines use Virtio drivers to access virtual backend devices emulated by the host. Despite interacting with virtual devices, Virtio drivers use regular bus mechanisms for tasks such as device discovery, interrupts, and DMA. Therefore, it is possible to repurpose Virtio drivers as generic device drivers for FPGAs and any other device that implements a compliant interface. Authors of [23], and [24] have presented a practical design strategy for FPGAs to implement a Virtio-compliant interface.

Using Virtio drivers allows us to deploy an FPGA as a network device without designing and implementing new custom device drivers. Presenting the FPGA to the host OS as a network device allows the KVS caching implementation to be completely transparent to the host OS and the KVS client applications executing on the host.

C. Related Work

Due to the importance of key-value databases, a large number of prior works have made various attempts at improving their performance. Firstly, there are the software-oriented approaches focused on speeding up in-memory key-value stores on CPUs. There are algorithmic and data structure optimizations [25], parallel data access based methods [26], and novel methods for managing storage [27]. Kernel-bypass to overcome the limitations introduced by the network stack is also a common strategy [26], [27], [28]. Another set of work is focused on using RDMA to avoid the limitations introduced by the TCP/IP protocol stack [29], [30], [31]. Apart from academic work, research efforts have been made by large cloud operators as well [18].

A significant number of research efforts have been made to improve KVS performance using FPGAs. Many of these prior efforts offload the KVS completely or partially to FPGAs to achieve higher bandwidth and lower latencies [1], [2], [4], [3], [5]. Some FPGA-based works also focus on caching KVS entries using FPGAs [32]. There are also implementations that are geared towards specific applications such as Blockchains [33], [34]. However, all of these works above focus on the KVS node instead of the client nodes. The authors of [35] discuss providing multi-tenant services with FPGAs where all the tenants are using the same service.

III. Method

A. Scope for the Proposed Solution

We propose an FPGA design to perform caching at the network interface for key-value store applications. This solution targets servers running client applications that need to access data from backend servers and rely on key-value databases that run on dedicated nodes to cache frequently requested data and reduce the load on the backend servers. The target server should be equipped with a network-facing FPGA. The FPGA is able to act as a secondary network interface card for the server. Figure 1 depicts the general deployment context targeted by the proposed solution. Here the frontend server (server (1)) is running some application that serves the requests from the users. The data required to service the user requests are stored in the database on server (2). Server (3) runs a key-value store that caches the frequently accessed data in order to reduce the load on the server (2).

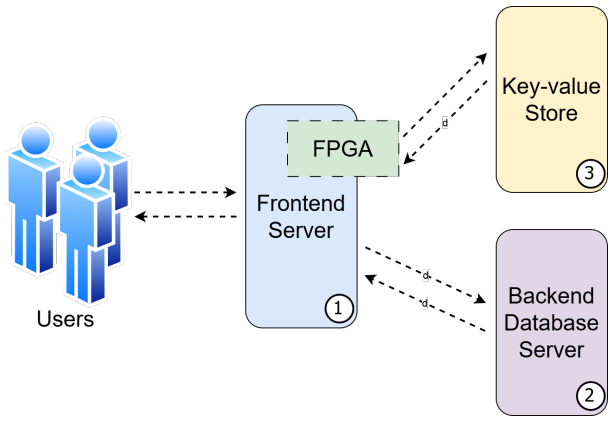

Fig. 1. Target deployment architecture (Adapted from [1])

With our proposed solution, all packets between servers (1) and (3) are sent through the network-facing FPGA on server (1). The FPGA implements a smaller key-value store and the network interface functionality. The architecture of the FPGA design is described in Section III-B. The proposed FPGA design includes a host interface compliant with the Virtio specification [6]. This allows the host OS to use in-kernel Virtio device drivers to communicate with the FPGA. Because our concrete implementation is a Virtio network device, the FPGA appears as a regular network interface card for the host OS. Because our current implementation does not implement any additional functionality such as checksum calculations, TCP segmentation offload, etc., it relies on the host operating system's network stack and the device driver to provide fully formed Ethernet frames to the FPGA. However, because Virtio specification supports feature negotiation between the device and the driver, future implementations can implement additional network stack functionality and enable offloading more work from the CPU to the FPGA. Any such additions do not require any changes to the current use model where the KVS client is oblivious to the existence of the FPGA or the caching at the network interface.

