Accelerating Memcached on AWS Cloud FPGAs

Jongsok Choi, Ruolong Lian, Zhi Li, Andrew Canis, and Jason Anderson
LegUp Computing Inc.
88 College St., Toronto, ON, Canada, M5G 1L4
info@legupcomputing.com

ABSTRACT

In recent years, FPGAs have been deployed in data centres of major cloud service providers, such as Microsoft [1], Amazon [2], Alibaba [3], Tencent [4], Huawei [5], and Nimbix [6]. This marks the beginning of bringing FPGA computing to the masses, as being in the cloud, one can access an FPGA from anywhere.

A wide range of applications are run in the cloud, including web servers and databases among many others. Memcached is a high-performance in-memory object caching system, which acts as a caching layer between web servers and databases. It is used by many companies, including Flicker, Wikipedia, Wordpress, and Facebook [7, 8].

In this paper, we present a Memcached accelerator implemented on the AWS FPGA cloud (F1 instance). Compared to AWS ElastiCache, an AWS-managed CPU Memcached service, our Memcached accelerator provides up to 9× better throughput and latency. A live demo of the Memcached accelerator running on F1 can be accessed on our website [9].

1. INTRODUCTION

FPGAs have evolved considerably over the years, from small chips used for glue logic to mega-sized chips housing 30 billion transistors [10] used to implement complex systems. Recently, FPGAs have been deployed in data centres. In 2014, Microsoft pioneered this by speeding up Bing Search with FPGAs [11]. For Microsoft, FPGAs are now used in production in their data centres to speed up the cloud network, Bing Search, and Azure Machine Learning [1]. In the last two years, Amazon Web Services (AWS), Alibaba, Tencent, Huawei, and Nimbix have also rolled out FPGAs in their public clouds. This is a game changer for the accessibility of FPGAs, as FPGAs have traditionally been expensive and difficult to buy and even more difficult to set up. Being in the cloud, anyone around the world can use an FPGA from their own location, without the obstacles of hardware acquisition and installation. Cloud service providers (CSPs) typically charge FPGA usage per hour, where for AWS, the hourly rate for an FPGA is comparable to GPUs and high-end CPUs [12]. Consequently, users no longer have to incur significant up-front capital expenses to use FPGAs. We believe that this is a major step towards making FPGAs mainstream computing devices – FPGAs can be as accessible as CPUs and GPUs.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2018) Toronto, Canada, June 20-22, 2018.

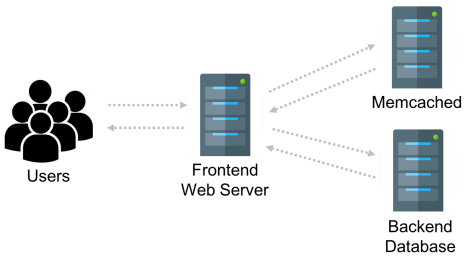

Figure 1: Memcached deployment architecture.

Another historic impediment to FPGA accessibility lies in their programmability or ease-of-use. FPGAs have traditionally been difficult to use, requiring expert hardware designers to use low-level RTL code to design circuits. In order for FPGAs to become mainstream computing devices, they must be as easy to use as CPUs and GPUs. There has been ample work to make FPGAs software programmable. Both Intel and Xilinx have invested heavily in building their high-level synthesis (HLS) tools, the Intel FPGA SDK for OpenCL [13] and Xilinx Vivado HLS [14]. LegUp Computing also commercially released LegUp HLS in 2016 [15], based on years of research and development at the University of Toronto [16]. One of the unique features of LegUp is that it can target FPGAs from any FPGA vendor, including Intel, Xilinx, Microsemi, Lattice Semiconductor, and Achronix. While different HLS tools may vary in terms of supported features and FPGAs, they share the same overarching goal – to make FPGAs easier to use.

1.1 Memcached

Web servers and databases are common data-centre workloads. Many CSPs, including AWS, Microsoft, and Alibaba, provide their own managed services for these applications [17, 18, 19]. Memcached is another commonly used application that is used to decrease the latency in serving web data and also to reduce the load on databases. AWS also provides its own managed Memcached service, called ElastiCache [20]. Its typical deployment is shown in Fig. 1.

Memcached stores key-value pairs, where for a given key, it can *set* a value to store it in the cache, and can also *get* the stored value for a given key. The *set* and *get* commands are the fundamental Memcached commands, out of its handful number of total commands [21]. In a typical web deployment, when a user accesses a website, the frontend webserver accesses Memcached for some of the needed data. If the data is present in Memcached (i.e., Memcached

hit), it will be rapidly returned to the user, avoiding trips to the backend database, which is slow due to the use of disk storage. If the required data is not found in Memcached (i.e., Memcached miss), the backend database needs to be accessed. Memcached stores all the key-value pairs in non-persistent memory (RAM), allowing low-latency data accesses, however, in the event of failure (i.e. machine failure, power shutdown), all stored data is lost.

