Hardware Acceleration of Key-Value Stores

Howard Mao
zhemao@eecs.berkeley.edu

Sagar Karandikar
skarandikar@berkeley.edu

Albert Ou
aou@eecs.berkeley.edu

Soumya Basu
soumyab@berkeley.edu

***Abstract*—In-memory key-value stores are an important part of many datacenter applications. Web applications frequently use such software to cache the results of frequently recurring computations. Reducing the latency of key-value lookups will therefore go a long way towards improving total request latency. As a significant portion of the processing time for a request is due to the OS and application overhead, we believe bypassing the software layer and servicing these requests from a hardware accelerator will deliver significant benefits. In addition, previous studies have determined that the nature of key-value store workloads makes it feasible to implement such an accelerator as a small SRAM cache, instead of serving responses from DRAM as previous accelerators have done. In this study, we implement a hardware accelerator for the Memcached key-value store on a Xilinx ZC706 development board and characterize its performance. Our initial evaluation with a realistic workload shows a 10x improvement in latency for 40% of requests without adding significant overhead to the remaining requests.**

I. Introduction

Key-Value stores are used in many important websites. For example, Dynamo is used at Amazon [1]; Redis is used at Github, Digg, and Blizzard Interactive [2]; and Memcached is used at Facebook, LinkedIn and Twitter [3], [4]. These applications store Key-Value pairs and commonly function as a cache for frequently recurring computations, such as complex SQL queries. Therefore, tuning the performance of these storage systems is essential for building efficient large-scale web services. Because latency is critical for these applications, many optimizations have been explored that aim to reduce the response latency of these systems.

When analyzing the performance of large kv stores, we see that there are many different sources of latency in a GET request, but not all contributions are equal. One study of a typical datacenter application, summarized in Figure 1, indicates that software at the endpoint accounts for 88% of the total request latency [5]. Thus, if we can improve the latency of a single node, we can make substantial improvements to the total request latency in a memcached cluster. This is the focus of this paper.

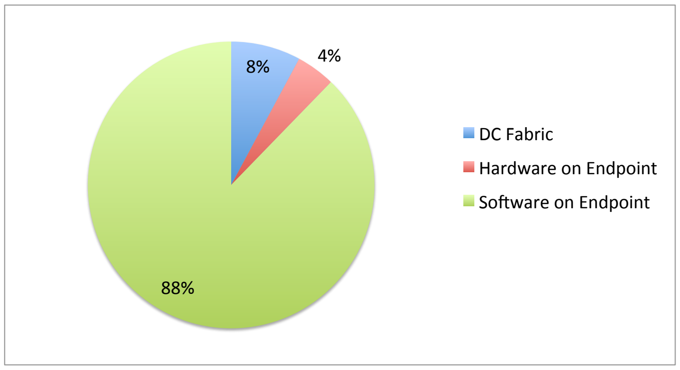

Fig. 1. Components of memcached request latency in a datacenter

We propose incorporating a dedicated cache on each node whose sole job is to serve memcached GET requests. This approach is effective due to the high skew that is present in the distribution of keys in realistic workloads, where a small number of keys make up most of the requests seen at a single node. GET requests are simple enough to handle with specialized hardware, allowing us to bypass the software networking stack entirely. Our preliminary evaluation shows that we reduce latency by a factor of ten for keys served from the accelerator when compared with a software implementation on the same board.

II. Related Work

Others have advocated and explored reducing the latency of datacenter applications, including memcached, by modifying the software stack [6], [7]. However, our scheme aims to remove the software altogether in the common case.

Additionally, previous efforts have accelerated memcached using hardware. However, these approaches either limit the functionality of memcached by attempting to implement all memcached features in hardware [8] or require expensive accesses to DRAM for each key that the accelerator serves, increasing latency [9]. We aim to design a system that minimizes latency for a small number of popular keys, while handling the remaining keys in software. Aside from some values being returned at extremly low latencies, an external observer should not be able to distinguish our accelerated FPGA-based memcached nodes from memcached running on standard server-class hardware.

III. System Overview

Our system consists of two novel components that enable the acceleration of memcached GET requests: a traffic manager and a key-value store accelerator, both written in Chisel, a hardware construction language developed at Berkeley [10]. These components are attached to a RISC-V Rocket Core with DMA-based networking support. The traffic manager directs incoming network traffic to the correct component of our system. Incoming UDP packets that represent GET requests in the memcached binary protocol are taken out of the receive stream and handed to the key-value store accelerator for processing. All other packets are handed off to the DMA engine attached to Rocket for standard software handling.

When a memcached packet is handed to the accelerator, the accelerator checks for the presence of the requested key-value

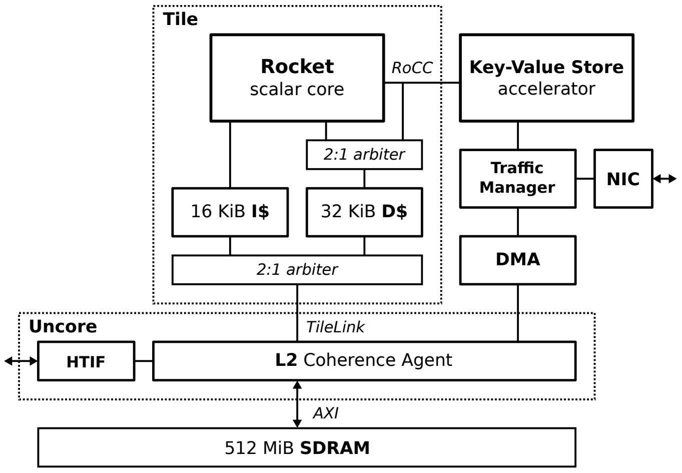

Fig. 2. Full System Design

pair in its SRAM. If the pair is cached in the accelerator, a memcached binary protocol GET response is constructed and transmitted without involving the application processor. If the pair is not found in the accelerator, the packet is forwarded to the DMA engine for transmission to software on the Rocket Core. Once a memcached request is handed to Rocket for processing, it is handled as on any other system running memcached.