Our current design implements a KVS compliant with the Memcached binary protocol [36]. It should be noted that the key idea of client-side caching of KVS values is not unique to a particular protocol. We have opted to implement a simple protocol to demonstrate the feasibility of the proposed approach because the proposed solution/optimization is orthogonal to the actual key-value store implementation. This also means that many of the optimizations focused on the KVS itself proposed in prior work are also applicable to this solution and could be included in future implementations.

After a packet is copied to the FPGA memory from the host memory using DMA, the KVS module reads the relevant fields from the packet to identify the type of request and the key. At present, our implementation only supports `Get` and `Set` requests in the Memcached protocol. A `Get` request triggers a KVS lookup. If the requested key is available in the local KVS, the FPGA generates a response ethernet frame and moves it to the appropriate buffer in the host memory. If the key is not found, the request packet is sent out into the network using the FPGA's network interface. When the response packet is received, the KVS module inserts the value into its memory while the DMA controller moves the packet to the host memory. The FPGA's behavior for a `Set` request is configurable. The two modes supported are: (i) `Set` requests trigger a KVS lookup; entry is updated for a hit; the request packet is sent out to the KVS server (ii) `Set` requests are never cached; If the corresponding entry is in the local cache, it is invalidated; Packet is sent to the remote KVS server. The behavior should be selected based on the kind of data cached and the consistency model enforced by the remote key-value store. For all other request types, the local KVS is bypassed and the packet is sent to the network interface. Any local copies of the entry targeted by the unsupported request are invalidated. Essentially, the FPGA behaves as if there is no local cache in place for all unsupported requests.

B. Architecture

Figure 2 provides an overview of the FPGA design. The major components of the design are described here.

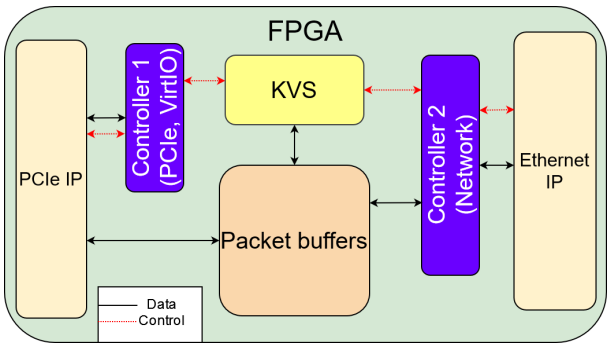

Fig. 2. Overview of the FPGA Design

1) Key-value Store: The key-value store implementation is compliant with the Memcached binary protocol [36]. A simple CRC16 hash is used as the hash function for the KVS. This allows us to implement a KVS with up to 2^{16} entries. Keys and values are stored in FPGA block RAMs (BRAM) to simplify the design. However, the design could be modified to use off-chip DRAM to store the values and BRAMs to store keys and pointers to the values in DRAM. The KVS is implemented as a direct mapped cache. Therefore, no replacement policy is implemented. A single FSM handles the orchestration of all computations, moving data to and from packet buffers, reading/writing keys and values from/to BRAMs, and any control communication with the two controllers that control the PCIe and Ethernet interfaces. The KVS module acts as an intermediary between the PCIe and Ethernet portions of the design because it should be able to prevent packets from being transmitted to the network if it is possible to service a request with data cached locally.

2) PCIe and Ethernet IPs: We have implemented our design on a Xilinx Ultrascale+ FPGA. Therefore, XDMA [37]

and CMAC [38] IP cores provided by Xilinx are used to implement the PCIe and Ethernet interfaces respectively. The CMAC IP is used without any modifications. However, the RTL for the XDMA IP is slightly modified to support the additional functionality necessary to implement a Virtio-compliant interface on the FPGA.