1.2 Our Work

We present a working prototype of an FPGA-accelerated Memcached deployed on the AWS FPGA cloud (F1 instance). The objective of our work is to showcase a real-world application running on the AWS F1 instance, illustrating the benefits of having FPGAs in the cloud, and also to analyze the AWS F1 architecture in terms of its network performance. To build the Memcached accelerator, we did not want to re-invent the wheel by implementing Memcached and a TCP/IP network stack from scratch, as there exists prior open-source work [22, 23, 24]. Instead, we built upon the open-source work to create a working system, which took three engineers roughly two months. Some of this time was spent understanding the Memcached application, the prior work on FPGA-implemented Memcached and the TCP/IP network stack, and the AWS F1 architecture. A large portion of the time was also spent on implementing the Virtual-Network-to-FPGA framework (described in Section 4.1) to provide network access to the FPGA, as the FPGAs on F1 instances are not directly connected to the network (c.f. Section 3). We also implemented support for Memcached pipelining (batching described in 4.3.1) to improve the throughput. Putting together all of the components to create a high-performance system on the cloud infrastructure was also a substantial task.The focus of this paper lies in how we implemented the Memcached accelerator on AWS F1 and some of the interesting things that we learned about F1 along the way.

Our contributions are:

1. An FPGA-accelerated Memcached server deployed on AWS F1. To the best of our knowledge, this is the first of its kind deployed on a public FPGA cloud.
2. An analysis of the AWS F1 architecture and its PCIe and network bandwidth. Although not officially disclosed, we learned that AWS throttles the Packets-Per-Second (PPS) on the network of their instances, directly impacting network performance.
3. The Virtual-Network-to-FPGA (VN2F) framework which provides network access to the FPGA on F1.
4. A quantitative analysis of our Memcached accelerator compared to AWS ElastiCache in terms of throughput, latency, and cost-efficiency.

2. RELATED WORK

Many CSPs provide their own CPU caching solutions: ElastiCache by AWS [20], Redis Cache by Microsoft Azure [25], ApsaraDB for Memcache by Alibaba Cloud [26], and Cloud Memorystore by Google Cloud [27], just to list a few. Typically, users can configure the specs of the cache instance according to their needs. For ElastiCache, AWS offers different instance types based on their RAM size, network bandwidth, and hourly cost. Redis [28] is another in-memory data storing solution similar to Memcached, but supports more complex data types such as lists, sets, and hashes, and offers advanced features such as replication and clustering, which are not available in standard Memcached.

On the research front, there has been ample work in speeding up in-memory key-value stores (KVS) on CPUs. MICA [29] accelerated KVS by using parallel data accesses and kernel bypass and by re-designing the data structures. MemC3 [30] also sped up Memcached by 3× by making algorithmic and data structure-based improvements. RAMCloud [31] employed a novel mechanism for managing storage and also used kernel bypass. FaRM [32] used RDMA (Remote Data Memory Access) to improve both throughput and latency.

A number of prior works centre on FPGA implementations of KVS applications. [22] accelerated Memcached on an on-premises FPGA, where they achieved line-rate Memcached processing for a 10 Gbps network. KV-Direct [33] used a programmable NIC to extend RDMA capabilities and minimize the PCIe latency, which allowed them to achieve performance close to the physical limits of the underlying hardware. [34] implemented Memcached on a Xilinx Zynq SoC FPGA to offload network processing and data look-up to hardware.

While some prior work focused on creating a custom KVS application to improve performance, we opted for standard Memcached, since for an FPGA application to be widely used, we think it is important to accelerate one that is already well known and used. In terms of FPGA implementations, we believe this to be the first deployment of a Memcached accelerator on a public FPGA cloud. As opposed to on-premises hardware, a public cloud has specific properties that can affect performance, which we discuss in the subsequent sections of this paper.

3. AWS CLOUD-DEPLOYED FPGAS (F1)

AWS publicly released the F1 platform for general availability in 2017. There are two types of the F1 instance, the `f1.2xlarge`, comprising a single Ultrascale+ FPGA and 8 Intel Xeon E5 vCPUs, and the `f1.16xlarge`, which has eight Ultrascale+ FPGAs and 64 Xeon E5 vCPUs [2]. Each FPGA has 64 GBs of DDR4 memory. We use the `f1.2xlarge` instance for this work. There are now 20 ready-made applications (Amazon Machine Instances – AMIs) which can be deployed on the F1, with applications ranging from machine learning, genomics to data analytics. Xilinx and AWS are actively working towards making the platform easier to use, and adding support for tools such as IP Integrator and SDAccel to raise the abstraction of FPGA programming.

AWS also provides a *shell* for the F1 FPGA, which has the major interfaces (e.g,. PCIe, DDR) one may need for their accelerator to communicate with the CPU and/or memory [35]. In Xilinx IP Integrator, the *shell* is presented as an IP core that can be configured and connected to user cores. When using SDAccel, the tool automatically synthesizes the kernel and connects it to necessary interfaces, raising the design abstraction even higher. Since we wanted fine-grained control of the hardware, we used the IP Integrator to manually connect our HLS and RTL cores together to the F1 shell.

Unfortunately, in the F1 architecture, the FPGA is not directly connected to the network, which is not ideal for networking applications. FPGAs have traditionally excelled at networking, as FPGAs can process extremely high band-

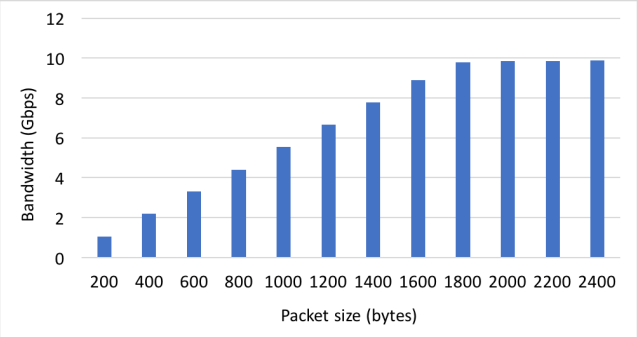

Figure 2: Network bandwidth on the AWS F1 instance.

