# An FPGA Memcached Appliance

Sai Rahul Chalamalasetti
University of Massachusetts Lowell[§]
Hewlett Packard[‡]
Sairahul.Chalamalasetti@hp.com

Kevin Lim
Hewlett Packard Labs[†]
Kevin.Lim2@hp.com

Mitch Wright
Hewlett Packard[‡]
Mitch.Wright@hp.com

Alvin AuYoung
Hewlett Packard Labs[†]
Alvin.Auyoung@hp.com

Parthasarathy Ranganathan
Hewlett Packard Labs[†]
Partha.Ranganathan@hp.com

Martin Margala
University of Massachusetts Lowell[§]
Martin_Margala@uml.edu

## ABSTRACT

Providing low-latency access to large amounts of data is one of the foremost requirements for many web services. To address these needs, systems such as Memcached have been created which provide a distributed, all in-memory key-value store. These systems are critical and often deployed across hundreds or thousands of servers. However, these systems are not well matched for commodity servers, as they require significant CPU resources to achieve reasonable network bandwidth, yet the core Memcached functions do not benefit from the high performance of standard server CPUs.

In this paper, we demonstrate the design of an FPGA-based Memcached appliance. We take Memcached, a complex software system, and implement its core functionality on an FPGA. By leveraging the FPGA's design and utilizing its customizable logic to create a specialized appliance we are able to tightly integrate networking, compute, and memory. This integration allows us to overcome many of the bottlenecks found in standard servers. Our design provides performance on-par with baseline servers, but consumes only 9% of the power of the baseline. Scaled out, we see benefits at the data center level, substantially improving the performance-per-dollar while improving energy efficiency by 3.2X to 10.9X.

## Categories and Subject Descriptors

B.5.1 [**Design**]: Styles; D.3.2 [**Programming Languages**]: Java

## General Terms

Algorithms, Performance, Design

## Keywords

Memcached appliance; FPGA; Low Power; Energy Efficiency; Data Centers

## 1. INTRODUCTION

Today's web services generate an unprecedented amount of structured and unstructured data. Because service providers derive tremendous value from obtaining fast access to such data, it is critical to have the right infrastructure for data storage and retrieval. Scale-out technologies and key-value stores, such as Memcached, have become the de facto standard for deploying such infrastructure, as most of the largest content-serving systems rely on the ability to scale quickly in response to increasing client traffic and data generation.

One of the most fundamental aspects to Memcached is its all in-memory design. By solely utilizing memory, Memcached provides substantially lower latency data access than other comparable storage systems. This low-latency performance is critical for interactive web applications, evidenced by web service providers such as Facebook and Zynga dedicating an entire tier of servers to Memcached. More broadly, Memcached is representative of a growing shift towards scale-out, in-memory applications; including MonetDB, VoltDB, and HANA.

Memcached posts unique challenges to existing servers due to its architecture. Memcached is created to be a key-value cache that stores key-value pairs in a distributed hash table. Its goal is to provide a fast caching layer in-between a web server and its backend storage (e.g., a database running MySQL). It is geared towards caching small values and keys, often on the order of 200 bytes or smaller. Thus Memcached clients generate many small requests to the server, placing importance on the server's ability to handle and generate responses to many small packets. Despite the small key and value sizes, often each server has 10's to 100's of GB of memory, placing an importance on large memory capacities. On the other hand, Memcached is designed as a very simple cache which offers minimal functionality on its data, and thus does little computation.

Based on the Memcached characteristics, alternate architectures can offer benefits for both performance and efficiency. There has been some prior work looking at alternate CPU and GPU architectures for Memcached, showing promise [1]. Unlike prior work, we address the bottlenecks by designing a system that more fundamentally addresses Memcached requirements of tightly integrated networking and memory coupled with light compute: we instead utilize an FPGA to implement a Memcached appliance. We leverage the FPGA to provide specialized logic to implement the base functionality of the Memcached server, consisting of accepting requests, finding the requested key-value pair, and performing the requested get or set operation.

To convert the Memcached software to an FPGA implementation we address several challenges, including handling high network rates, variable length keys and values, and dynamic memory allocation. We show that an implementation is feasible, and through its integration and specialization achieves comparable peak performance to a standard server while consuming only 9% of its power. At datacenter scale, an FPGA Memcached appliance can provide up to 5.9X performance-per-TCO-$ improvement versus existing servers by providing high performance, low power alternatives.

Sai Rahul Chalamalasetti worked on this research project while he was a summer intern in Hewlett Packard.

[§]1 University Avenue, Lowell, MA, USA, [†] 11445 Compaq Center West Drive, Houston, TX

[‡] 1501 Page Mill Road, Palo Alto, CA

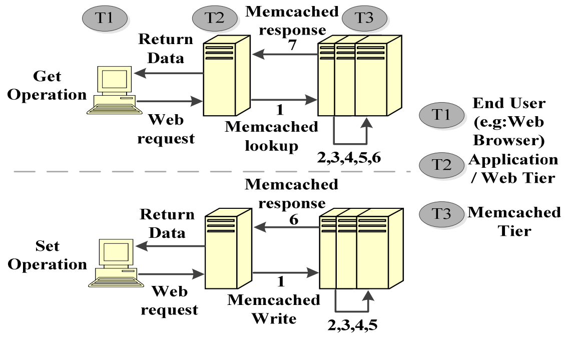


Figure 1: Memcached architectural diagram and use case

The rest of the paper is organized as follows. Section 2 provides a background on the Memcached software and key bottlenecks. Section 3 provides motivation behind implementing Memcached on an FPGA. Section 4 goes into our detailed Memcached appliance design description. Section 5 provides an evaluation of the performance and power of the appliance, and its impact at the datacenter level. Section 6 discusses future work, and Section 7 discusses related work. Section 8 concludes the paper.

## 2. BACKGROUND

### 2.1 Memcached overview

Memcached is an in-memory distributed data cache primarily used at the application-layer of a Web stack. Many large and well-known content-serving systems such as Facebook, Google App Server, Twitter, Flickr, Zynga, and Wikipedia rely on Memcached to alleviate the load on their back-end databases and provide clients with low-latency and high-throughput access to content.

Given its critical role in so many Web 2.0 infrastructures, Memcached is an important application to consider in the class of Web 2.0 workloads. A typical deployment places a cluster of Memcached servers in between the Web (or application) tier and back-end database. The Memcached servers are used by the Web tier to cache popular objects, such as pre-processed text, images, session state, or page counters, which would have otherwise been retrieved from the database directly, possibly via disk I/O. Based on public and inferred numbers, Memcached clusters can vary in size from as small as 30 physical machines to thousands of machines with a significant memory footprint.

A Memcached cluster provides a lightweight, distributed *hash table* for storing small objects (up to 1 MB), exposing a simple set/get interface. Each object's key is used to determine which individual Memcached server within the cluster will store the object. Typically, a hash function is chosen to balance keys evenly across the cluster [2]. Individual Memcached servers do not communicate with each other, as each server is responsible for its own independent range of keys. Because the servers do not interact, the performance of a single Memcached server can be used to generalize the behavior of an entire cluster.