3) PCIe + Virtio Controller: The FPGA design follows a modular approach where independent controllers control different interfaces. The module responsible for controlling the PCIe IP core also implements the data structures and the FSMs required to implement the Virtio functionality. The Virtio interface implementation is described further in Section III-C. This controller informs the KVS module when new packets are moved to a packet buffer. Similarly, the KVS module signals the PCIe/Virtio controller when there are packets to be moved to host memory.

4) Network Interface Controller: This module is responsible for controlling the interface with the network. This entails controlling the Ethernet IP core, moving packets ready to be transmitted from packet buffers to the IP core's input ports, and moving the packets received on the output ports of the IP core to packet buffers. The KVS module signals this controller when packets in the buffers are ready to be transmitted. Similarly, this module signals the KVS after packets received over the network are moved to packet buffers.

5) Packet Buffers: The packet buffers are implemented using FPGA BRAMs. Since each buffer is accessed by more than one controller, arbitration logic is implemented to grant access to the buffers. The sizes of the buffers are configurable parameters. In-memory data structures are used to exchange information such as packet sizes, starting addresses, etc. among different controllers. This simplifies the signaling between the modules. For instance, when the PCIe controller signals the KVS module to indicate that new packets are available, the KVS module reads a data structure placed alongside the packet buffers in memory to find out the starting addresses and sizes of the new packets.

C. Virtio Interface

One of the novel aspects of this work is using a Virtio interface for host-FPGA PCIe communication. This removes the overhead of designing and implementing custom device drivers for the FPGA developer and the overhead of setting up and maintaining the drivers for a system administrator. Since the FPGA appears as a regular network interface card, no changes are necessary for the KVS client application. The implementation is based on the design presented in [23], and [24]. The PCIe+Virtio controller module implements several data structures that are used by the Virtio device drivers for device initialization and regular operations such as sending notifications to the device to trigger data movement between the host and the FPGA. It also includes the FSMs that implement the queue functionality used by Virtio drivers and devices to communicate.

Virtio drivers are designed to be used by guest operating systems executing in virtualized environments to access virtual devices emulated by the host. Therefore, the drivers are not designed to manage DMA engines on any particular device. This means that the PCIe controller module is also responsible for programming the DMA engine which is part of the PCIe IP core. This requires the target device to have an IP core (provided by the vendor, third party, or developed by a user) that allows the DMA engine to be programmed by logic implemented on the FPGA. If not, a device driver that presents the FPGA as a network device to the host OS is necessary to implement the proposed solution.

D. Extension to Multiple client Applications/VMs

Since Virtio drivers are intended for guest operating systems and the drivers do not make any distinction between virtual and physical devices as long as the correct interface is implemented, it is possible for guest VMs to also access the FPGA without additional device drivers. We have tested our current implementation with a single PCIe function and a single VM by using PCIe passthrough to assign the FPGA to the VM. We expect to extend this approach to multiple VMs using multiple PCIe functions and the Single Root I/O Virtualization (SR-IOV) technique. This suits the target deployments we have considered for this solution, which are cloud nodes running applications that are key-value store clients. Typically such applications run inside virtual machines and share the server resources with other applications.

If there are multiple KVS client programs executing inside different VMs on the same host machine, it is important to provide each of them with a simple but secure mechanism to access the FPGA. If the different clients are accessing different key-value databases, the corresponding PCIe functions implemented on the FPGA could have their own KVS controllers and memory. Even if the different clients are accessing the same database, each could be assigned their own cache on the FPGA in order to provide additional security, performance isolation, and quality of service guarantees.

Figure 3 depicts our vision for a multi-tenant deployment where multiple KVS client programs are running inside individual VMs on the same server. The PCIe interface of the FPGA presents multiple PCIe functions (physical or virtual) to the host OS, which are assigned to individual VMs. VMs 2, 3, and 4 are running KVS clients and therefore are connected to the FPGA. VM 1 could be running a non-KVS program and therefore is not connected. Within the FPGA, functions 3, and 4 share the same KVS controller. However, function 2, which is connected to VM 2 uses its own KVS controller. This could be due to VM 2 accessing a different KVS from the other two VMs or due to security or performance guarantees required by the program running in VM 2. The KVS controllers implemented on the FPGA share the external interfaces such as PCIe and Ethernet.