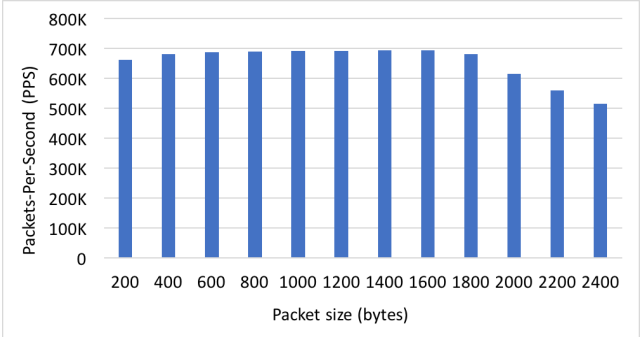

Figure 3: Packets-per-second on the AWS F1 instace.

width data at line-rate – a task that cannot be handled by CPUs. On the AWS F1, the CPU is connected to the network and the FPGA is connected to the CPU over PCIe. Purportedly, this is for security reasons. Microsoft, on the other hand, has their FPGAs connected directly to the network and reports to have the fastest cloud network [1]. However, the FPGAs in the Microsoft cloud are not publicly accessible.

3.1 Network Throughput on F1

Network bandwidth is an important consideration as it sets the upper bound in performance for networking applications. In the cloud, the CSPs control the network bandwidth and the maximum network bandwidth on AWS varies by the selected instance type. The `f1.2xlarge` instance is stated to have *Up to 10 Gigabit* network bandwidth. We ran an experiment to assess this limitation. To measure network bandwidth of the F1, we used two F1 instances in the same availability zone (i.e., region) [36] and ran `iperf` [37], a commonly used network bandwidth measuring tool, to send packets from one instance to the other. Fig. 2 shows the measured network bandwidth for different packet sizes. When packets are 200 bytes, the total network bandwidth is only about 1 Gbps. With larger packet sizes, the bandwidth continues to increase, until it plateaus at around the 1800-byte packet size (9.8 Gbps at 1800 bytes, and 9.9 Gbps at 2400 bytes). When sending 8989-byte-sized packets (jumbo packets), we were able to reach 10.1 Gbps. It is apparent that the 10 Gbps network cannot be saturated with small packets. This is because AWS imposes a packets-per-second (PPS) limitation on its network [38].

In the cloud, hardware is virtualized and shared among multiple users. Hence, a 10 Gbps network link does not have to be dedicated to a single machine instance, but can be shared across multiple instances. The PPS throttling allows the network bandwidth to be divided among many users. AWS does not officially disclose PPS limits for different instance types, and even AWS support did not disclose this information when we inquired. However, the PPS can be calculated from the results shown in Fig. 2, by dividing the measured bandwidth by the packet size (i.e., Gbps×1e9/8/packet size in bytes). Fig. 3 shows the PPS for different packet sizes. For smaller packet sizes below 1800 bytes, the maximum PPS is around 700K, due to AWS throttling. For 1800-byte packets and above, the PPS starts to drop, and this is because with bigger packets, we saturate the 10 Gbps total bandwidth, hence bandwidth becomes the bottleneck. As a sanity check for the PPS results, we used another methodology described in [39], which also gave us the same 700K maximum PPS (when receiving and transmitting concurrently, the PPS is 700K each away). We then confirmed this finding with AWS engineers, who verified that the PPS is what they would expect for the `f1.2xlarge` instance.

As a reference, the theoretical maximum PPS can be calculated by dividing the maximum network bandwidth (10 Gbps) by the transferred packet size. With 200-byte packets, the theoretical maximum PPS is 6.25 million, 9× the PPS on F1. The theoretical maximum PPS and the F1's PPS starts to align for larger packet sizes when the network bandwidth becomes the bottleneck and PPS throttling is no longer applicable (at 1800-byte packets, max PPS = 694K, measured PPS on F1 = 681K). Hence for small packet sizes, PPS throttling is quite significant. Other type of instances, such as the `f1.16xlarge`, have higher network bandwidth and may have higher PPS limits, but the `f1.16xlarge` instance costs $13.2/hour, 8× the price of an `f1.2xlarge`, so from a cost-effectiveness perspective, we think the use cases for the `f1.16xlarge` are quite limited.

We found the 700K maximum PPS to be a significant limitation of the F1 instance, as other CPU instances have over a million PPS limit [38], and for networking applications, the PPS can directly impact the performance (i.e., the PPS can become the upper-bound for performance, leaving the FPGA under-utilized). The PPS limitation does not exist for on-premises FPGA data centres, where one has complete control of the network. We plan to implement our Memcached accelerator on a discrete Intel Programmable Acceleration Card (PAC), with an Arria 10 FPGA [40], where we can fully utilize the networking-processing capacity of the FPGA.