Due to its simplicity, Memcached is the de facto standard for distributed caching, and is used extensively. In fact, it plays such a critical role in Web infrastructures that larger deployments, such as those of Zynga and Facebook, dedicate physical servers to run *only* Memcached [3,4]. Therefore designing better performing and more cost-effective servers for Memcached workloads can offer significant benefits.

### 2.2 Memcached architecture in-depth

Figure 1 illustrates the Memcached architecture and how it is used in web infrastructures. Memcached provides a simple set of commands for accessing data, with the two most dominant commands being get and set [5]; other commands, for example, include add, increment, and compare and swap. Gets and sets have ratios of up to 30:1 gets:sets seen in real world deployments [5]. Below we review the access paths for get and set commands.

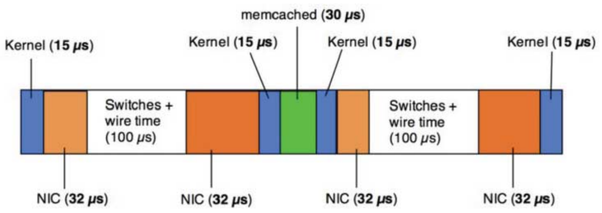


Figure 2: Latency bottleneck analysis on a standard server [6]

**Get:** A get will retrieve the value associated with the user-specified key, if it is located in the Memcached server. If it is not found, it is up to the user to determine where to obtain the proper value. (Typically a miss will then lead to a database lookup and/or recomputation to determine the appropriate value.) A requesting client first determines which Memcached server to access. A key, which can potentially be an ASCII string up to 250 characters long, is sent to the server in a message including the command (get), the key length, and any optional message flags.

A get performs the following steps: 1. The request is received at the network interface and is sent to the CPU. 2. The Memcached server will read the data out of the request packet to identify the key. 3. The server performs a hash on the key value to translate the key into a fixed 32 bit value. 4. The value is used to index into a hash table that stores the key-value pairs. 5. If the key is found, its value data is accessed and prepared to be sent back to the client in a response message. 6. The entry corresponding to the key is also promoted in a doubly linked list that is used to perform least recently used (LRU) replacement if the Memcached server is full. 7. The server either sends out a reply to the client with the key-value pair, or a message indicating that the key was not found.

**Set:** A set will write the specified key-value pair into the Memcached server's storage. Values are typically small objects, often a few hundred bytes large. To handle memory management, Memcached uses *slab allocation*. In slab allocation, Memcached allocates a large chunk of memory and breaks it up into smaller segments of a fixed size according to the slab class' size. This method of allocation reduces the overhead of dynamically allocating and deallocating many small objects. Memory is therefore handled in fixed sizes, with values stored in the smallest slab class that will accommodate the size of value. (Thus there may be some internal fragmentation per object.) When storing a new value, the LRU list for the slab class is checked to see if the last element can be evicted. If there are no free segments within a slab class, a new slab is allocated if there is free memory.

A set performs the following steps: 1. The request is received at the network interface and is sent to the CPU. 2. The server will read the data from the packet to identify the key, value, flags, and total message size. 3. The server then requests a slot from the correct slab memory class to store the key-value pair. The item is promoted to the most recently used position of the slab's LRU list. 4. After copying the data into the slab element, the server performs a hash on the key to determine the hash bucket to store the data. 5. The data is written to the head of the hash bucket. 6. A reply is sent back to the client to indicate the request is completed.

## 2.3 Bottlenecks

As described in the previous section, Memcached servers primarily handle network requests, perform hash table lookups, and access data. This design implies that network latency is quite important, as is providing fast access to memory. Figure 2 shows previously profiled [6] Memcached latency breakdowns on modern servers. To gain additional insight into Memcached performance, we performed similar tests using stress test clients that send a mixture of 40:1 gets:sets, trying to achieve the maximum throughput possible.

**Performance:** Both the prior analysis and our performance analysis show that there is significant time spent in the networking stack, and in calculating the hash values of the keys. In general the CPU shows poor utilization (low IPC), despite thousands of requests per second being processed, and spends most of its time processing work other than the core Memcached functions.

The results highlight some of the key bottlenecks present in current Memcached systems. Most of the network processing, for example TCP/UDP, is carried out in the software stack. In a Memcached application, requests initiate from web application servers. Due to its architecture with many small, cached objects, multiple web application servers will generate millions of requests, which results in the Memcached server spending most of its processing time to handle and keep track of the requests. For example, a study from RAMCloud research by Stanford [6] has shown the distribution of latencies over the whole Memcached packet processing, where 64 $\mu$s is spent for transferring packet from Network Interface Card, 30 $\mu$s on the Linux software stack, and merely 30$\mu$s on the Memcached software. To measure network latency numbers specifically for UDP, we carried out a test between two servers, with one acting as client, and another as the Memcached server. Our tests showed that the NIC plus Linux UDP processing took 25$\mu$s for sending, and another 25$\mu$s for responding back to client. Both results confirm that network processing is a major bottleneck for Memcached servers.

**Power:** Our measured baseline server has two Intel Xeon CPUs (12 cores total) with 64 GB of DRAM. It consumes 258W of total power, of which 190W of power is distributed between the two CPUs in the system. The rest of 64W and 8W is consumed by DRAM memory, and 1GbE Ethernet NIC. Compared to the total system utilization, the CPU consumes a disproportionate amount of power. It is therefore one of the biggest challenges to achieving better energy efficiency for Memcached servers.

**Low-power systems:** While low power Atom or ARM systems may seem like an ideal alternative, our experiments with an Atom-based system showed it only achieved 8-12% of the performance of the Xeon system, largely due to its inability to handle the high network rates. As the overall performance-energy efficiency is lower than our baseline system, we only consider our Xeon server in the rest of the paper.

## 3. UTILIZING FPGAS FOR MEMCACHED

### 3.1 Opportunities for an FPGA appliance

Based on the discussion in the previous section, it is apparent that existing servers are not well matched for Memcached. Instead we consider alternate architectures that can have both better performance and better energy efficiency than existing servers. We specifically design an FPGA Memcached appliance where the networking, compute, and memory are tightly integrated and software overhead is removed. By appliance we refer to a system that specifically performs the Memcached protocol, as opposed to a general purpose server.

An FPGA has several advantages versus a traditional CPU; one of the foremost is power consumption. It is difficult to reduce CPU power as static power of processors is growing as transistors become smaller. The millions of transistors required to implement multiple cores in a general purpose processor affect the total power consumption of the system, and based on our Memcached analysis, do not provide significant performance benefits. Another drawback of traditional CPUs is their high frequency of operation (2-3 GHz), which is directly proportional to dynamic power consumption. Conversely, an FPGA can be customized for the application being run, providing high performance without needing general purpose processors or high clock frequencies.