The cache replacement policy used by our system to determine the set of keys placed on the accelerator is implemented entirely in software. Thus, it can be tuned without making significant modifications to the hardware. Additionally, the accelerator is sufficiently general that it can be used with any system that benefits from handling some requests directly at the network interface card. On the other hand, generalizing the traffic manager is left as future work. One avenue is to replace the traffic manager with a fully-programmable I/O coprocessor, allowing on-the-fly selection of packet filtering policies.

IV. Hardware

A. Key-Value Store Accelerator

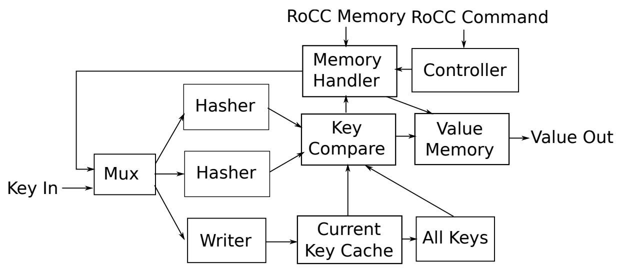

Fig. 3. Key-Value Store Accelerator

1) Lookup Pipeline: The main part of the accelerator is a lookup pipeline, which takes keys in through an input port, looks up the value, and streams the value out through an output port. Each port consists of two decoupled interfaces with ready-valid signals. The first decoupled interface sends the length of the data to come, and the second sends the data itself eight bits at a time.

When a key comes into the accelerator, the first thing done is to compute a primary and secondary hash value. The hash algorithm used is the Pearson hashing algorithm, a non-cryptographic hashing algorithm implemented as follows.

```
h = array of size n
for j from 0 to n-1
    h[j] = T[(x[0] + j) & 0xff]
    for i from 1 to length(x) - 1
        h[j] = T[h[j] ^ x[i]]
```

In this algorithm, h is one byte in the final hash value, x is the message, and T is a table containing a random permutation of all the integers from 0 to 255. The outer loop of the algorithm is parallelized by replicating the following hardware n times.

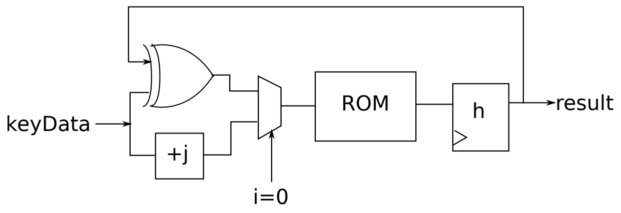

Fig. 4. Hasher

We use two different tables in order to generate two different hash values, a primary and a secondary hash. At the same time, we write the key into the current key memory. When the hasher is finished, it passes the two hashes to the key comparison unit.

The key comparison unit uses the hash value to index into the key memory. The key memory reserves 256 bytes for each key, so the starting address of a key in the memory can be found simply by taking the hash value and shifting it to the left by a certain number of bits. The shift amount depends on the word size of the memory, which can be parameterized. Assuming a word size of eight bits, the algorithm for the key comparison unit would be as follows.

```
function keycompare(hash, curKeyLen)
    cmpKeyLen := lenMem(hash)
    if (cmpKeyLen != curKeyLen)
        return false
    for i from 0 to (curKeyLen - 1)
        allInd = (hash << 8) | i
        if (curKeyMem(i) != allKeyMem(allInd))
            return false
    return true
```

Note that there is also a memory to store the lengths of keys. The key comparison unit will check the primary hash value and, if that does not match, the secondary hash value. If one of the two possible keys matches, the correct hash value is then passed to the value memory unit. The value memory will then look up the starting address and length of the value from a table, and begin streaming out the value through the output interface. If neither of the two keys match, the value memory unit will send a zero through the length interface to indicate that no value was found.

The interfaces between subsequent stages (hashing, key comparison, and value streaming) are constructed so that all

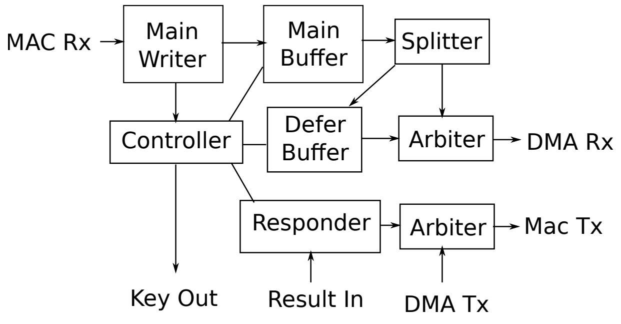

Fig. 5. Traffic Manager

	0	1	2	3
0	Request ID		Sequence Num	
4	# Datagrams		0x00	0x00
8	0x80	0x00	Key Length	
12	Zeros			
16	Total Body Length			
20	Zeros			
24				
28				
32 ...	Key			