4. SYSTEM ARCHITECTURE

The high-level system architecture is shown in Fig. 4. There are three major components of the Memcached server accelerator system on F1: 1) The Virtual-Network-to-FPGA (VN2F), 2) the TCP/IP and UDP offload engine on FPGA, and 3) the Memcached server accelerator on FPGA. All components need to be tightly integrated and optimized for high throughput to create a high-performance Memcached server. Since network processing is a core strength of FPGAs, where modern FPGAs are capable of handling 100 Gbps network and above [41] at line-rate, we want to offload the network processing to the FPGA. However, since

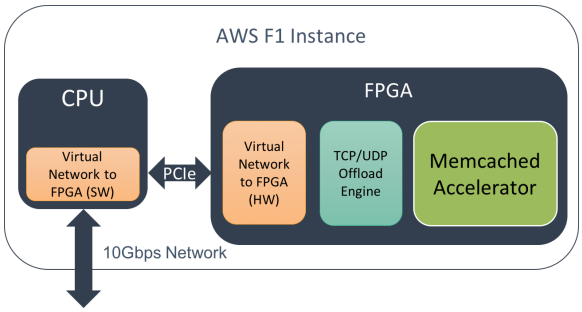

Figure 4: System architecture on F1.

the network is not directly connected to the FPGA on F1, we created the VN2F framework, which is a combination of both software and hardware components (described in 4.1), to bypass the OS kernel on the F1 CPU so that it transfers raw Ethernet packets from the NIC (Network Interface Card) over PCIe to the network stack on the FPGA (described in Section 4.2). The FPGA network stack connects to the FPGA Memcached server (described in Section 4.3) for Memcached processing.

The high-level operation of the F1 Memcached server accelerator system is as follows: 1) A client sends network packets containing Memcached commands to the IP address of the F1 instance. 2) The software component of the VN2F Framework receives network packets from the NIC and DMAs them over PCIe to the hardware component of VN2F framework on the FPGA (which includes on-chip buffers). 3) The FPGA network stack reads from the on-chip buffer to process the network packets. It decodes the packets, drops any invalid ones, and sends Memcached commands and data to the Memcached core. 4) The Memcached core performs the caching operations according to the received commands. For our Memcached prototype, we currently support the `set` and `get` commands, used to store-to and retrieve-from the cache. For each command, Memcached forms a response: For a `set`, the response indicates whether the set operation succeeded; For a `get`, the response includes the corresponding value of the key if it exists in the cache, or otherwise indicates a cache miss. 5) The response is sent back to the FPGA network stack, which packetizes the response with the appropriate TCP/IP/Ethernet frame headers. The network stack stores the network packets in another on-chip buffer of the VN2F hardware. 6) The VN2F software on the CPU reads from the on-chip buffer and writes to the NIC, which sends the network packets back to the client.

Note that the FPGA hardware (the network stack, and the Memcached core) is fully pipelined so that it processes multiple network packets and Memcached commands in flight. Our VN2F framework also concurrently transfers NIC-to-FPGA and FPGA-to-NIC continuously to minimize transfer latency.

4.1 Virtual-Network-to-FPGA (VN2F)

The architecture of the VN2F framework is shown in Fig. 5. The software component's task is to transfer network packets from NIC to FPGA and from FPGA to NIC. Our kernel bypass application is built on top of DPDK (Data Plane Development Kit) and F1 drivers. The DPDK is a set of libraries and drivers used for fast packet processing [42].

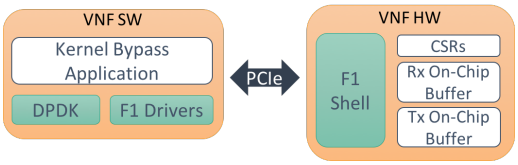

Figure 5: The Virtual-Network-to-FPGA (VN2F) framework.

It runs in the user space, allowing one to bypass the software network stack running in the kernel space. We use DPDK to receive/transmit network packets from/to NIC. The F1 drivers include AWS-provided drivers to set up the F1 FPGA, as well as DMA drivers used to transfer data to/from FPGA over PCIe. On the hardware side of VN2F, we have the AWS-provided F1 shell, which provides the hardware connectivity to the FPGA over PCIe. Connected to the F1 shell are CSRs (Control and Status Registers), and Rx (receive) and Tx (transmit) on-chip buffers. CSRs are used to configure the FPGA hardware (e.g., the MAC and IP addresses assigned to the NIC) and to check the status of the on-chip buffers. Rx/Tx buffer are used to buffer incoming and outgoing network packets.

Before implementing the VN2F framework, we first wanted to assess the achievable PCIe bandwidth on F1 using the AWS F1 shell, as this is a crucial factor for performance. We used the `cl_dram_dma` example provided by AWS, which allows data transfers between the host CPU and FPGA's DRAM [43]. The default PCIe DMA driver on F1 is EDMA, the Amazon Elastic-DMA (EDMA) driver [44]. Using this driver with the example, we only achieved 1GB/s PCIe bandwidth with a 1 MB data transfer (the bandwidth varies depending on size of data). Other F1 users also reported similar bandwidths on the F1 Developer Forum [45, 46]. This is low considering that the F1 platform uses PCIe Gen3x16, which has a theoretical maximum bandwidth of 16 GB/s for each direction (32 GB/s combined). Through investigation, we found that using the XDMA driver, Xilinx's PCIe DMA driver, can more than double the bandwidth. The main difference we noticed between the drivers is how the DMA read/write functions are implemented. EDMA uses transient buffers to hold data to send/receive to/from DMA, whereas XDMA directly uses users' data buffers by pinning down their pages. We also found that using `posix_memalign` [47] to align the buffer with the page size drastically improves the bandwidth.