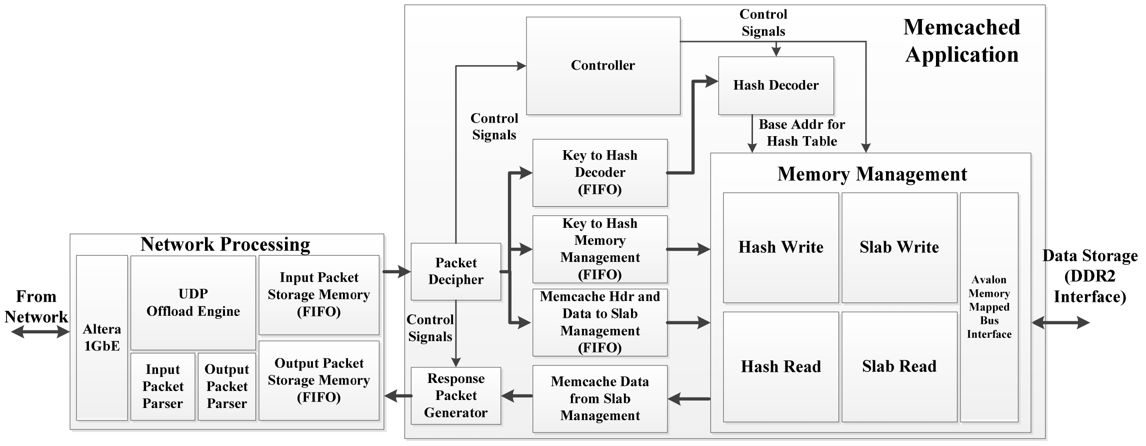
FPGAs also offer a means to reduce the network latency. The study from Stanford on latencies of transferring packets from network to Memcached software illustrated that 50% and 25% of total processing time is spent on NIC transfer and the software stack, respectively. Thus network processing latency is currently a significant bottleneck. One option is to replace the software stack functionality in hardware by using Offload Engines (OEs), such as TCP/IP or UDP OE. However, a solution of directly augmenting OEs to a NIC card would only increase performance by 25%. An efficient hardware interface to server CPUs is needed to address the major portion of NIC interface latency.

In our approach, we propose to bring the main Memcached application *closer* to the NIC and OE so that we can decrease the 75% network-related latency that CPU systems suffer. We map the NIC, OE and Memcached application running all together on a single FPGA. Since, FPGAs are traditionally used for networking applications, such as switches, routers, and flow control, etc., we leverage the FPGA devices and network connections on our FPGA board. Due to the maturity of FPGA devices in networking applications, Ethernet IPs to communicate with the MAC and read Ethernet frames are readily available.

### 3.2 Key Implementation Challenges

Despite FPGAs providing many solutions to problems that are encountered by contemporary CPU systems, a direct replacement of CPU system with a standalone FPGA is a cumbersome task. One main reason is the use of high level programming languages and tool chains for CPU systems which are designed to ease programming burden and quickly allow users to check the operation of the written application. On the other hand, FPGAs are mainly programmable through hardware description languages (such as VHDL/Verilog), which take a longer time to design and test on the board. Even though many alternatives such as C to gates, Impulse C, Mitrion-C, etc. have been proposed, they do not effectively optimize the algorithm to be mapped on the hardware. Therefore implementing applications in an FPGA must be done on relatively mature software that has wide-spread adoption to justify the programming overhead.

Apart from programming support, the other challenge of porting software code written for CPUs to FPGAs is the direct portability of software algorithms to FPGA hardware. Therefore, alternative, FPGA-friendly solutions have to be devised and implemented. For example, Memcached uses Least Recently Used (LRU) replacement policy to select data to be replaced when its capacity is full. These LRU lists are maintained using a doubly linked list, a structure that is not efficiently implementable on hardware or FPGAs. Therefore, an alternative replacement algorithm must be used in our designs. There are several other similar algorithm challenges faced throughout the Memcached appliance, such as



Memcached Application
Control Signals
Controller
Hash Decoder
Base Addr for Hash Table
Control Signals
Key to Hash Decoder (FIFO)
Memory Management
Network Processing
Key to Hash Memory Management (FIFO)
Hash Write
Slab Write
Packet Decipher
Input Packet Storage Memory (FIFO)
UDP Offload Engine
Avalon Memory Mapped Bus Interface
Data Storage (DDR2 Interface)
From Network
Memcache Hdr and Data to Slab Management (FIFO)
Altera 1GbE
Control Signals
Hash Read
Slab Read
Input Packet Parser
Output Packet Parser
Output Packet Storage Memory (FIFO)
Response Packet Generator
Memcache Data from Slab Management

Figure 3: Overall FPGA Memcached appliance architecture

handling the variable length data sizes normally handled by dynamic slab allocation. Doing dynamic memory allocation of varying sizes is difficult in hardware, and thus we designed the system to enable the user to specify at run time the number of slabs to use, and utilize power of 2 sizes for the slab classes.

Lastly, FPGAs do not achieve the multi-GHz frequency of a general purpose processor. Thus any designs implemented on an FPGA must be carefully programmed to take advantage of the parallelism and support for complex operations that FPGAs offer.

## 4. FPGA MEMCACHED DESIGN DETAILS

### 4.1 FPGA Appliance Overview

The proposed system shown in Figure 3 depicts the block level diagram of our FPGA-based standalone Memcached appliance architecture. Our design consists of two main blocks: a networking block and a Memcached application block, along with an Altera TSE MAC for Ethernet interface, and a DDR2 memory controller interface to store cache data in DRAM modules. Our current design is implemented on an Altera Terasic DE-4 board.

The Memcached system transfers command and data between machines through an Ethernet interface. Hence, in our system initially the command or data packets received on 1Gbit Ethernet port are forwarded to a MAC IP module. The data from the MAC is then transferred to the network *offload engine* (OE), which parses through the input packet flow to detect relevant packets, and transmits required Memcached command and data to later stages by extracting packet header and user data information. The offload engine module is also used to generate the response packets with the data from the Memcached appliance. After the received packet is available from the OE, the data is stored on the FPGA internal memory in a FIFO so that continuous requests from multiple application clients will not be discarded while a prior user command request is being processed.

Subsequent to the OE receiving the network packet and placing it in the FIFO, an *input packet parser* module is then used to read the packet data to distinguish between different operations, for example get and set. Because the Memcached packet format in the Ethernet packet varies with the type of operation, the parser module is used to extract the relevant data needed to complete the requested operation. The parser signals the main controller to start executing the user command when the complete packet data is parsed and stored.

Following the packet parser, the controller signals and monitors the hash decoder and memory management modules in tandem depending on the received command. The hash decoder performs a hash on the key sent by the client and determines the specific address of the requested item in the hash table. Once the address for a specific hash table location is determined, the controller will perform the respective operation depending on the read or write request from the client. The memory management module is responsible for allocating memory to items as they are written to the Memcached storage, or reading data from memory for get operations. In particular Memcached uses slab allocation, which allocates memory in large chunks based on fixed, pre-defined sizes, to help speed allocation of many small (often <1 KB) objects. The memory management either finds items to replace or allocates new memory on writes, if available.

Any data that needs to be sent back to the client is written into a response packet, which is then handled by an *output packet parser* module. In the case of set operations, these responses are a simple “complete” message, and in the case of get operations that do not find a key, they are a simple “miss” message.