IV. Evaluation

The purpose of allocating compute resources to run key-value stores is to provide fast access to data frequently accessed and reduce the load on backend databases. However,

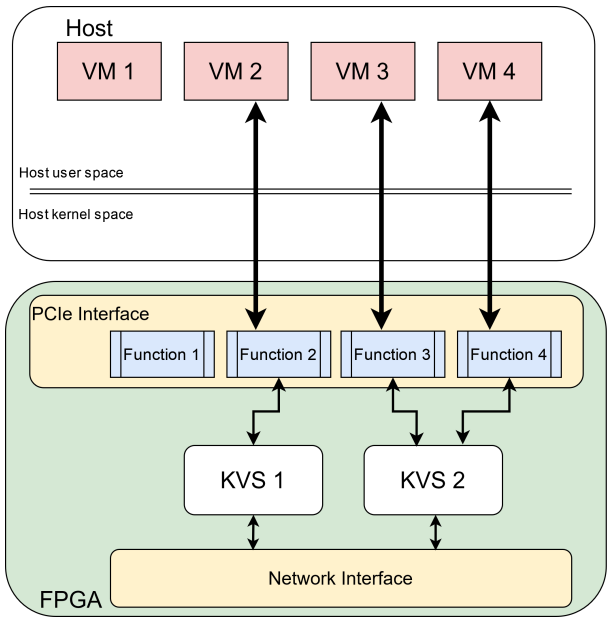

Fig. 3. Example Multi-tenant Deployment

since traditional servers a prone to performance bottlenecks when running key-value stores, many FPGA-based solutions have been proposed to improve the throughput and latency of the key-value stores. These approaches are typically evaluated based on the improvement in application throughput. We also evaluate our proposed solution based on application throughput.

A. Experimental Setup

We have implemented our design on a Bittware CVP13 [39] development board that uses a Xilinx Ultrascale+ FPGA (part number: xcvu13p-figd2104-2-e). The machine hosting the FPGA and running the KVS client program and the server running the key-value store program are both connected to the same 1 Gbps local network. We consider the KVS client performance without the FPGA-based solution as the baseline for our analysis. For time measurements, the test applications use the `clock_gettime()` function with the `CLOCK_MONOTONIC` option.

Since the real KVS workloads are dominated by read requests [40], we focus our analysis on the performance improvement for Memcached `Get` requests. Higher hit rates on the local KVS cache correspond to better client application performance since fetching a value from the FPGA takes considerably less time compared to fetching the same value from a remote server running the KVS. We assume that the KVS client application's performance is only limited by accessing the key-value store on a remote node.

B. Results

In this section, we model the performance improvement achieved by the proposed solution. The model is based on two types of latency measurements made using the setup described in Section IV-A.

1) Time taken for the client program to issue a `Get` request to the key-value store running on a different machine on the same network, and receive the response with the value. This is considered to be the baseline performance when there is no FPGA KVS cache implementation in place.
2) Time for the client program to receive the response when the KVS entry requested is cached in the FPGA.

Figure 4 shows the effective read throughput in for the client program for different sizes for the value field ranging from 16 Bytes to 512 Bytes versus different cache hit rates for the KVS cache in the FPGA. Performance for the 0% hit rate is the same as the performance without the FPGA cache in place. The highest performance improvement is seen for the smallest values. This can be explained by the higher overhead of network packets for smaller values. For the smallest sizes, the overhead of the protocol headers alone is closer to the size of the actual payload. The performance benefits decrease with the size of the value. However, the negative slope of the (light blue) line representing 512 Byte values cannot be explained only by the decreasing overhead of the network packets compared to smaller value sizes. The other factor contributing to the poor performance is that our FPGA design is still not optimized to handle larger values efficiently. With an optimized design that can handle larger values better, the proposed solution should result in improved performance for a wider range of payload sizes.

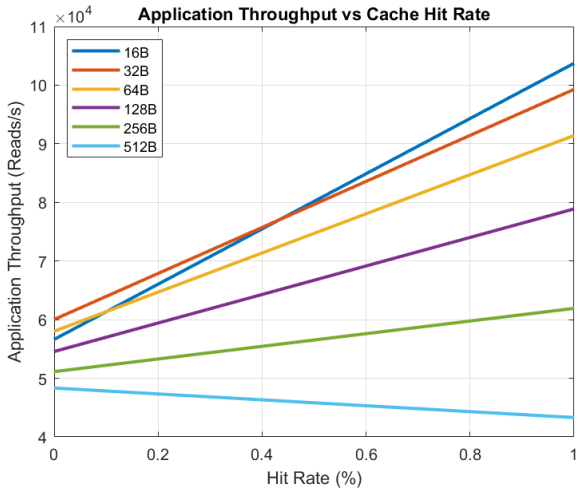

Fig. 4. Application Throughput versus Cache Hit Rate

Figure 5 shows the network bandwidth versus the cache hit rates. Please note that this is the network bandwidth perceived by the client application and not the actual bandwidth utilized. Because the application is unaware of the caching at the network interface, from the point of view of the application, it is experiencing higher network bandwidth utilization. For the no-cache scenario (0% hit rate), the application only reaches around 7% of the available network bandwidth of 1 Gbps when fetching 16 Byte values from the KVS. However, with a 100%

hit rate, it could go up to around 13% which is almost double the bandwidth for the no-cache case. The hidden benefit of this is that the actual number of packets transmitted over the network is reduced lowering the actual network traffic. The reason for the poor performance of the 512 Byte value scenario is the same as explained above.

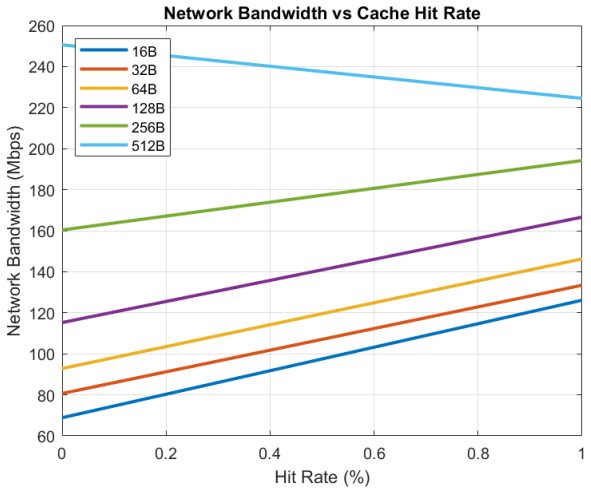

Fig. 5. Network Bandwidth versus Cache Hit Rat

V. Conclusion

In this work, we explore a novel use case for FPGAs enabled by more FPGAs being deployed in the data center: caching key-value store data in the network interface of the client server. We have presented a lightweight FPGA design and presented our analysis based on preliminary results. We demonstrate that the proposed method can improve application throughput and reduce the network traffic generated by KVS client applications.

One future research direction is optimizing the FPGA implementations of the KVS by applying optimizations from prior work. Another is implementing support for larger caches by using off-chip DRAM to store values and using FPGA BRAMs only to store keys and pointers to the values in DRAM. We are also working on implementing multi-tenant support as described in Section III-D.

Acknowledgments

This work was supported, in part, by Red Hat through award 2024-01-RH08.

References

[1] J. Choi, R. Lian, Z. Li, A. Canis, and J. Anderson, "Accelerating Memcached on AWS Cloud FPGAs," in *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2018, pp. 1–8.

[2] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached Appliance," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2013, pp. 245–254.

[3] M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-based In-line Accelerator for Memcached," *IEEE Computer architecture letters*, vol. 13, no. 2, pp. 57–60, 2013.

[4] W. Liang, W. Yin, P. Kang, and L. Wang, "Memory Efficient and High Performance Key-value Store on FPGA Using Cuckoo Hashing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[5] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with {FPGAs}," in *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*, 2013.