With the above changes, we measured the bandwidths for different transfer sizes, shown in Fig. 6. The highest bandwidth achieved was slightly higher than 8GB/s (7GB/s for a 1MB transfer). This is significantly improved from the previous 1GB/s result, but is still roughly half of the theoretical maximum bandwidth. Xilinx engineers confirmed that this is the expected bandwidth on F1, and AWS engineers told us that they are working on improving the shell to allow higher bandwidths[1]. For Memcached, we found that PCIe bandwidth is not the bottleneck, as data sizes are typically not very large and the 10 Gbps network bandwidth becomes a bottleneck before PCIe.

The VN2F framework is responsible for packet transfers

[1]The bandwidth measurement was done using FPGA Developer AMI v.1.3.3. Subsequent versions may achieve different results.

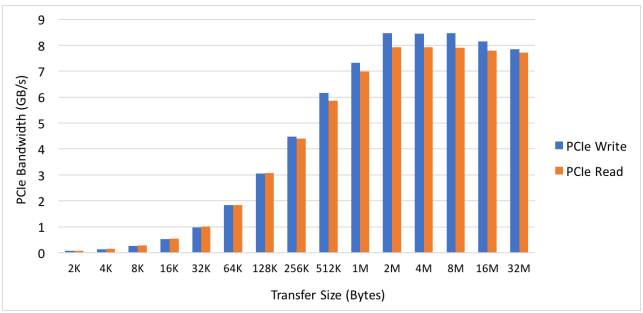

Figure 6: PCIe bandwidth on F1.

between host and FPGA, and since this is an extra latency that is incurred due to the FPGA not being directly connected to the network, it needs to be minimized. In order to achieve high PCIe DMA bandwidth, data to be transferred has to be in contiguous memory. The VN2F software then moves network packets received from NIC to a contiguous buffer, and also inserts a preamble at the start of each packet to denote packet boundaries. Then, the contiguous buffer is transferred to FPGA's Rx on-chip buffer via DMA. The VN2F hardware examines the contiguous block of data and splits them up into individual packets based on the inserted preamble. The reverse happens on the path from FPGA to CPU, where the preamble is inserted by the FPGA, the contiguous block of data is transferred from Tx on-chip buffer to CPU, and the CPU splits up the packets to be sent out to the NIC. We measured this VN2F Rx/Tx processing and data transfer to take about 20∼50μs for each direction, depending on the amount of data being transferred. Some of the Rx/Tx time can be overlapped though, as we can run them concurrently on separate threads.

4.2 Network Stack

Our network stack implementation is based on the open-source implementation [48], which can support a 10 Gbps network, matching the network bandwidth of the `f1.2xlarge` instance. Since Memcached uses the TCP protocol for the server-client communication, the network stack includes modules handling the TCP/IP protocols, as well as an ARP module for resolving MAC addresses between server and clients.

One of the key optimizations we did to the network stack was to minimize the number of packets sent over the network to overcome the packet-per-second limitation described in Section 3.1. One possible approach would be to use the Nagle's algorithm [49] – it intends to improve network efficiency by reducing the number of outgoing TCP packets. The algorithm postpones a packet transmission for a TCP connection until enough of the to-be-transmitted data has been accumulated.However, the drawback of Nagle's algorithm is the increase in latency due to the postponed transmission and the additional memory accesses required for accumulating packets. The latency is even more severely impacted when delayed acknowledgment is enabled at the receiver end of the connection.

Since long latency is undesirable for a Memcached server, we opted to not use Nagle's algorithm, and created our own custom solution. We added a packet-stitching block on the Tx path between the Memcached server and the TCP layer. The packet stitching block receives packets from the Memcached server, examines the packets' TCP connection IDs, and stitches consecutive incoming packets together into a bigger packet if the packets are heading to the same TCP connection. Then, the packet stitching block sends aggregated packet data to the TCP layer in any one of the three cases: 1) when the accumulated size has reached the maximum segment size, 2) when the new incoming packet is for a different TCP connection as the accumulated ones, or 3) when there have been no packets coming from the Memcached server for a certain amount of time (e.g., 1024 cycles). Because only consecutive packets heading to the same connection can be combined, we can simply use a FIFO to queue the incoming packets for the current TCP connection and flush out the FIFO to the TCP layer in any one of the three cases above. Our approach does not require any complex storage method to accumulate packets for multiple connections.

Compared to Nagle's algorithm that only transmits a packet when the TCP connection has enough accumulated data or has received all acknowledgements (Linux by default delays outgoing ACKs for up to 40ms), our approach introduces a significantly lower latency that is up to 1024 cycles (4μs with a 250MHz clock) at the worst case. Although only consecutive packets can be combined, we found that this approach is effective in reducing packet counts for the Memcached application. The reason is that the Memcached server always responds to a pipelined request (to be described in Section 4.3.1) with a sequence of consecutive packets responding to the same connection, providing enough opportunity to stitch packets.

4.3 Memcached Core

Our Memcached core is fully pipelined with an initiation interval (II) of 1, meaning that it can accept a new piece of data every cycle and can process multiple requests concurrently. Incoming Memcached requests are decoded in sequence and then partitioned into key and value pairs. The key component of a single request is hashed to a hash value, which is then used to look up the memory address to store the corresponding value. Once the address is determined, the core stores/retrieves the value to/from DDR memory and creates a response to return to the client. This core is built on top of an existing open-source work [22] and it has been enhanced to support Memcached pipelining.

4.3.1 Memcached Pipelining

Pipelining in Memcached permits packing multiple requests into a single network packet. This reduces the packet processing overhead. With a pipelining of 10, each packet has 10 requests, meaning only one packet has to be processed per 10 Memcached requests. Whereas, without pipelining, a packet has to be processed for every single request. Hence, pipelining is an important feature to reduce packet-processing overhead. With the PPS limitation described above, if pipelining is not used, the maximum throughput of Memcached would be limited to the PPS value.