#### *4.1.1 Limitations*

Our Memcached appliance is currently in the early prototype stage as we add more functionality. In terms of application limitations, whereas the baseline Memcached software supports multiple commands [7], we only support the two most common ones, get and set. There are very modest logic requirements to support these operations (shown in 5.2.1), and we expect there would only be minor additional logic to support the other operations as well.

Similarly while the software supports variable length keys, we currently support a fixed 64 byte key size, which is a common size according to previous studies [5]. While most of our modules are designed to accommodate variable length keys, and our memory system supports up to 256 byte keys, there are several of our state machines and serializing logic that need to be modified to support variable length keys. We plan to fix these shortcomings in the near future. While our appliance takes in and stores all of the flags and fields from full set and get messages, we currently do not consider the extra fields, mainly time stamp during Memcached operation. In the Memcached software, the time stamp is not used to proactively remove expired items, and thus we expect it is feasible to support the time stamps in our design.

Regarding networking limitations, we currently use a UDP interface to our Memcached appliance, as opposed to the more ubiquitous TCP interface. This limitation is due to only UDP OEs being available in the public domain; once TCP OEs are available, we will modify our design to use those. As Memcached read operations (i.e., gets) are sometimes used with UDP in actual large deployments [8, 9], we expect the drawbacks to be minimal. While UDP does not guarantee reliable transmission, and therefore may be unsuitable to be deployed, we believe our early prototype system is suitable to measure performance and power consumption of memcached appliance on FPGA.

## 4.2 Network Processing

### *4.2.1 Network Offload Engine*

To implement the network Offload Engine we use a UDP OE [10] to extract packet data and header information. The UDP OE communicates with the MAC to read the data transferred to it, and parse for Ethernet IP information. The UDP OE checks the Ethernet IP information, such as IP address, and MAC to confirm that packet is destined for the appliance. It also checks if the input packet is a UDP packet, and stores the Ethernet packet header information in registers (e.g., IP address, source port, destination port). After the packet header information is registered, the OE routes the data from the MAC directly to the core on the other end by signaling that new data is available. In a similar manner, to transfer data onto the network, the module transmitting the data initially passes Ethernet packet information to the UDP OE. The OE checks if the data being sent has a valid IP address, and proceeds to send the data to the Altera MAC by assembling the Ethernet packet header field along with the data.

### *4.2.2 Packet Parser and Storage*

Memcached command packets generally include command information and key(s), which are stored in the Ethernet data field. Additional data may also be included based on the command; for example, set commands will also include the value to store. As data is being extracted by the OE it is sent to an *input packet parser* that separates out the Ethernet header packet information (source network IP address, source port) and data, and places them into the *input packet storage memory*. This memory consists of two FIFOs, implemented in on-chip MRAM blocks. One FIFO contains the entire Memcached request, and the other FIFO contains only request Ethernet packet header information. The header FIFO is only written once when the complete packet is saved to the input packet FIFO, and has a special header padded to it indicating the packet is valid. By writing the packets to a FIFO, the appliance is able to buffer incoming packets while the Memcached backend is in use.

Apart from routing the input UDP packets to the FIFO, the packet storage memory module also takes care of overflow conditions of the FIFO when the FIFO memory is full and any new packet request or ongoing packet request that is being stored has to be discarded. Since the size of memory on FPGA is limited—normally on the order of a few MBs—the size of the input packet FIFO is designed to be only 512 KB. Challenges with dropped packets can potentially be overcome by storing the data on off-chip QDR-II SRAM modules that are common with network processing-oriented FPGA boards. However due to the unavailability of low latency QDR-II SRAM memory blocks on the DE-4 board, we are instead using an Altera FIFO IP based buffer in our current design.

### *4.2.3 Request Response Handling*

The UDP OE is also used to send out responses to Memcached requests. The responses are formed by the Memcached application block with Ethernet packet header and Memcached packet data (key, value). The responses are placed into *output packet storage memory*, which consists of output packet FIFOs analogous to the input packet FIFOs. The *output packet parser* monitors the header FIFO, waiting for a response to be fully generated. Once completed, it is sent to the UDP OE which handles forming a UDP packet and sending it to the network.

## 4.3 Memcached Application

### *4.3.1 Packet Decipher*

After analyzing the structure of Memcached packets [11], we observed that not all request packets have the same data fields. Hence, the *packet decipher* module analyzes the received Memcached request, and stores respective field information for further command processing. The Memcached command packet can be distinguished as two types: request or response packet. Irrespective of request or response packet, the Memcached packet consists of a header field, which includes data such as the opcode, the key length, and total data length. After the header field, the packet format varies depending on the type of operation (set or get). For example, set operations carry data to be stored in the hash table, and thus user data in conjunction with the key are concatenated after the header field. In a similar manner, for a get operation the client sends a packet with basic header field, and a key to index the hash table.

Once the packet decipher module detects data in the input header FIFO, it checks for the special header padding to confirm if the complete packet is available. If the header indicates an error (due to an overflow), the module removes the invalid data from the input header and packet FIFOs. After confirmation of data availability in the input packet FIFO, the decipher module ensures the packet is a Memcached request. Then the rest of the header field information is stored into corresponding field registers so other modules can access them to complete the requested operation. The rest of the packet data is stored into FIFO memory modules so that they can be routed to *memory management* to eventually be saved into DRAM. We use three different FIFO memory modules to store the data before memory management processes it. The key data is needed for both the *hash decoder* and memory management. However the decoder uses a 96-bit data interface, thus two different FIFO's are used to route key data. The third FIFO is used to store user data, which is used primarily for set operations where data has to be stored into DRAM.

### *4.3.2 Response Generator*

Any data retrieved for a get command is sent to a FIFO which stores data prior to the *response packet generator*. The response packet generator assembles the data with the command header. It sends the information to the output packet storage memory, which is processed and sent to the UDP OE. However, when the requested key cannot be found a miss response packet is generated. A set command will lead to the default Memcached set response that it has completed the request.

### *4.3.3 Memcached Controller*

As shown in Figure 3, the controller coordinates the hash decoder and hash and slab memory management to perform the requested command. The controller first directs the hash decoder to perform a hash on the key to determine the hash table address. Once the decoder signals the controller that it has completed, the controller then signals the hash or slab memory management module to perform either get or set operations. For example, during get operations once the hash value is ready the hash memory management performs a lookup on the hash table address. Once the value is retrieved, the controller has the data placed into a FIFO in preparation for the response packet generator. If the data

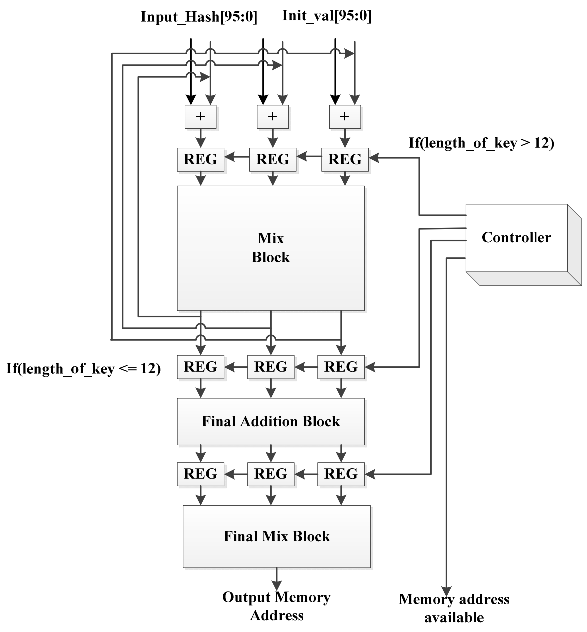


Figure 4: Structural layout of hash function

is not found in the hash bucket, the controller has the response packet generator generate a miss response.