Fig. 6. Memcached Binary Protocol GET Request

	0	1	2	3
0	Request ID		Sequence Num	
4	# Datagrams		0x00	0x00
8	0x81	0x00	0x0000	
12	0x04	Zeros		
16	Total Body Length			
20	Zeros			
24				
28	0x00000001			
32	0xdeadbeef			
36 ...	Value			

Fig. 7. Memcached Binary Protocol GET Response

three stages can operate simultaneously. Tags are used to allow external hardware to determine which response corresponds to which key. The one complication in decoupling the pipeline stages is that both the hasher/writer stage and the key comparison stage access the current key memory. This problem is solved by doubling the key memory. The writer writes to one half of the memory while the key comparison unit reads from the other half. When both units finish processing a key, they switch halves so that the key comparison unit reads the key which was just written and the writer overwrites the key which was just processed.

2) Control through RoCC interface: The accelerator is configured from software through the RoCC command interface. To place a key into the accelerator, the lookup pipeline must first be placed into write mode. In this mode, the hasher will no longer take keys from the traffic manager, but will instead take the key from the memory handler, which reads in the keys from DRAM through the RoCC memory interface.

Once the key is read in and hash values computed, the key compare unit will then determine which hash slot the key can be placed in. A key can be placed in a hash slot if the slot is empty, the key in the slot is the same as the key to be placed, or the key has a lower weight than the key being placed. The weight is simply a saturating count of how frequently the key is accessed. The count can be reset from software so that keys which were once popular but no longer are can be evicted.

Once the lookup pipeline has determined where the new key can be placed, the controller will instruct the memory handler to read the value through the RoCC memory interface and write it into the value memory.

B. Traffic Manager

Sitting between the key-value store accelerator and the NIC is the traffic manager, which routes network packets between the accelerator and the CPU.

Packets from the network card are first written into the main buffer. At the same time, the controller inspects the packet header to determine if the packet is an IPv4 UDP packet and that the payload is a Memcached binary GET request. A memcached GET request packet is broken into fields as shown in Figure 6.

The first eight octets are a pseudo-TCP header which is used by memcached to match requests and responses. We only ever use single-packet messages, so the sequence number is always zero and the number of datagrams is always one. The controller saves the request ID (as well as MAC addresses from the ethernet header, IP addresses from the IP header, and port numbers from the UDP header) to use in constructing the response packet. Bytes 8 - 31 are the memcached binary protocol headers. The first byte is a magic value 0x80, which indicates that the packet is a memcached request (as opposed to a response). The second byte is an opcode 0x00, indicating that it is a GET request. The other important fields are the key length and total body length, which should be the same. Bytes 32 and on are the key.

If the packet does not contain a GET request, the packet is sent on to the DMA engine, which will transfer the packet to the CPU. If it is a GET packet, the key is sent to the accelerator and the packet is moved from the main buffer to a defer buffer, allowing the traffic manager to process subsequent packets without waiting for the result to come back from the accelerator.

If a result does come back, the deferred packet is removed from the buffer and the responder unit constructs a response by adding the necessary headers and computing the IP and UDP checksums. The response is then sent back to the NIC. A memcached binary GET response should be as shown in Figure 7.

As before, the first eight bytes are a pseudo-TCP header. The request ID here should be the same as the request ID of the corresponding request. The binary protocol header starts with the magic number 0x81, indicating a response, followed by the opcode 0x00, indicating a response to a GET request. The response body begins with a 4-byte extras section containing

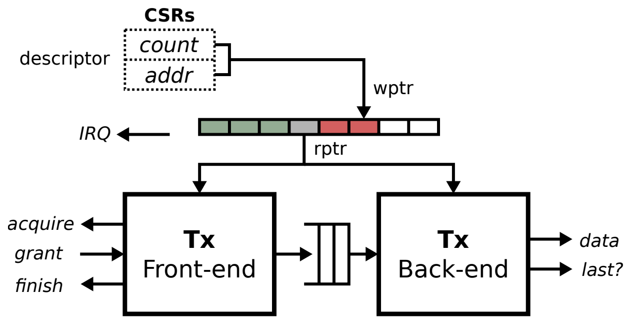

Fig. 8. DMA engine: Transmission channel

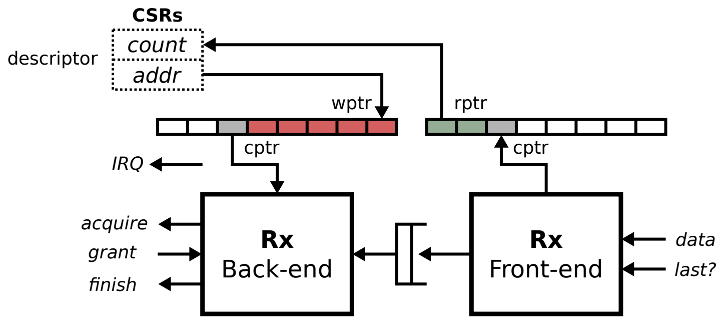

Fig. 9. DMA engine: Reception channel

the bytes DEADBEEF. The total body length should thus be set to the length of the value plus four. The rest of the body is the value.

If a zero-length result is returned, indicating that the key was not found in the accelerator, the deferred packet is sent on to the DMA engine.

The traffic manager also takes outbound packets from the DMA engine. An arbiter is used to interleave transmission of these packets with transmission of memcached response packets from the accelerator.

C. DMA Engine

Both the baseline and accelerated systems feature a direct memory access (DMA) engine for transferring Ethernet frames to and from main memory. A significant increase in I/O performance is realized by minimizing the involvement of the processor, exploiting a wider memory interface for improved throughput, and avoiding pollution of the inner caches. The DMA engine attaches directly to the media access controller of the NIC in the baseline, and alternatively to the traffic manager with the accelerator present.

The DMA engine comprises two simplex channels dedicated to receive (Rx) and transmit (Tx). A channel is internally structured as a pair of ingress and egress units, respectively designated the *front-end* and *back-end*, which coordinate by ready/valid handshaking to exchange blocks of data. Each end fully encapsulates the protocol-specific logic for its external interface and presents a generic FIFO abstraction at the other terminal, enabling units of different types to connect and interact in a consistent modular fashion. The design is thus intended to accommodate a variety of peripheral interfaces through interchangeable sets of front-ends and back-ends.

The memory units do not directly interface with the DRAM controller but instead communicate with the L2 coherence agent through the TileLink protocol, the primary on-chip system interconnect. This layer normally mediates access to the shared last-level cache, if present, or to main memory otherwise, as in this case. Closer integration with the cache hierarchy, although perhaps unconventional for peripheral I/O, simplifies cache coherence in a system-on-chip environment: The DMA engine is treated as simply another client like a processor tile. By marking DMA requests as uncached, the coherence agent ensures that all necessary cache flushes and invalidations occur on behalf of the DMA engine, which conveniently avoids interaction with coherence traffic.

Each TileLink transaction delivers a 64-byte cache line and supports access to the complete 32-bit physical address space, thus eliminating the need for bounce buffers. The exact widths of TileLink addresses and data are configurable system-level parameters.

One possibility briefly considered but left unpursued is adding an arbiter port to the L1 data cache for DMA operations. Since a TLB already resides in the L1 cache, this would naturally facilitate zero-copy transfers into user-mode virtual address spaces without requiring page fixing or pinning of physical memory. The eight-fold reduction in throughput would be somewhat compensated by a significant improvement in latency, possibly an acceptable trade-off given the narrow I/O width at the peripheral end. The overriding concern, however, is cache pollution by relatively large Ethernet frames.

1) Transmit: Figure 8 illustrates the Tx channel architecture. The descriptor ring lists pending buffers to transmit. To initiate a transfer, the processor enqueues a buffer descriptor with the base address and count, which the controller then inserts into ring at the slot denoted by an automatically incremented write pointer. A read pointer indicates which buffer is being consumed by the channel.

Once the transfer completes, the controller raises an interrupt request. Independent of the DMA engine, the software maintains a pointer to the earliest descriptor yet to be acknowledged, initialized with the read pointer after reset. The interval from this pointer to the current read pointer, non-inclusive, identifies the set of buffers that may be safely deallocated.

A shallow two-entry queue decouples the front-end from the back-end. This enables the front-end to prefetch the next cache line simultaneously as the back-end streams the previous line. The front-end currently serializes TileLink transactions; although multiple outstanding requests could be pipelined as addresses can be generated immediately, managing the potential out-of-order arrival of responses costs extra buffer space and control logic. In practice, this optimization proves to be largely unneeded. After the penalty of initial load, the throughput provided by TileLink adequately saturates the back-end to completely hide the memory latency of 30 to 50 cycles.

2) Receive: The Rx channel, depicted in Figure 9, possesses an architecture broadly similar to its Tx counterpart. The descriptor ring holds the base addresses of empty buffers which have been allocated beforehand and await to be filled with incoming frames. A write pointer reports the next available slot and automatically advances as descriptors are enqueued by the

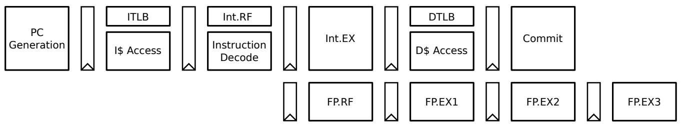

Fig. 10. RISC-V Rocket Core Pipeline Diagram

processor. For simplicity, the buffers are assumed to extend to the maximum length of an IEEE 802.3 Ethernet frame without the 802.1Q tag (1514 octets).

An internal current pointer tracks which buffer is presently active. Upon completion of a frame, the final count is stored in a separate ring at the slot indexed by the current pointer, while the controller raises an interrupt request. Counts are dequeued according to a read pointer, whose value can be used to associate each count with the original buffer. Note that the read pointer cannot overtake the current pointer, which itself cannot overtake the write pointer. Since channel activity stalls when the descriptor ring becomes empty, the processor must resupply buffers at a rate equal to consumption.

Due to the shorter latency of store operations, the Rx front-end and back-end do not need to be decoupled to the same degree as with the Tx channel. The back-end can perform the two-way TileLink acknowledgement (grant and finish) while the front-end populates the subsequent cache line.

The existing TileLink interface omits byte-granular write masks, thus complicating the handling of partial writes. Consequently, receive buffers must be aligned at a cache line boundary and, to prevent clobbering, cannot share cache lines with other data. As the kernel network stack also expects four-byte alignment of the IPv4 header, frames are written with a two-byte initial padding. These restrictions could be averted by resorting to read-modify-write operations for the first and last cache lines, but this would adversely impact latency and expose the DMA engine to coherence probes for only minor benefits in programming convenience.

V. Infrastructure

A. Pre-existing Components

Our system is built on a Xilinx ZC706 FPGA Evaluation platform. This board contains a Xilinx Zynq-7000 XC7Z045-2FFG900C AP SoC, with two ARM Cortex-A9 MPCore application processors attached to programmable logic. Our full-system design runs on the programmable logic, with the ARM cores used only for bootstrapping and non-network I/O purposes. In order to create an ethernet interface accessible to the programmable logic, we take advantage of the ZC706's SFP cage with a Brocade 1 Gbit Copper SFP Transceiver.

As our application processor, we use the open-source 64-bit RISC-V Rocket Core. The Rocket core is a 6-stage, single-issue, in-order pipeline running at 50 MHz on the FPGA fabric. The Rocket pipeline is elaborated in Figure 10. This core is supported by Rocket-Chip, which contains uncore components such as caches and coherence agents, along with the host-machine interface (HTIF) [11]. On our ZC706 board, Rocket has 512 MiB of DRAM, a 16 KiB instruction cache, and a 32 KiB data cache, while the HTIF bus supports block devices, a console, and externally initiated DMA transfers into DRAM.

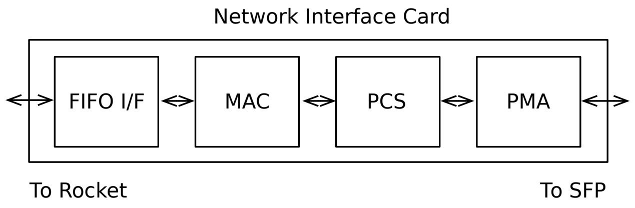

Fig. 11. ZC706-Compatible NIC from Xilinx IP

B. Custom Components

For our base system, we used the standard, publicly-available distribution of the RISC-V Rocket Core for Xilinx Zynq FPGAs. As of the start of our project, this distribution contained only a Rocket Core on the FPGA fabric, without I/O peripherals. As a result, during our bringup phase we built many components from scratch that other projects take for granted. We hope that the presence of these components on the open-source RISC-V platform will accelerate future research efforts. The following sections describe our experiences bringing up various hardware and software components that support our accelerator.

1) SFP Bring-up: In order to give the programmable logic access to a network, we use the small form-factor pluggable (SFP) cage on the ZC706 board to house a one gigabit SFP transceiver. Although the ZC706 contains an existing Ethernet PHY, it is tied to the ARM cores on the Zynq-7000 SoC, and attaching it to the programmable logic is not feasible without introducing significant latency overheads.

The SFP requires a low-jitter 125 MHz clock for operation. This is provided by an on-board Si5324 jitter attenuator chip. This chip does not default to 125 MHz and instead must be programmed over I^2C [12]. Since the I^2C bus is exposed only to the ARM cores on the Zynq SoC, bringup for this clock is the responsibilty of a Linux driver running on the ARM core.

2) Building the Network Interface Card: Our system uses Xilinx IP included in Vivado 2014.2 in order to construct a 1Gbps Network Interface Card. The first of these components is the LogiCORE IP Ethernet 1000Base-X PCS/PMA or SGMII core [13]. The PCS/PMA core implements the 1000Base-X Physical Coding Sublayer and Physical Medium Attachment standards, as described in IEEE 802.3-2008. The core integrates a device specific transceiver that is compatible with our Brocade SFP Module. On the client side, the PCS/PMA core provides an MDIO interface for control and a GMII interface for data.

The second component of the NIC is the Xilinx LogiCORE IP Tri-Mode Ethernet MAC [14]. The MAC implements the Ethernet Medium Access Controller protocol as defined in the IEEE 802.3-2008 specification. The MAC handles ethernet framing protocols and error detection. It attaches to the PCS/PMA IP through a GMII interface for data and an MDIO interface for control. On the CPU-side, the core exposes AXI4-Stream interfaces for transmit and receive and an AXI4-Lite Interface for management [15]. Both of these interfaces attach to custom hardware interfaces added to the Rocket core.

Resource	w/o A+TM	w/A+TM
Slice LUTs	17.09%	21.79%
Slice Registers	6.18%	8.01%
Memory	21.65%	63.85%

Fig. 12. ZC706 System Utilization

C. Bootstrapping the System

1) ARM Core: When power is first supplied to the board, our full-system design, containing Rocket along with the traffic manager, accelerator, DMA engine, and NIC is pushed onto the programmable logic in the Zynq SoC. Next, a copy of Linux with support for SFP clock bringup is booted on the ARM cores in the Zynq SoC. The `fesvr-zynq` application then executes on the ARM core. This program copies a specified RISC-V binary into the Rocket Core's DRAM over the HTIF bus and resets the Rocket Core. In our case, we supply a `vmlinux` binary containing the RISC-V Linux kernel, along with a root filesystem.

2) Rocket Core: From this point forward, the Rocket Core executes independently of the ARM core. The ARM core now handles only non-network I/O for the Rocket core over the HTIF bus. Upon boot, the Rocket core uses our custom TEMAC driver to bring up the NIC, using custom control registers for configuration of the NIC and DMA engine.

D. Future Infrastructure Work

Although not relevant to system correctness, we do not adhere to the RISC-V vision for I/O as defined in the upcoming privileged specification. For reference, the resource utilization of our full system is noted in Figure 12.

VI. Software

A. Programming Model

The key-value store accelerator is controlled through the RISC-V RoCC co-processor interface. This interface allows the CPU to interact with the accelerator by sending it custom instructions. A RoCC instruction is encoded as described in Figure 13.

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7
roccinst[6:0]	src2	src1				dest	*custom-0/1/2/3*

Fig. 13. RISC-V RoCC Instruction Encoding

We use only the `custom0` instruction, so the opcode is always `0001011`. The rd, rs1, and rs2 fields refer to the address of the destination and first and second operand registers, respectively. The xd, xs1, and xs2 bits are set if the register is actually used by the instruction, and clear otherwise. The funct7 section of the instruction is essentially a secondary opcode. Our accelerator specifies seven instructions, which are distinguished by the value of the funct7 field.

funct7	name	xd, xs1, xs2
0	Switch Mode	000
1	Delete Key	111
2	Reserve Key	111
3	Associate Address	011
4	Associate Length	011
5	Write Value	011
6	Reset Counts	000

The "switch mode" instruction changes the accelerator to either write mode or read mode, depending on the value of the rs1 field. Read mode is rs1=0; write mode is rs1=1. The accelerator must be in read mode in order to serve responses to the traffic manager, but must be in write mode in order to run any of the other instructions.

The "delete key" instruction removes a key-value association from the accelerator. For this instruction, the first source register contains the starting address of the key in memory, and the second source register contains the length of the key. The hash value that the key was deleted from is placed in the destination register. If the key was not present in the accelerator, the special value `0xffffff` is placed in the destination register.

The "reserve key" instruction adds a key to the accelerator. As in the "delete key" instruction, the first source register is the starting address of the key. The second register serves a dual function. The lowest eight bits of the register value are treated as the length of the key (keys are limited to 255 bytes in size), and the six bits above are treated as the weight. The reserve key instruction will replace an existing key if the weight provided is greater than the access count stored for that key in the accelerator. The hash value at which the key was placed is returned in the destination register. If the accelerator cannot find a place to put the key, the special value 0xffffff is placed in the destination register.

The "associate address" instruction associates an address in the value RAM with a hash value. The hash value is placed in the first source register and the address is placed in the second source register.

The "associate length" instruction associates the length of a value with a hash value. The first source register holds the hash value and the second source register holds the length.

The "write value" instruction transfers a value from the processor's memory to the accelerator's value memory. The first source register holds the hash value and the second source register holds the starting address in the CPU memory. The starting destination address and length must be set earlier using the 'associate address" and "associate length" instructions. The accelerator makes no attempt to stop one value from overwriting another. Software running on the CPU performs slab allocation of the value SRAM to ensure that the values do not overlap.

The "reset counts" instructions reset the access counts for all hash values to zero. The access counts are saturating 6-bit counters. Every time the key at a given hash value is accessed, the counter is incremented (unless it is already saturated). Resetting the counts to zero every once in a while ensures that old keys which were once popular but no longer are can be evicted from the accelerator.

To set a key on the accelerator and then activate the accelerator, we would write a program like the following pseudocode.

```
function setKey(key, value, weight)
    // set to write mode
    setMode(1)
    len_weight = len(key) | (weight << 8)
    hash = reserveKey(key, len_weight)
    if hash == 0xffffff then
        raise exception
    // allocate some space on the value SRAM
    // (done in software)
    addr = sramAlloc(len(value))
    // write the value to the accelerator
    assocAddr(hash, addr)
    assocLen(hash, len(value))
    writeVal(hash, value)
    // switch back to read mode
    setMode(0)
```

B. *Cache Policy*

In our system, the software decides which keys to place in the accelerator. There are many constraints that this decision must satisfy. First of all, the key itself should popular in the present and future and not just based on past requests. Another more important constaint is that the decision must be made *quickly*. Otherwise, cache handling will become the bottleneck. Lastly, the software can only push keys to the accelerator every so often, since the accelerator cannot serve requests while in write mode.

In order to find the hot keys and make quick decisions, we randomly sample each key with probability $\frac{1}{8}$. This allows us, on average, to get keys that appear frequently while ignoring many of the keys that only get called a few times. Since the distribution of requests has a long tail, the sampling mitigates any effects that this tail might have.

Since we would like to avoid setting the accelerator to write mode too often, we batch key pushes. This means that we only push keys after we accumulate 100 distinct keys.

The accelerator stores keys as a two-way set-associative cache. Thus, there might be a collision between the key being inserted and a key already set in the cache. To determine whether to evict the previously set key or reject the new key, the accelerator compares the count stored on the accelerator with the count passed in through the reserve key instruction. After each batch insertion, the software resets the hardware counts in order to avoid looking at outdated information.

VII. Evaluation

Our preliminary evaluation demonstrates that this approach shows a lot of promise. However, investigation into better replacement policies will likely yield improvements in system performance.

A. *Methodology*

In our experimental setup, we generated key-value pairs, placed them in memcached, and performed a GET request for each key. We then measured the latencies for a series of GET requests generated from a predetermined probability distribution. These distributions include one checking only a single key, a uniform distribution, a normal distribution, a Pareto distribution, and a distribution based on the ETC workload from a Facebook study [16].