When multiple requests are packed into a single network packet, we need to have the logic to break the packet into multiple individual requests. Also, we need to handle the cases where there is an incomplete request at the beginning and/or the end of a network packet (when one packet cannot hold the full pipelined request).

To support Memcached pipelining, we implemented a block

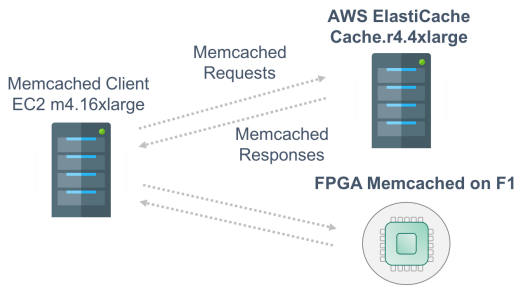

Figure 7: Experimental setup on AWS.

on the Rx path between the network stack and the Memcached core. As packets reaches the network stack, this block peeks into the headers of incoming requests to determine if a complete request has been received. Only complete requests are forwarded to the Memcached core for processing, and incomplete requests are left inside the buffer until more data arrive. Requests are passed to the Memcached core one at a time through a streaming interface, so this block also takes care of data re-alignment and end-of-packet signaling to properly break down network packets into individual Memcached requests.

5. EXPERIMENTAL STUDY

In this work, we compare the throughput, latency, and cost-efficiency of our FPGA Memcached server accelerator to ElastiCache, an AWS-managed Memcached service running on CPUs. Fig 7 shows our experimental setup on AWS. We use an EC2 m4.16xlarge instance, which has 64 vCPUs, to run the Memcached client, `memtier_benchmark` [50] – an open-source Memcached benchmarking tool by Redis Labs. Using the client instance, we connect to an ElastiCache instance (in the same availability zone) to measure its throughput and latency, then also connect to to our FPGA Memcached server (also in the same availability zone and placement group).

Table 1: FPGA Memcached server area.

	LUTs	Registers	BRAMs
Memcached Core	32.5 K	4.3 K	147
Network Stack	27.2 K	2.9 K	275
VN2F HW (without F1 Shell)	5.7 K	1.1 K	233
F1 Shell	181.4 K	22.8 K	321
Total	246.8 K	31.2 K	976
Device Utilization	20.9 %	13.2 %	45.2 %

The FPGA Memcached server runs at 250 MHz and its resource usage for the F1's UltraScale+ FPGA is shown in Table 1. Among all major components, the AWS F1 shell uses the lion's share of the resources (73% of the entire design's LUT and register usage and 33% of the design's block RAM usage). The F1 shell alone takes up about 15% of the total available LUTs on the FPGA. The network stack also uses a considerable number of block RAMs, which are primarily for storing each TCP connection's state information (e.g., the sequence/acknowledge numbers used by each connection in TCP header). This storage space is allocated for supporting up to 10,000 concurrent connections, and can be reduced proportionally if fewer concurrent connections are required. The majority of block RAMs used by the VN2F HW are for the on-chip buffers on the Rx and Tx paths. The entire design uses about 21% of the total LUTs, 13% of registers, and 45% of block RAMS on the FPGA.

For ElastiCache, we choose the instance type that is closest to the `f1.2xlarge` instance in terms of network bandwidth, RAM capacity, and cost, as shown in Table 2.

Table 2: ElastiCache and F1 Instance Comparison.

Type	vCPU	RAM	Network Bandwidth	Cost
Cache.r4.4xlarge	16	101.38 GB	Up to 10 Gbps	$1.82/hr
f1.2xlarge	8	122 GB	Up to 10 Gbps	$1.65/hr

The `memtier_benchmark` can be configured for different data sizes, numbers of connections (number of users connected to the Memcached server), types of Memcached commands, and amount of Memcached pipelining (among many other options). We use Memcached pipelining of 16 (16 Memcached requests are combined into a packet) for `set` and `get` commands with 100-byte data sizes and vary the number of connections. We also set the ratio of `set` and `get` commands to be 1:1. When using many connections (results below), we ran the `memtier_benchmark` with up to 40 threads (threads=40) with up to 35 connections each (clients=35). Each result shown below is an average of 3 `memtier_benchmark` runs.

5.1 Results

Fig. 8 shows the throughput (ops/sec) of ElastiCache and our FPGA Memcached server accelerator as we vary the number of connections from 5 to 1400. An operation is a `set` or a `get`. A Memcached server typically connects to many clients, thus an understanding of how the performance scales with different numbers of clients is important. We observe that the FPGA Memcached outperforms ElastiCache starting from 5 connections all the way to 1400 connections. ElastiCaches throughput is capped at around 1.4 million ops/sec with 80 connections and remains about the same when the number of connections is increased to 1400, as the CPU can no longer keep up with processing of network and Memcached commands. As for our FPGA Memcached, throughput continues to increase to reach over 11 million ops/sec with 1400 connections, corresponding to 9× better throughput compared to ElastiCache. At 1400 connections, we have reached the PPS limit of the AWS network and are bottle-necked by it[2]. The FPGA hardware is still not fully utilized, hence without the PPS limit, the FPGA's throughput would be even higher.