For sets, the controller first commands the slab memory management to store the data item. Once the data is written, the address is recorded and passed to the hash memory manager. The hash decoder performs a hash of the key to determine the hash table location that the new key-value pair should be stored in, and the hash memory manager writes it into the appropriate entry. Once completed the controller instructs the response packet generator to reply to the client with a completion message.

In addition to coordination, the controller also takes of system startup configuration operations, such as creation of slabs, and cleaning DRAM. Since metadata is stored along with Memcached data in DRAM, any invalid data in the DRAM could disrupt operation of the system. Therefore memory cleaning is performed before system is made available.

### *4.3.4 Hash Decoder*

The Memcached application uses a hash table data structure to store and lookup its data, providing fast, constant time access. The hash table is indexed using the key data, which can be a variable number of bytes. The key data is passed through a hash function to generate a 32 bit index value. The Memcached software already uses an efficient hash decoding scheme [12] that can operate on variable length input keys. Thus we used the already implemented function and tried to port an efficient hardware module that replicates its functionality.

The hash decoder is customized to accept inputs of three 32 bit segments (12 bytes or 96 bits) of the input key distributed over three internal variables a, b and c (each 32 bits). Initially, the hash algorithm accumulates the first set of 12 byte key segments with a constant, so that the mix module has an initial state. After the combine state is processed the input variables are passed to the mix state. At this point the counter, *length_of_key* decrements by 12 bytes for each iteration of combine and mix module execution. After each iteration the hash decoder compares the length_of_key counter to check if the remaining hash key length is greater than 12. If the remaining length is less than or equal to 12, the intermediate key will be routed to the final addition block, which

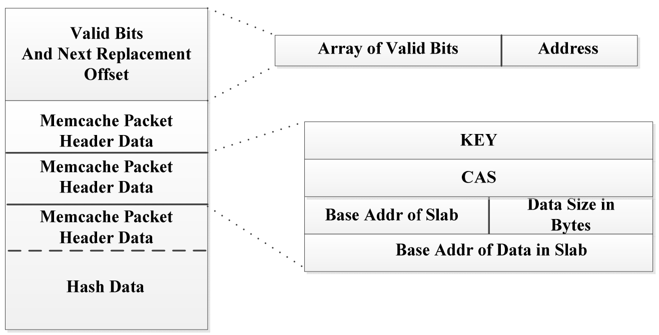

Valid Bits And Next Replacement Offset
Memcache Packet Header Data
Memcache Packet Header Data
Memcache Packet Header Data
Hash Data
Array of Valid Bits
Address
KEY
CAS
Base Addr of Slab
Data Size in Bytes
Base Addr of Data in Slab

Figure 5: Layout of hash bucket data structure

takes care of the combine functionality for key lengths less than or equal to 12. The internal variables a, b and c are then computed with the final addition/combine block. The variables are passed to the final mix data path to post process the internal states so that the final constant hash value is computed.

The main challenges of directly porting a pipelined version of the software algorithm are the throughput and resource utilization. The direct pipelined implementation of the software hash function to hardware resulted in 6 clock cycles (cc) of latency in the mix module, and 7 cc of latency in final mix state. Hence, for commonly seen key sizes of 60 bytes [5], the latency is about 34 cc, resulting in poor overall performance. Hence, a design space throughput exploration was performed to identify a design that best balanced both hash latency and overall module frequency. From the study we concluded the ideal hash decoder implementation has the mix block take 3 cc, and the final block take 4 cc of latency.

### *4.3.5 Memory Management*

The DRAM-based memory for the Memcached appliance is categorized into two memory sections, one as hash memory, and another as slab memory. The slabs accommodate variable sizes to store different sized user data. As explained in Section 4.3.3, in our system the controller creates slabs depending on user inputs, such as the number of slabs, and amount of bytes to allocate to the slabs. After the slab memory has been reserved, the remaining DRAM capacity is then assigned to be used as hash memory.

#### *4.3.5.1 Hash Memory Data Structure*

The layout for the hash data is shown in Figure 5. The hash table stores entries with only the data needed to provide fast lookups for key-value pairs, including the key, base address of value, and size of value. While the hash table entries include the key, they do not directly include the value data, which is instead handled by the slab memory management. To access the hash memory for read and write operations, we use the *hash read* and *hash write* sub-modules respectively.

The hash table consists of buckets which hold multiple hash entries; the buckets allow multiple items which map onto the same index to be present in the hash table. When a key is looked up in the hash table, the entries in the hash bucket are iterated over to compare each of their keys, and Compare and Swap (CAS) to the one being searched for. The 32 bit hash value is masked with the value (number of hash buckets – 1) to determine the final index value. Our appliance has a fixed number of entries (4) per hash bucket, unlike the software, which does not have a limit to hash bucket length.

The other two fields of slab address storage information, such as *base address of slab, data size stored in slab, and base address of*

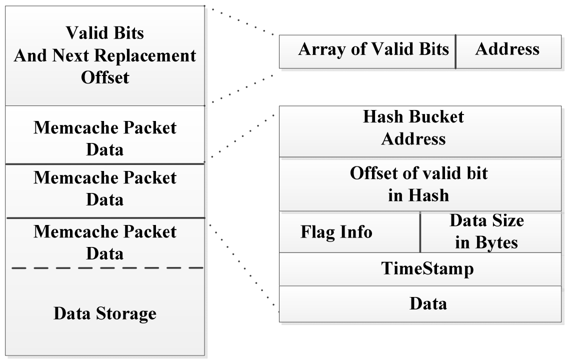


Figure 6: Layout of slab data structure

*data item in slab* are used for get or set operations. More detail on the use of these fields will be explained in the following sections on get and set operation. In addition to lists in the hash bucket, the *valid bit* field is used to assist in the data eviction policy when all lists in the bucket are utilized. Our implemented replacement policy will be explained in more depth in Section 4.3.5.3.

**Write Management:** The controller first signals to the hash write sub-module that the hash value is available. The hash write module reads valid bits field from the indexed bucket, starting at the *next replacement offset* to check for empty entries in the bucket. If there are no empty entries, a replacement policy is performed to evict a data item, and allocate the entry to the new data write request. After the offset for hash entry is determined, the hash write sub-module writes the key, and in parallel signals the slab write management module to start writing the data (stored in the data FIFO) into the allocated slab memory. Once the data is written, the slab write sub-module acknowledges the hash write with the base address of slab, base address of data stored, and size of data stored in slab. The hash write module uses this information to complete its write operation, and signals to the controller that the set operation is completed.