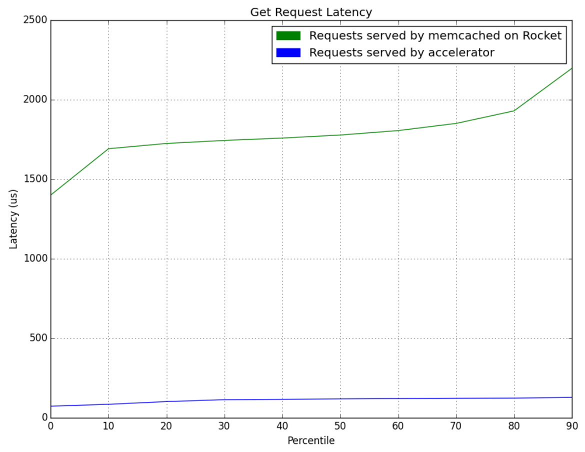

Fig. 14. Latency of GET requests when only getting 1 key

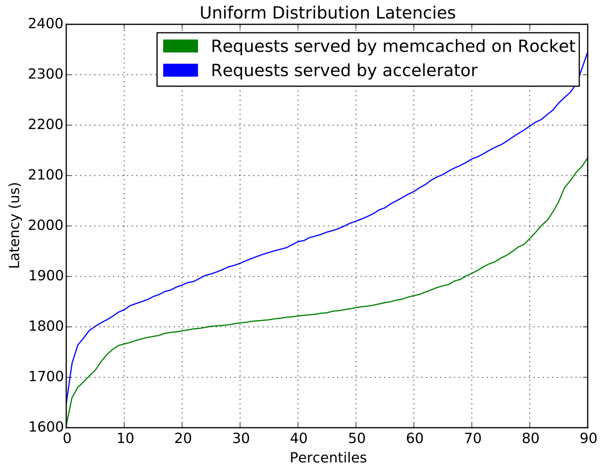

Fig. 15. Latency of GET requests when keys follow a uniform distribution

B. *Results*

In our first benchmark, we compared the latency of our accelerator to the latency of software memcached in responding to requests for a single key. For this experiment, we set a single key on the accelerator and in the unmodified memcached software. We then issued repeated GET requests with small random delays in between and recorded the latency of each request.

As we see from Figure 14, there is approximately an order of magnitude improvement between a request served by the accelerator and a request served by memcached software running on the CPU.

After the single-key benchmark, we tested a series of requests chosen according to a uniform distribution of keys.

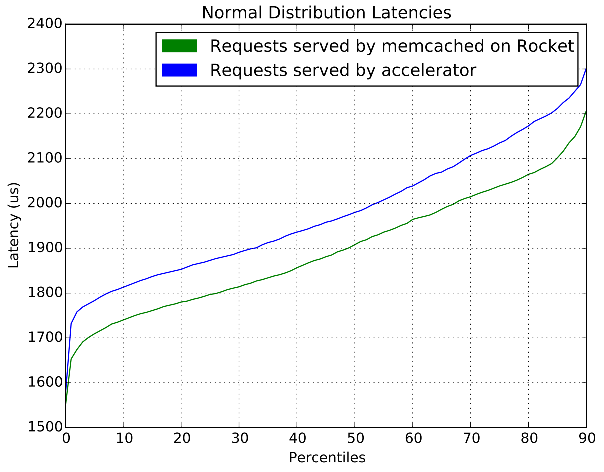

Fig. 16. Latency of GET requests when keys follow a normal distribution

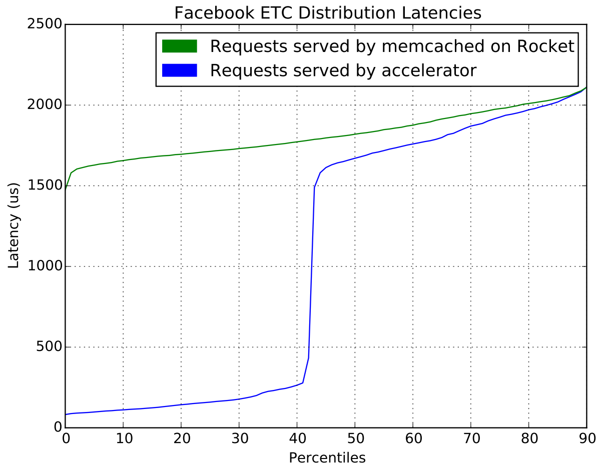

Fig. 18. Latency of GET requests when keys follow the Facebook ETC distribution

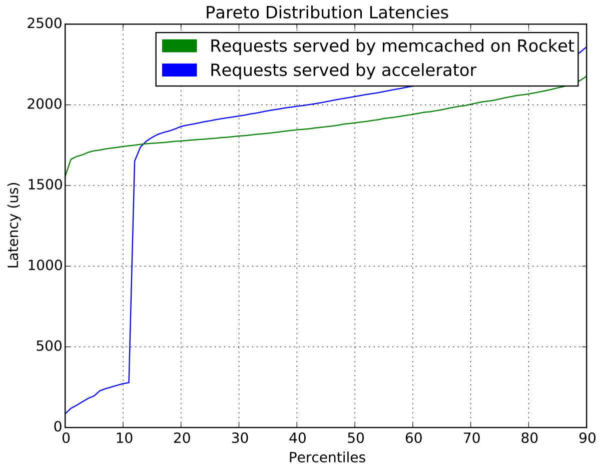

Fig. 17. Latency of GET requests when keys follow a pareto distribution

In Figure 15, we see that, for this distribution, the hardware-accelerated implementation performs worse than than the pure software implementation. The driving factor behind the poor performance is the overhead incurred in the traffic manager when checking whether each key is in the accelerator. Since all requests on the accelerated implementation must pay this penalty, we expect the accelerated implementation to perform poorly for key distributions that are not heavily skewed. The results for a normal distribution (Figure 16) show similarly poor performance for the accelerator compared to the software implementation.

For skewed distributions such as a Pareto distribution (Figure 17), we see a drastic improvement in the accelerated implementation over the pure-software implementation. In this test, the most popular keys are placed on the accelerator and are not easily evicted. For the these keys, the 10x latency improvement from serving from the accelerator more than makes up for the latency penalty incurred in the traffic manager. However, only about 11% of the requests benefit from acceleration. Requests for keys not placed on the accelerator still have higher latency than in the software implementation.

Finally, we used the Facebook ETC distribution [16] in order to see how our system would fare against a more realistic workload. In Figure 18, we see that this is a very promising start. 40 percent of all requests get a factor-of-ten improvement in latency. However, the poor performance in the non-skewed distributions suggests that improvements can be made in our caching policy to enhance system performance.

VIII. Future Work

Beyond the hardware being functionally complete, the next step will be to raise the throughput of the I/O subsystem closer to wire speed. The extensive buffering within the traffic manager suggests a natural widening of the stream interface servicing the DMA engine and accelerator. Other RTL optimizations can eliminate dead cycles in the state machines. The traffic manager, accelerator, and DMA engine could also be situated in a faster clock domain, ideally at 125 MHz alongside the MAC, rather than be constrained to the relatively slow core frequency of 50 MHz. These components are already decoupled from the rest of the design through ready/valid handshake schemes, which streamline the task of inserting asynchronous FIFOs and synchronizers at the crossings.

As for software, there is potential to improve the hit rate of the accelerator by tuning the cache replacement policy with a more accurate heavy hitters algorithm and possibly workload-aware speculation. Fragmentation of the value cache could become an issue with a larger dataset, which might be mitigated by more sophisticated allocation strategies than a simple linear scan.

In the long term, viability as a commodity datacenter appliance rests particularly on replacing fixed-function blocks with a programmable substrate, without compromising latency. Much greater efficiency is attainable with a VLSI implementation of the accelerator, but an appropriate degree of generalization and reusability is necessary to justify its expense.

The memcached-specific logic for classifying requests and constructing responses is isolated entirely within the traffic manager. By exchanging the traffic manager for a programmable I/O co-processor, it becomes possible to perform arbitrary packet filtering and to support more complex network processing, such as TCP offload. This also permits the accelerator to handle other varieties of key-value stores. Futhermore, examining a broader set of latency-sensitive applications for alternative uses of scratchpad memory attached to the NIC might lead to a more flexible design for the accelerator.

IX. Conclusion

In this paper, we described a hardware accelerator for the Memcached key-value store which we developed and evaluated on the ZC706 FPGA development board. By storing keys and values in a dedicated SRAM cache and serving responses directly to the NIC without involving the CPU, our accelerator delivers an order of magnitude improvement in latency. We furthermore showed that our accelerator can store a number of key-value pairs sufficient to serve 40% of the requests in a real-world workload.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, Stevenson, Washington, USA: ACM, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281.

[2] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: ACM, 2010, pp. 314–325. DOI: 10.1145/1815961.1816002.

[3] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004.

[4] J. Petrovic, "Using Memcached for Data Distribution in Industrial Environment," in *Proceedings of the Third International Conference on Systems*, ser. ICONS '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 368–372. DOI: 10.1109/ICONS.2008.51.

[5] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable Low Latency for Data Center Applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, San Jose, California: ACM, 2012, 9:1–9:14. DOI: 10.1145/2391229.2391238.

[6] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's Time for Low Latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13, Napa, California: USENIX Association, 2011, pp. 11–11.

[7] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL: USENIX, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[8] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached Appliance," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13, Monterey, California, USA: ACM, 2013, pp. 245–254. DOI: 10.1145/2435264.2435306.

[9] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: ACM, 2013, pp. 36–47. DOI: 10.1145/2485922.2485926.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012, pp. 1212–1221.

[11] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th*, 2014, pp. 199–202. DOI: 10.1109/ESSCIRC.2014.6942056.

[12] S. M. Srinivasa Attili Sunita Jain, "PS and PL Ethernet Performance and Jumbo Frame Support with PL Ethernet in the Zynq-7000 AP SoC," 2013. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1082-zynq-eth.pdf.

[13] "LogiCORE IP Ethernet 1000BASE-X PCS/PMA or SGMII v14.2," 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/gig_ethernet_pcs_pma/v14_2/pg047-gig-eth-pcs-pma.pdf.

[14] "LogiCORE IP Tri-Mode Ethernet MAC v8.2," 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v8_2/pg051-tri-mode-eth-mac.pdf.

[15] "Vivado Design Suite, AXI Reference Guide," 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "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*, ser. SIGMETRICS 2012, London, England, UK: ACM, 2012, pp. 53–64. DOI: 10.1145/2254756.2254766.