[6] M. S. Tsirkin and C. Huck, "Virtual I/O device (VIRTIO) version 1.2," May 2022. [Online]. Available: https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html

[7] "Memcached - A Distributed Memory Object Caching System." [Online]. Available: https://memcached.org/

[8] "Project Catapult." [Online]. Available: https://www.microsoft.com/en-us/research/project/project-catapult/

[9] A.M. Caulfield, et al., "A cloud-scale acceleration architecture," in *MICRO*, 2016.

[10] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[11] Amazon.com Inc., "Amazon EC2 F1 Instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[12] "Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances." [Online]. Available: https://alibaba-cloud.medium.com/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances-74b9aeac98ed

[13] "Cloud FPGA Accelerators: Turbocharge Your Workflows for Efficient Processing." [Online]. Available: https://www.nimbix.net/fpga-compute/

[14] "Redis - The Real-time Data Platform." [Online]. Available: https://redis.io/

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[17] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 18, no. 2, pp. 41–49, 2013.

[18] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[19] "Amazon ElastiCache." [Online]. Available: https://aws.amazon.com/elasticache/

[20] "Azure Cache for Redis." [Online]. Available: https://azure.microsoft.com/en-ca/services/cache

[21] "ApsaraDB for Memcache." [Online]. Available: https://www.alibabacloud.com/product/apsaradb-for-memcache

[22] "Google Cloud Memorystore." [Online]. Available: https://cloud.google.com/memorystore

[23] S. Bandara, A. Sanaullah, Z. Tahir, U. Drepper, and M. Herbordt, "Enabling VirtIO Driver Support on FPGAs," in *8th International Workshop on Heterogeneous High Performance Reconfigurable Computing*, 2022, doi: 10.1109/H2RC56700.2022.00006.

[24] ——, "Performance Evaluation of VirtIO Device Drivers for Host-FPGA PCIe Communication," in *31st Reconfigurable Architectures Workshop (RAW)*, 2024, doi: 10.1109/IPDPSW63119.2024.00043.

[25] B. Fan, D. G. Andersen, and M. Kaminsky, "{MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 371–384.

[26] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "{MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 429–444.

[27] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum *et al.*, "The ramcloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–55, 2015.

[28] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "{BMC}: Accelerating memcached using safe in-kernel caching and pre-stack

processing," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 487–501.

[29] W. Tang, Y. Lu, N. Xiao, F. Liu, and Z. Chen, "Accelerating redis with rdma over infiniband," in *Data Mining and Big Data: Second International Conference, DMBD 2017, Fukuoka, Japan, July 27–August 1, 2017, Proceedings 2*. Springer, 2017, pp. 472–483.

[30] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance rdma capable interconnects," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 743–752.

[31] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[32] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura, "Caching memcached at reconfigurable network interface," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–6.

[33] Y. Sakakibara, K. Nakamura, and H. Matsutani, "An fpga nic based hardware caching for blockchain," in *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2017, pp. 1–6.

[34] A. I. Sanka, M. H. Chowdhury, and R. C. Cheung, "Efficient high-performance fpga-redis hybrid nosql caching system for blockchain scalability," *Computer Communications*, vol. 169, pp. 81–91, 2021.

[35] Z. István, G. Alonso, and A. Singla, "Providing multi-tenant services with fpgas: Case study on a key-value store," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 119–1195.

[36] Memcached, "Binaryprotocolrevamped." [Online]. Available: https://github.com/memcached/memcached/wiki/BinaryProtocolRevamped

[37] Xilinx, "DMA/Bridge Subsystem for PCI Express v4.1," Jun 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/pg195-pcie-dma

[38] ——, "UltraScale+ Devices Integrated 100G Ethernet Subsystem Product Guide," Nov 2023. [Online]. Available: https://docs.amd.com/r/en-US/pg203-cmac-usplus/UltraScale-Devices-Integrated-100G-Ethernet-Subsystem-v3.1-LogiCORE-IP-Product-Guide

[39] "CVP-13 FPGA Cryptocurrency Mining Board." [Online]. Available: https://www.bittware.com/cvp-13/

[40] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," vol. 40, no. 1, 2012. [Online]. Available: https://doi.org/10.1145/2318857.2254766