**Read Management:** The 32-bit hash values from different keys will potentially alias to the same hash bucket, which is why each bucket has multiple entries. Hence, the hash read sub-module is responsible for finding the entry with the matching key in the hash bucket (if one exists). Once the address to a hash bucket is available, the hash read sub-module reads the key information from individual entries sequentially to compare with the packet's key. If they match, the base slab address of data and size of data are passed to the slab read sub-module to read data from DRAM. However, if the keys do not match, a new data read request will be passed to the controller to access the next entry. If a miss occurs with all 4 entries stored in the hash bucket, the hash read sub-module signals the controller that it is a get miss.

#### *4.3.5.2 Slab Memory Data Structure*

The slabs are used to store the main Memcached packet information, including any *flags*, the *data size*, any *time stamps*, and the data itself. Keys from the command, and their corresponding slab base address information are only stored within the hash table memory. Similar to hash bucket entries, each slab data item stores *hash base address* information, and the *location of the entry in the hash bucket*. In addition to slab data, the slabs store *valid bit* information to assist in data replacement policy. There are *slab write* and *slab read* sub-modules that write and read data from slab memory, respectively.

**Write Management:** While the key is stored in a hash bucket entry, the hash write sub-module requests the appropriate slab base address given the associated data size. The slab write sub-module then reads the valid bits, and the *next replacement offset* to find an empty slot within the slab to store the input data. If none is found, data must be evicted from the slab using a replacement policy. Once a slot is available, the slab write sub-module starts writing the header information, and data stored in the header and data storage FIFO.

In the case where data was evicted, the slab write sub-module also clears the data's corresponding valid bit in its hash bucket, indicating that the item has been evicted.

**Read Management:** The slab read sub-module is responsible for reading data from the slab slot after the hash read sub-module transfers the base address of the data and offset. The data retrieved from the slab slot is stored into a response FIFO which the response packet generator uses to generate a get response packet.

#### *4.3.5.3 Replacement policy*

As Memcached provides caching of data, it must eventually evict items as its capacity becomes full. As described above, there are two scenarios where the capacity is full: either the hash bucket has no free entries, or the slab class has no free entries. In both cases we use round robin replacement to select the entry to evict. While an LRU policy would potentially provide greater eviction accuracy (the software uses an LRU policy), it is difficult to implement efficiently in hardware for the multi-GB DRAM sizes used in the appliance. Hence, we use pseudo-round robin replacement policy where the next data item after the latest written data item is considered for replacement when all the data items in hash or slab are full. To augment round robin and make it more effective, we also keep track of the valid bits within the hash buckets and slab classes. These valid bits are first checked to see if there are any entries that are invalid (for example, they have expired or they have been deleted). Any invalid entries are used first before evicting currently valid entries.

When items are evicted from either a hash bucket or a slab class, their corresponding slab or hash entry must be evicted as well. Thus each hash entry has the base address of the slab entry, and each slab entry has a pointer to the hash entry. These are used to invalidate the corresponding entries.

#### *4.3.5.4 Metadata storage requirements*

Our design requires a few extra bytes of metadata not required in the original design. These include a 32 byte array of valid bits per hash bucket. Per slab, there is a 32 byte counter, and a variable number of valid bits based on the size of the elements in the slab. Assuming a 128 MB slab, this metadata overhead is about 256KB. There is some additional data required for storing addresses for the hash bucket and hash offset, as well as slab base address, for slab entries and hash entries respectively. We utilize unused bits in the data words that store the CAS value and flag values to store these addresses, avoiding additional metadata overhead.

## 5. EVALUATION

### 5.1 Methodology

To test our proposed FPGA Memcached appliance, we mapped the design on a DE-4 development board [13] with an Altera Stratix IV 530 FPGA. The board is interfaced with two 4GB DDR2 memory modules, four 1GbE ports, and standalone power supply of 12V, and 3.3V. The Stratix IV 530 FPGA used on the DE-4 development board has total resources of 424K logic cells, and 21Mb of memory to map user functionality. While 8 GB of total memory may be considered small for a Memcached server, this capacity was due to the limitations of readily available

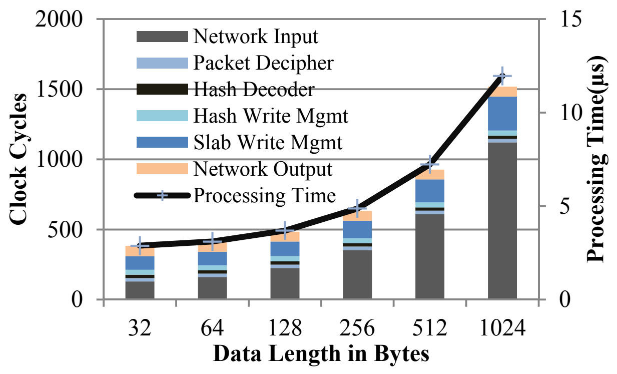


Figure 7: Set Operation Clock Cycle Distribution

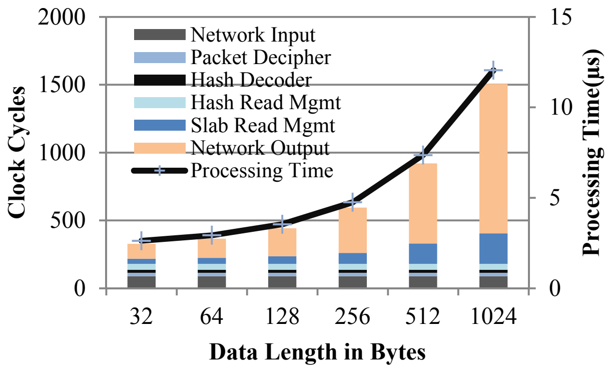


Figure 8: Get operation clock cycle distribution

memory controller IP, which has limited addressing range, and the development board design, which only supports SODIMMs. We anticipate newer FPGA boards to support larger capacities that would be more amenable for Memcached servers.

We implemented our Memcached appliance using a holistic approach of integrating our design with other Altera IPs. The Altera IPs used in our implementation are the Altera Triple Speed Ethernet IP to communicate with 1GbE port, and the DDR2 memory controller to interface with SODIMM memory modules. After the design is programmed onto the FPGA, an Altera system console is used to program the appliance's IP and load slab memory registers prior to use.

We test our appliance using a microbenchmark that uses the binary protocol format [11], and generates packets with different data sizes. The various data sizes we tested are 10, 32, 64, 128, 512, and 1024 bytes. However, we kept the key length constant at 64 bytes. The microbenchmark generates a stream of commands (get, set). The initial accesses are interleaved to populate the cache, but otherwise are streams of only one operation.

## 5.2 Detailed Analysis

### *5.2.1 Hardware Resource Utilization*

The current appliance design operates at 125MHz, and has a total resource utilization of 7% of the logic, and 16% of the memory bit resources on the DE-4's Stratix IV 530 FPGA (including all of the Altera cores). A more detailed illustration of resource utilization for individual components is shown in Table 1. While the application uses a large amount of memory, its logic requirements are not very high, as it is composed mostly of state machines. Most of the logic utilized by the design is for the Altera cores.