Latency is another key metric for Memcached, as databases need to respond as quickly as possible to front-end web servers in order to provide a responsive experience for web users. Fig. 9 shows the latency of ElastiCache and our FPGA Memcached as we vary the number of connections. Note that the reported latency is the total round-trip time (RTT) measured at the client side, which includes round-trip

[2]700K packets/second × 16 ops/packet = 11.2M ops/sec. We noticed that the throughput reported by `memtier_benchmark` can be slightly higher than the PPS×Pipelining amount when there are many connections. We think this is due to a minor average miscalculation of the tool, but as we are applying it to both ElastiCache and the FPGA Memcached, the minor discrepancy is acceptable for this comparative analysis.

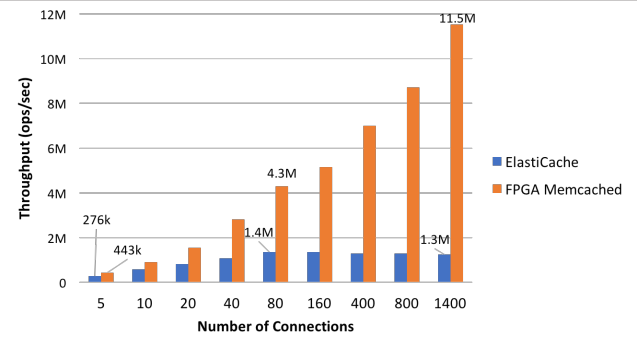

Figure 8: Throughput of FPGA Memcached vs. ElastiCache.

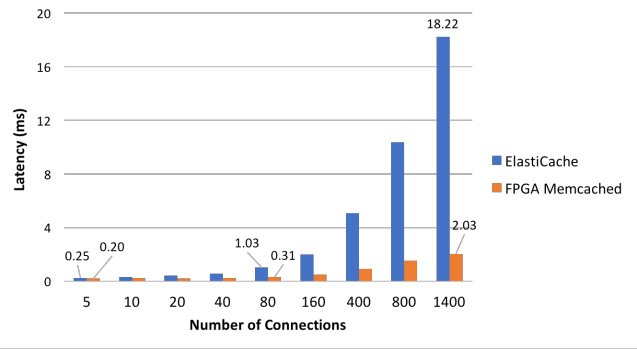

Figure 9: Latency of FPGA Memcached vs. ElastiCache.

network latency in addition to Memcached latency. As the number of connections increases, ElastiCaches latency spikes to 18.22 ms, as the CPU can no longer keep up with the number of connections and their requests. For the FPGA Memcached, the RTT latency is sub-millisecond up to 400 connections, and increases up to 2.03 ms with 1400 connections, at which point the latency is 9× better for LegUp. Again, we are reaching the max PPS at this point.

As mentioned, this latency is the total RTT including network latency. The latency of the FPGA Memcached server accelerator alone is measured to be 0.2∼0.3 ms with 1400 connections (includes the network stack and Memcached server latency, but excludes the delay on the network link between instances).

For many Memcached users, 11M ops/sec may not be necessary. However, total cost of ownership (TCO) is important for everyone. To illustrate the cost-efficiency of the FPGA Memcached, we show the throughput/dollar (ops/sec/$) in Fig. 10. With 5 connections, our FPGA Memcached is 77% more cost-efficient, and with 1400 connections, it is 10× more cost-efficient. In other words, if we allow multiple tenants in our single FPGA Memcached server, where the throughput and the memory space are split across users, the TCO can be greatly reduced.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a FPGA Memcached server accelerator deployed on AWS F1. Results show that our FPGA Memcached achieves 9× better throughput and latency with 10× better cost-efficiency vs. Amazon's Elasti-

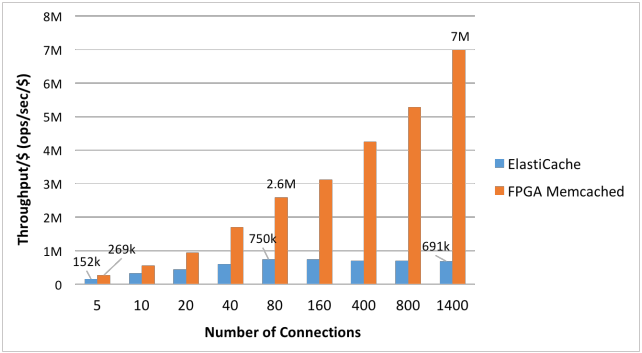

Figure 10: Throughput/$ of FPGA Memcached vs. ElastiCache.

Cache. A live demo of the Memcached accelerator running on F1 can be accessed on our website [9]. As part of the FPGA Memcached server, we also presented the Virtual-Network-to-FPGA (VN2F) Framework, which enables network access to the FPGA on F1, where the FPGA is not directly connected to the FPGA. We presented a quantitative analysis of the achievable network bandwidth and the packets-per-second limitation, as well as the PCIe bandwidth of the AWS F1 instance.

For future work, we would like to experiment with adding more computations to the FPGA. Even with 11M ops/sec, the FPGA LUT area is only about 20% utilized. There is abundant room to add more computations, such as an in-line compression module to increase the effective cache size, or in-line encryption for added security. Due to the pipelined nature of FPGA hardware, adding more computations in-line does not necessarily affect the throughput – it is possible to maintain throughput at 11M ops/sec. We also plan to release a *network shell* for F1, which integrates our VN2F framework with an FPGA network stack, so that users can simply design the core application in software using LegUp HLS and create a complete networking accelerator on F1.