The basic Memcached design ($FPGA_1$) only occupies a small fraction of the total FPGA resources. Thus we mapped a second instance of the Memcached appliance ($FPGA_2$) onto the FPGA, interfaced to the second memory controller, SODIMM, and ethernet port. Thus each instance is independent. While it reduces the memory capacity per instance, it doubles the throughput; as each instance will cache a smaller range of keys, we do not expect it to negatively affect the hit rate. We used the dual Memcached mapping on the FPGA to analyze the throughput impact due to multiple instances being mapped. Even with two Memcached appliances operating on the single FPGA, the operational frequency of both designs is still 125 MHz. Table 1 also shows the utilization results on the Stratix IV FPGA.

### *5.2.2 Memcached application analysis*

We measured the performance of our proposed system on the FPGA board by running multiple test cases of varying user data size for the Memcached requests. Our results are shown in Figure 7 and Figure 8. We used counters on the FPGA to keep track of the number of clock cycles that each module of the appliance contributes to overall operation execution time. It can be seen from the latency figures that for the FPGA Memcached appliance, a majority of the time for a Memcached operation is spent on the network side and memory read/write operations.

**Set Operation:** The cycle breakdown of set operations is shown in Figure **7**. As set operations must read in a packet with key and value data, the network input module responsible for transferring the packet to the Memcached application takes a significant amount of clock cycles. The network input time grows as data size is larger, growing at an almost linear rate from 512 B to 1024 B. The primary reason for the high network input clock cycles with larger data payloads is because the UDP OE we used has an 8-bit I/O interface. The rest of the appliance interfaces are 256-bit, thus data must be converted before being transferred from the OE to the rest of the appliance, resulting in sub-optimal performance. We expect that greater network bandwidths and a wider OE interface the network performance would greatly improve performance.

After the packet parser concatenates the packet, the Memcached packet decipher module is responsible to distinguish the Memcached packets, and store the data into intermediate FIFOs. Hence the packet decipher latency also varies with input packet size. Although the hash decoder is designed to function on variable length keys, we only send 64 byte keys, so the hash decoder latency is constant at 23 clock cycles.

The time spent in slab write management also varies with respect to input data size. As the size of the data increases the slab write sub-module must spend more clock cycles writing the data to DRAM. In contrast, the hash write management consumes a very small fraction of total execution time as the amount of data written is constant. After the data is written into memory, a response packet will be passed back to the client to confirm data write operation. Since the response packet consists of a basic Memcached header field, the network output module accounts for

Table 1: Hardware Resource Utilization

| | Total Memcached Instances | | UDP OE | Packet Parser | Memcached Application |
|---|---|---|---|---|---|
| | One | Two | | | |
| Comb. ALU's | 21,044 | 42,478 | 1,501 | 154 | 8,593 |
| Logic Reg. | 22,517 | 45,164 | 1,342 | 238 | 6,447 |
| Mem (b) | 3.2M | 6.4M | 20,400 | 0 | 2.22M |
| DSP | 7 | 14 | 0 | 0 | 7 |

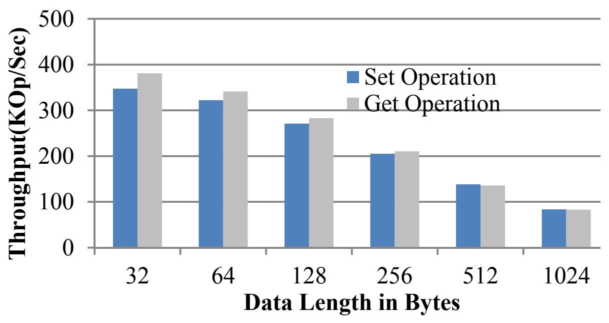


Figure 9: FPGA Memcached Performance

only a small portion of the total set operation clock cycles.

Overall we see that the FPGA performance is influenced by data sizes, and performs best with smaller data. As Memcached is often used with very small objects [5], we expect the appliance to fit well with potential use cases.

**Get Operation:** The get operation packet consists of a Memcached header field with a key to look up data in the cache. Therefore the size of the input packet is far smaller than in set operations. Unlike the set operation, the network input module only consumes a small fraction of total get operation execution time. Other modules that are responsible for only a small number of cycles are the hash decoder, and hash read management modules. The hash read management is only used to compare input key with key stored in memory and consumes only 42 clock cycles to read 64 byte key from memory, and compare it with input command key. During the actual data memory reads, the slab read sub-module consumes more clock cycles, and also increases as the data size read from memory increases. .

Similar to slab read management, the execution time of the network output module responsible to transmit data from the Memcached appliance to the client system increases by increasing the data size of the request.

## 5.3 Overall Appliance Performance

### *5.3.1 Throughput*

Based on the clock cycles required for the get and set operations, we calculated the overall throughput of our appliance performing either gets or sets. The results are shown in Figure 9. We see that at 128B data sizes, the appliance achieves nearly 270KOps/s for gets and sets, which is comparable to baseline server results that we obtained. As the data size increases we see a drop off in performance due to the reasons described in the previous section. Based on our results, we found that having larger width interfaces to the OE would significantly benefit performance.

### *5.3.2 Power and Energy Efficiency*

We measured the power consumption of our appliance through a current clamp on the input 12V power supply to accurately measure the power by deducting any AC/DC conversion losses.

Table 3: Power, Performance and Energy Efficiency Comparison

| System | Power Consum. (W) | Perf. (KOps/Sec) | Perf./W (KOps/Sec/W) |
|---|---|---|---|
| $CPU_1$ | 202 | 1000 | 4.95 |
| $CPU_2$ | 202 | 300 | 1.49 |
| $FPGA_1$ | 17.4 | 283 | 16.26 |
| $FPGA_2$ | 18.84 | 566 | 30.04 |

Table 2: Total Cost of Ownership comparison for 1000 CPU servers

| | **$CPU_1$** | **$CPU_2$** | **$FPGA_{2A}$** | **$FPGA_{2B}$** | **$FPGA_{3A}$** | **$FPGA_{3B}$** |
|---|---|---|---|---|---|---|
| **Space (T$/Y)** | 456 | 456 | 456 | 456 | 456 | 456 |
| **Power & Cooling (T$/Y)** | 2033 | 2033 | 593 | 296 | 593 | 296 |
| **Infrastructure Cost (M$/Y)** | 38 | 38 | 51 | 76 | 51 | 152 |
| **Total(M$/Y)** | 40.5 | 40.5 | 52 | 77 | 52 | 153 |
| **Perf/$ (Ops/s/$)** | **24.69** | **7.40** | **43.51** | **14.79** | **88.62** | **29.92** |
| **Perf (Mops/s)** | 1000 | 300 | 2262 | 2262 | 4608 | 4608 |

As shown in Table 3, we found that the whole appliance consumed only 17.4W ($FPGA_1$). When we mapped another Memcached appliance onto the FPGA, the power consumption only increased to 18.8W ($FPGA_2$), indicating there is significant benefit to maximizing the per-FPGA resource utilization.

Using the measured power consumption, along with the throughput achieved for get operations of 128B data sizes, we calculated the overall performance-per-Watt efficiency of our single and dual Memcached appliance designs. We also compared to two baseline servers: one that achieves 300K Ops/s ($CPU_2$), and one with optimistic performance of 1M Ops/s ($CPU_1$). In both cases we used a server with power based on our baseline server, which was a dual-socket Xeon, but reduced to 8 GB of memory to match the FPGA. Versus both of the baseline servers we see a 3.2X to 10.9X improvement in energy efficiency for the single Memcached appliance, and a 6.1X to 20.2X improvement for the dual Memcached appliance. The specialized implementation and low-power nature of the FPGA help to dramatically reduce the power consumption for our Memcached appliance.

## 5.4 Data Center Total Cost of Ownership

To compare performance and total cost of ownership (TCO), we used a cost model [14] that reflects monthly cost of operating a data center. We use similar cost factors considered by [15] to calculate TCO of a data center running a tier of 1000 Memcached servers. We assumed the baseline servers each had 64 GB of memory, and that the datacenter operator wanted to achieve that specific total cluster memory capacity.

Given that goal, we considered a cluster made up of the previous baseline servers $CPU_1$ and $CPU_2$. We also considered two different FPGA designs. One ($FPGA_2$), is based on the dual Memcached appliance design from before, with 16 GB of RAM, and using the performance achieved in the previous section. The second ($FPGA_3$) is a forward-looking higher-end design that also uses two Memcached appliances, but has 32 GB of RAM, and uses performance based on estimates with the OE I/O width bottlenecks removed. Each FPGA had two price points, A, which costs $1000, and B, which costs $3000. We expect higher end FPGAs to have costs closer to B, while lower end FPGAs or ones leveraging commodity pricing may cost closer to A.

Based on these systems, we calculated the overall TCO costs as space, power and cooling, and infrastructure costs, and estimated the overall Performance-per-TCO-$ (Ops/s/TCO-$) of the entire cluster. We see that a low cost FPGA ($FPGA_{2A}$) is able to provide substantially higher perf/$ than $CPU_2$, due in part to its low cost, its low power, and relatively high performance. Even a high cost FPGA ($FPAG_{2B}$) can slightly improve perf/$. A higher performing, but more expensive solution ($FPGA_{3B}$), continues to show benefits even against optimistic server designs ($CPU_1$).

## 6. FUTURE WORK

Based on our analysis, we found that the performance is constrained through the network I/O interfaces for 10GbE Ethernet connections. The UDP OE design used has an I/O interface of 8-bit whose bandwidth saturates just over 1Gbps. Also, the Memcached appliance is designed to operate with 256-bit memory I/O and 32-bit memcached packet parsing; hence, glue logic is required to convert the network I/O interface from 8-bit to 32-bit, and 256-bit words for parser/response and memory I/O respectively, resulting in extra latency that increases with packet size. Modifications to the current UDP OE or implementing an efficient network OE with wider data width that could utilize the full potential of the 32-bit Altera 10GbE IP core would leverage next generation network connections (e.g. 10GbE).

Lastly, although the UDP protocol is used for get operations in some Memcached clusters, due to lack of reliable packet delivery, TCP is typically used for set operations. To allow our system to directly operate in conjunction to contemporary CPU based systems, we would like to incorporate TCP OE functionality in parallel to an optimized UDP OE.

## 7. RELATED WORK

Several related works have explored alternative architectures for Memcached servers. Work by Berezecki et al. [9] used a many-core Tilera processor to run Memcached, and they proposed software modifications to optimize the software to run on multiple cores. Their architecture improved performance, and they also noted that improved network processing was a significant reason for the advantages. In comparison, our design focuses on a specialized, low-power, highly integrated architecture.

Work by Hetherington et al. [1] explores using GPUs to perform certain portions of the Memcached software. They implement get operation lookups on a GPU, taking advantage of the GPU's parallelism to operate on multiple requests simultaneously. They find that GPUs can provide performance improvements, but a key limitation is moving data between the CPU and GPU. In contrast, our work addresses the significant network-related bottlenecks, and supports both get and set operations.

Work by Jose et al. [16] explores using high performance Infiniband RDMA fabrics for Memcached. They rearchitect the Memcached software to use their modified interface and RDMA transfers. They are able to significantly improve performance, also confirming our conclusions that network processing latency is the primary performance bottleneck. Our work addresses the whole Memcached stack, offering significantly lower power along with a high performance, standard Ethernet interface.

Other work by Atikoglu et al. [5] has examined and characterized a large scale Memcached deployment at Facebook. Their work serves as a motivation for much of our work, identifying key trends in the sizes of keys and data, and overall cluster usage.

## 8. CONCLUSION

The continued growth in web services that need to provide low-latency access to large amounts of data, in turn, motivate improved system designs for in-memory distributed data caches such as memcached. In this paper, we argue that the I/O-intensive nature of such workloads, requiring fast communication and DRAM data access, better matches accelerator-based systems compared to currently-used traditional general-purpose servers.

We propose an FPGA-based implementation of a Memcached appliance that tightly integrates networking and memory access with compute for improved performance and energy efficiency. Our design addresses important challenges specific to the memcached workload such as high network rates, variable-length keys, and dynamic memory allocation, and includes algorithms adapted to hardware-efficient algorithms. We prototyped our architecture on an Altera DE-4 development board. Our design consumes 9% of the power of a corresponding traditional baseline server, and provides significant energy efficiency improvements (3.2X to 10.9X). Additionally, at a data-center level, our approach enables significant total-cost-of-ownership improvements compared to traditional designs. Overall, our results argue for future systems for Memcached servers to be accelerators based, but also point to the broad promise of accelerator-based solutions in the emerging era of data-centric computing.

## 9. REFERENCES

[1] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. 2012. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE Computer Society, Washington, DC, USA, 88-98.

[2] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R.Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.

[3] J. Sobel. Keeping Up (The Facebook Blog). http://blog.facebook.com/blog.php?post=7899307130, 2011.

[4] A. Arsikere. 2011. Building a scalable game server. http://code.zynga.com/2011/07/building-ascalable-game-server/

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, USA, 53-64.

[6] Low latency RPC in RAMCloud, forum.stanford.edu/events/.../2011plenaryRosenblum.pdf

[7] Memcached Commands http://code.google.com/p/memcached/wiki/NewCommands

[8] Scaling memcached at Facebook, http://www.facebook.com/note.php?note_id=39391378919

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. 2011. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*. IEEE Computer Society, Washington, DC, USA, 1-8.

[10] UDP Offload Engine, http://www.opencores.org

[11] Memcached Command Binary Protocol. http://code.google.com/p/memcached/wiki/BinaryProtocolRevamped

[12] Memcached hash function. http://burtleburtle.net/bob/hash/doobs.html

[13] Terasic DE-4 Development board, www.terasic.com

[14] C. Patel and A. Shah. 2005. Cost model for planning, development and operation of a data center. *Hewlett-Packard Laboratories Technical Report*, 2005.

[15] S. Chalamalasetti, M. Margala, W. Vanderbauwhede, M. Wright, and P. Ranganathan. 2012. Evaluating FPGA-acceleration for real-time unstructured search. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE Computer Society, Washington, DC, USA, 200-209.

[16] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. 2011. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 743-752.