7. REFERENCES

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

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

[3] Intel FPGAs Power Acceleration-as-a-Service for Alibaba Cloud. [Online]. Available: https://newsroom.intel.com/news/intel-fpgas-power-acceleration-as-a-service-alibaba-cloud/

[4] Tencent FPGA Cloud Server. [Online]. Available: https://cloud.tencent.com/product/fpga

[5] FPGA-accelerated Cloud Server. [Online]. Available: https://www.huaweicloud.com/en-us/product/fcs.html

[6] Xilinx FPGAs on the Nimbix Cloud. [Online]. Available: https://www.nimbix.net/xilinx/

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

[8] 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 *USENIX NSDI*, Lombard, IL, 2013, pp. 385–398.

[9] LegUp Memcached Live Demo. [Online]. Available: http://www.legupcomputing.com/main/memcached_demo

[10] Intel Stratix chip performs over 10 trillion calculations per second. [Online]. Available:

https://www.techspot.com/news/74235-intel-stratix-chip-performs-over-10-trillion-calculations.html
[11] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ACM/IEEE ISCA*, 2014, pp. 13–24.
[12] EC2 Instance Pricing - Amazon Web Services (AWS). [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/
[13] Intel FPGA SDK for OpenCL Overview. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
[14] Vivado High-Level Synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[15] LegUp Computing: High-Level Synthesis for Any FPGA. [Online]. Available: https://www.legupcomputing.com/
[16] High-Level Synthesis with LegUp. [Online]. Available: http://legup.eecg.utoronto.ca/
[17] Amazon Relational Database Service (RDS). [Online]. Available: https://aws.amazon.com/rds/
[18] Microsoft Azure SQL Database. [Online]. Available: https://azure.microsoft.com/en-ca/services/sql-database/
[19] Alibaba Cloud ApsaraDB for RDS. [Online]. Available: https://www.alibabacloud.com/product/apsaradb-for-rds
[20] Amazon ElastiCache. [Online]. Available: https://aws.amazon.com/elasticache/
[21] Memcached Commands. [Online]. Available: https://github.com/memcached/memcached/wiki/Commands
[22] HLS implementation of Memcached pipeline. [Online]. Available: https://github.com/Xilinx/HLx_Examples/tree/master/Acceleration/memcached
[23] TCP/IP Offload Engine Design Using Vivado High-Level Synthesis and Vivado. [Online]. Available: https://github.com/Xilinx/HLx_Examples/tree/master/Acceleration/tcp_ip
[24] TCP/IP Network Stack for FPGAs. [Online]. Available: https://github.com/fpgasystems/fpga-network-stack
[25] Azure Redis Cache. [Online]. Available: https://azure.microsoft.com/en-ca/services/cache/
[26] ApsaraDB for Memcache. [Online]. Available: https://www.alibabacloud.com/product/apsaradb-for-memcache
[27] Google Cloud Memorystore. [Online]. Available: https://cloud.google.com/memorystore/
[28] Redis. [Online]. Available: https://redis.io/
[29] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, Berkeley, CA, USA, 2014, pp. 429–444.
[30] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *USENIX NSDI*, Lombard, IL, 2013, pp. 371–384.
[31] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015.
[32] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *USENIX NSDI*, Seattle, WA, 2014, pp. 401–414.
[33] 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 *ACM SOSP*, 2017, pp. 137–152.
[34] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 36–47, Jun. 2013.
[35] AWS Shell Interface Specification. [Online]. Available: https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md
[36] Regions and Availability Zones - Amazon Elastic Compute Cloud. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html
[37] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. [Online]. Available: https://iperf.fr/
[38] 2 Million Packets Per Second on a Public Cloud Instance. [Online]. Available: http://techblog.cloudperf.net/2016/05/2-million-packets-per-second-on-public.html
[39] Benchmarking Throughput and Packet Count with iperf3. [Online]. Available: https://discuss.aerospike.com/t/benchmarking-throughput-and-packet-count-with-iperf3/2791
[40] Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA. [Online]. Available: https://www.altera.com/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html
[41] Intel FPGA IP for Ethernet Products Portfolio. [Online]. Available: https://www.altera.com/products/intellectual-property/ip/interface-protocols/ethernet.html
[42] Data Plane Development Kit. [Online]. Available: https://dpdk.org/
[43] CL_DRAM_DMA CustomLogic Example. [Online]. Available: https://github.com/aws/aws-fpga/tree/master/hdk/cl/examples/cl_dram_dma
[44] Using AWS EDMA in C/C++ application. [Online]. Available: https://github.com/aws/aws-fpga/tree/master/sdk/linux_kernel_drivers/edma
[45] PCIe performance measure on F1 instance. [Online]. Available: https://forums.aws.amazon.com/thread.jspa?messageID=785591785591
[46] DMA bandwidth test. [Online]. Available: https://forums.aws.amazon.com/thread.jspa?messageID=812716812716
[47] POSIX_MEMALIGN(3) Linux Programmer's Manual. [Online]. Available: http://man7.org/linux/man-pages/man3/posix_memalign.3.html
[48] D. Sidler, Z. Istvn, and G. Alonso, "Low-latency tcp/ip stack for data center applications," in *FPL*, Aug 2016, pp. 1–4.
[49] J. Nagle, "Congestion control in ip/tcp internetworks," *SIGCOMM Comput. Commun. Rev.*, vol. 14, no. 4, pp. 11–17, Oct. 1984.
[50] NoSQL Redis and Memcache traffic generation and benchmarking tool. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark