Achieving 10Gbps line-rate key-value stores with FPGAs

Michaela Blott, Kimon Karras, Ling Liu and Kees Vissers
Xilinx Inc.

Jeremia Bär, Zsolt István
ETH Zürich

Abstract

Distributed in-memory key-value stores such as memcached have become a critical middleware application within current web infrastructure. However, typical x86-based systems yield limited performance scalability and high power consumption as their architecture with its optimization for single thread performance is not well-matched towards the memory-intensive and parallel nature of this application. In this paper we present the design of a novel memcached architecture implemented on Field Programmable Gate Arrays (FPGAs) which is the first in literature to achieve 10Gbps line rate processing for all packet sizes. By transformation of the functionality into a dataflow architecture, the implementation can not only provide significant speed-up but also operate at a lower power consumption than any x86. More specifically, with our prototype we have measured an increase of up to a factor of 36x in requests per second per Watt that can be serviced in comparison to the best published numbers for regular servers with optimized software. Additionally, we show that through the tight integration of network interface, memory and compute, round trip latency can be reduced down to below 4.5 microseconds.

1 Introduction

Many well-known web-sites deploy distributed in-memory caches, which are implemented with standard DRAM on a large array of x86 servers, to reduce access load on databases, improving both performance and scalability of the site. As detailed in [7], [4] and discussed in Section 2, standard cloud servers provide sub-linear scalability to these kinds of scale-out workloads. For memcached, which is a well-known key-value store implementation, the best possible performance numbers are still substantially below the maximum packet rate of a 10Gbps Ethernet interface. Given memcached's streaming nature, mostly moving data between network and DRAM with little compute, we investigated a dataflow implementation which is typically more apt for these types of applications and commonly deployed within the networking context.

To prove the suitability of such an architecture and to get a full understanding of all potential limitations, we have built a memcached compliant prototype and analyzed it for throughput, latency and power consumption. For the implementation we utilized FPGAs which enable the implementation of customized integrated circuits through programming rather than designing and manufacturing custom chips. The circuit itself is designed as a custom-tailored pipeline that fully extracts the parallelism in the application. The prototype demonstrates full line-rate processing, handling up to 13 million of requests per second (RPS), while providing a round trip latency below 4.5 microseconds (us). Power consumption of the FPGA and its subsystem is around 50Watts (W), whereby the FPGA itself consumes less than 15W. With small exceptions, the majority of the functionality could be successfully incorporated as discussed in section 4.

The rest of this paper is structured as follows: Section 2 provides some background and reviews related work. Section 3 describes our architecture and implementation, while Section 4 presents the results of the implemented prototype. Finally, Section 5 summarizes our findings and points towards future work.

2 Background & Related Work

Memcached implementations are almost exclusively deployed using x86 servers at their basis, although it is well-established that they are not optimized for this kind of workload, which in essence moves data between network and DRAM with little compute ([7],[4]). Low instruction- and memory-level parallelism in scale-out applications means that the large, 4-wide-issue super-

scalar core pipelines are often underutilized. Furthermore, the processor's last-level data cache, which consumes as much as half of the entire processor die area, becomes ineffective given the random-access nature and required memory size of the application, and with that causes considerable energy waste. Finally, throughput and latency are both heavily impacted by the high latency of the communication stack. Other bottlenecks such as lock mechanisms employed in the multi-threaded memcached architecture have meanwhile been successfully addressed. To the best of our knowledge, Wiggins and Langston provide in [13] the most fine-tuned implementation available in literature. In this work a performance of 3.15 MRPS is achieved with a median round-trip latency around 200us on a dual-socket Xeon E5 processor.

To overcome key limitations of the TCP/IP-related bottlenecks, Jose et all [10] investigated the use of RDMA-based communication on an Infiniband QDR network. Their work clearly illustrates that latency can be dramatically reduced to below 12microseconds and performance increased to roughly 1.8 MRPS for GET operations by using high-performance networking clusters. In contrast to this work, we address the bottlenecks by rearchitecting the server while maintaining compliance with the industry-standard memcached implementation.

Besides x86, several other approaches have been investigated to speed-up key-value stores. Berezecki et al. [4] utilize a 64-core Tilera processor to run memcached. The elimination of serializing bottlenecks on the Tilera processor and the allocation of different cores to different functions allow a single Tilera CPU to reach 0.335MRPS with a round trip latency ranging from 200-400us.

GPUs are particularly adept in accelerating massively parallel tasks, something leveraged in a combined CPU-GPU system by Hetherington et al [8]. The paper concludes that CPU/GPU hybrids outperform their respective counterparts by a factor ranging from 4 to 8 whereby moving data between CPU and GPU is the bottleneck.

Finally, Chalamalasetti et al. presented in [5] a 1Gbps memcached implementation using an FPGA. The chosen architecture, which is in essence a centralized design, achieves a throughput of 0.556MRPS with significantly reduced power consumption to the compared server implementations. Similarly to our approach, they utilize the tight proximity of network and DRAM to achieve lower latency and provide benefits in power consumption through customized architectures. However, the described architecture, which is mostly centralized around a controller, differs fundamentally from the dataflow architecture presented in this paper which can achieve significantly higher throughput by exploiting task and instruction level parallelism while reducing latency and providing scalability to higher rates. Additionally, we address in our work the problem of handling variable key sizes and support further protocols (ASCII and TCP) and operations.

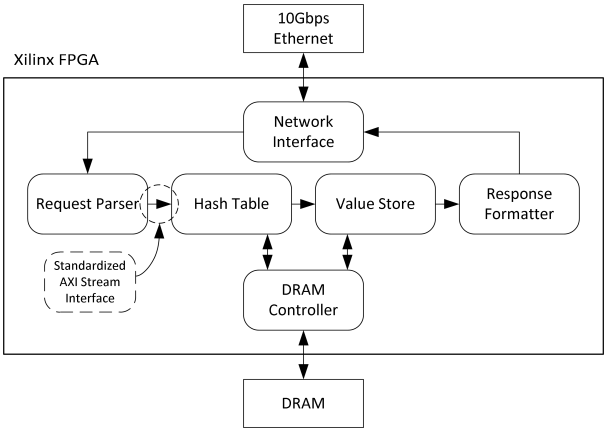

Figure 1: Memcached dataflow pipeline

3 FPGA-based Memcached Design

Our key objectives in designing an FPGA-based memcached server were threefold: 10Gbps line rate processing with a scalable architecture, minimal latency, and power efficiency. To achieve these design goals, we made specific architectural choices which are described in Section 3.1. Section 3.2 details the implementation with emphasis on the hash table.

3.1 Architecture

Fully pipelined dataflow architecture. As previously mentioned, a fully pipelined dataflow architecture is deployed. This applies to the macro-level architecture as shown in Figure 1, but also to the implementation within each individual pipeline stage. This architecture fully exploits task- and instruction-level parallelism to achieve high data rates for streaming applications [1].Within the networking domain, these architectures are well-known to cater for data rates of 100Gbps. Scalability in throughput is achieved by widening or duplication of the data path. Thereby, our system architecture is fundamentally designed to scale to higher rates. Besides throughput, other key advantages of this architecture include its cut-through operation which results in low latency. Furthermore, the customized nature of this circuit minimizes power consumption, where the benefit stems both from a lower power consumption per operation and a highly efficient memory architecture. Finally, as packets are being processed in fixed order through the various pipeline stages, arbitration conflicts on shared memory resources

are greatly simplified, and the memory arbitration overhead and need for locking mechanisms commonly seen in existing software solutions essentially evaporate.

Tightly integrated network and memory interface. As detailed in Section 2, key bottlenecks in existing x86 implementations are related to the network stack's processing overhead and its large latency which is the result of the indirect access to the network via PCIe®. On our chosen architecture, we minimize this distance between network interface, compute resources and memory by integrating both interfaces directly on the FPGA. The resulting tight coupling significantly reduces latency and removes any processing overhead that would be associated with transporting data between various system components such as network adapter and CPU.

Modular design through standardized interfaces. Providing a modular design is essential to future-proofing implementations and ensuring that additional functionality can be easily added at later stages. To achieve this, we deploy a modular approach whereby each stage in the pipeline provides identical input and output interface formats. These are based on the AXI-4 streaming protocol [1] and standardize how key, value, and meta-data, are conveyed between macro-level pipeline stages.

3.2 Implementation

As is illustrated in Figure 1, the dataflow architecture consists of five key processing stages, namely *network interface, request parser, hash table, value store* and *response formatter*. Packets are received on the board on a 10Gbps Ethernet interface and streamed back-to-back through these processing stages before being transmitted back into the network. The first stage, the *network interface*, handles all related processing to Ethernet and includes a full UDP and TCP offload engine for which we leveraged existing third-party IP[2]. Only the memcached requests themselves, bar all additional headers, are passed to the *request parser* together with a connection identifier. The *request parser* analyzes the memcached packets to extract key, value, and meta-data information and generates an opcode for the currently supported subset of operations. Currently only ASCII and binary protocols are supported, however further protocols can easily be added with no impact on performance. Independent of the incoming protocol, the *request parser* normalizes all packets to the format of the standard interface. This information is then passed to the *hash table*. The *hash table*'s responsibility is to produce an index into the *value store* for any incoming key. The *value store* simply supports read or write operations on a corresponding area of memory as defined by the opcode. For SET operations, the presented value is written into memory. In case of GET operations, the retrieved value is added to the packet information as it streams to the *response formatter*. Finally, the *response formatter* supports formatting of responses according to the supported protocols.

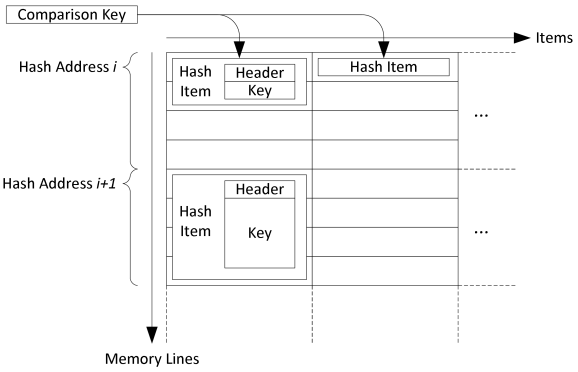

Figure 2: Hast table architecture

Key challenges in the implementation are support for flexible key sizes, collision handling, memory management and expiration handling. To support flexible key sizes, we stripe the keys in a hash table item over multiple DRAM locations as is shown in Figure 2 and then match the read access bandwidth of the *hash table* with the incoming packet bandwidth. This is essential to avoid stalling of the pipeline. Collision handling is in software solutions typically solved by chaining a flexible number of keys to a single hash table index. In hardware, collision handling is often supported through a parallel lookup [3] of a fixed number of keys that map to the same hash table index. This creates a limit for the maximum number of keys that can coexist for a given hash table index, often referred to as bucket size. In our implementation the supported bucket size is 8. As illustrated in Figure 2, the chosen approach resolves collisions only up to a certain degree. If, however unlikely, a further key maps to the same hash index, then the request simply fails and an appropriate response is generated. We believe this to be an acceptable behaviour for what is essentially equivalent to a cache full/miss. Both flexible key length and collision handling trade-off throughput and design complexity with memory density. The exact penalty depends on the actual key distribution for individual use cases, performance of the selected hash function and fill-level of the hash table and is with that hard to quantify. Further discussion on this topic can be found in [9].

Memory management can in theory be handled from

[1]The protocol specification can be found at: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html`

[2]These are provided to us by Maxeler Technologies.

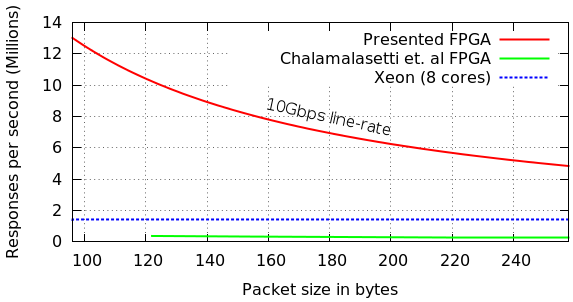

Figure 3: Performance of the FPGA for GET operations as a function of network packet size

within the FPGA, however this involves additional memory access bandwidth. Given our current platform limitations, we selected the host CPU to assist with this task. Free addresses are pushed to the *hash table* via PCIe®. Deleted or expired items are returned in a similar manner back to the host. We currently support three different value size categories, however further categories can easily be added to achieve better memory utilization. Finally, in regards to cache management, we implement a completely different approach to common software algorithms such as LRU [12]. These require linked list data structures which again demand significant memory access bandwidth. Instead, we opportunistically inspect items which are collocated at the same memory address when accessing the *hash table* for SET operations. Hash items with an expired time are then subsequently freed. As part of our future work, we will investigate other strategies that can achieve better cache utilization and move memory allocation inside the FPGA to work towards standalone appliances.

4 Results

In this section, we present the results of our memcached implementation with regards to throughput, latency and power and compare it to existing implementations. Our experimental set-up included the following components: The memcached server itself is implemented on a Xilinx®Virtex®6 FPGA board. The FPGA is directly connected to 24GBs of dedicated DDR3 memory and two 10Gbps Ethernet ports. The board is hosted in a PCIe®slot of a Maxeler workstation. For testing protocol compliance, we relied on standard software memcached clients (mcblaster, memaslap). For performance and latency testing, we used a Spirent C1 network testing appliance. Finally, to measure power consumption, we used a power meter at the wall plug,

Platform	RPS	Latency	RPS/W
TilePRO (64 cores)	0.34M	200-400 us	3.6K
TilePRO (4x64 cores)	1.34M	200-400 us	5.8K
Intel Xeon (single socket, 8 cores)	1.4M	200-300 us	7K
Chalamalasetti FPGA	0.27M	2.4-12 us	30.04K
FPGA (board only)	13.02M	3.5-4.5 us	254.8K
FPGA (with host)	13.02M	3.5-4.5 us	106.7K

Table 1: Comparison of our FPGA solution with current state-of-the-art ([4, 13, 5])

Xilinx's®power estimator tool (XPE) [3], and Maxeler's diagnostic tools which provide current readings for various power rails.

Performance. In our experimental set-up, we populated the FPGA with 1 million key-value pairs of constant size and issued binary GET requests for random keys at the maximum rate the 10Gbps Ethernet connection allowed. We repeated this experiment for different key sizes, ranging from 6 to 168 bytes which encompass typical use cases as described in [2]. To ensure a symmetric input and output data rate, values were always equal in size with the keys. As a result, the network packet size ranged from 96 to 258bytes. We used UDP with its minimal overhead to ensure maximum throughput over the network. Figure 3 shows that our system handles full 10Gbps Ethernet line-rate regardless of key and value size. Expressed in number of successfully served requests per second, we can handle as much as 13.02MRPS for small binary UDP encoded packets, while the number drops naturally as the packet size increases due to network saturation. For ASCII encoded messages, which have lower protocol overhead, the measured performance reaches 13.74MRPS. This significantly outperforms other known implementations in literature as illustrated in Table 1. For SET operations, we have measured full 10Gbps line-rate performance as well, which translates to 12.5MRPS. The slight discrepancy is purely caused by a slightly larger protocol overhead for SET operations. For mixed operations of GETs and SETs, we need to avoid read after write hazards when 2 or more operations on the same hash index reside within a small and critical part of the pipeline. In this rare event, the pipeline stalls selectively the conflicting GET operation while allowing other operations to overtake. The probability of this event is very small and depends on the specific use case, in particular on the size of the working set. This is further discussed in [9].

Round trip latency. With regards to round-trip time (RTT), we recorded between 3.5us and 4.5us depending on packet size. This is a two-order of magnitude improvement over standard x86 based approaches ([13],

[3] `http://www.xilinx.com/products/design_tools/logic_design/xpe.htm`

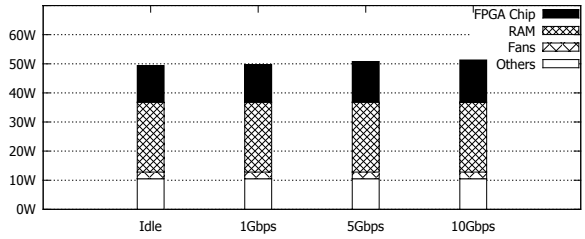

Figure 4: Power consumption of the FPGA board under increasing load

[4]) and significantly below the typical service level agreements which are around 1ms [13]. Similar latencies have been achieved only by the alternative FPGA implementation [5] as well as an Infiniband-based implementation [10] which eliminates the typical latency penalty associated with the TCP/IP communication stack. The low access latency within the server is of significant value when servicing GET requests with multiple keys in the argument as the overall latency equals to the worst case latency. Furthermore, on the application level, this might play an important role when data structures such as graphs reside within the key-value store that require multiple lookups with intermediate dependencies.

Power. We measured power consumption utilizing a wall-plug power meter, and correlated the numbers with diagnostic reports from Maxeler, estimates from XPE and various data sheets. In our experimental set-up, the cache was populated with 80byte keys and 250byte values, which were subsequently read via GET commands at different input rates (1, 5 and 10Gbps). As shown in Figure 4, the power consumption of the FPGA subsystem remains almost constant at around 50W with a workload variation of below 2W. The FPGA itself consumes 15W, the memory subsystem draws 24W and auxiliary infrastructure accounts for the rest[4]. Published x86 systems operate significantly less efficiently as can be seen from Table 1.

Resource utilization. Our current prototype requires roughly speaking 80k lookup tables and 63k flip-flops, both of which are an FPGA-specific measurement of the device's utilization. Expressed differently, this accounts for less than 26% of the available resources in the Virtex6®SX475T.

Limitations. The foremost limitation of the hardware accelerated design is the cost of increased development time and NRE given the low-level nature and complexity of traditional hardware design flow. Current advances in High Level Synthesis [6] are looking to offset this deficiency through new C-based programming flows. Other limitations are only temporary and platform related: Our current prototype requires light assistance from the host CPU (below 3% of 1 core for 10% SET operations) to handle memory management. In our case, the Sandy Bridge host CPU is clearly over-dimensioned for the given task. In a potential future product, an integrated SoC such as a next generation Xilinx®Zynq®[5] might very well be sufficient. The chosen simplified cache management algorithm may not result in the same cache effectiveness as an LRU could offer. Furthermore, we currently only support SET, GET, DELETE, and FLUSH operations. The third party TCP offload engine limits the number of concurrent sessions to 64. In our current work we are investigating how to scale to thousands of sessions. Finally our prototype is limited to 24GB DRAM which is a constraint of the chosen platform. With current and future devices, the density that can be potentially connected to an FPGA is significantly larger and can easily accomodate typical installations.

5 Conclusions & Future Work

In this paper, we presented a novel system architecture to implement in-memory key-value stores. By transforming the pthreaded software architecture into a customized data-flow pipeline, we can achieve consistent line rate processing at 10Gbps for any packet size. Furthermore, the FPGA-based implementation delivers a worst case round trip time of 4.5us and achieves an increase of 36x in RPS/W over the best published x86 numbers. With energy costs being estimated to contribute over 30% to a three-year total cost of ownership (TCO) [11], we expect TCO of an FPGA-based memcached appliance to be substantially smaller.

The key limitation of the hardware accelerated design is the cost of increased development time and NRE given the low-level nature and complexity of traditional hardware design flow. To address this issue, we have started an investigation into implementation of key functions, for which simple API support is insufficient, through new C-based programming flows using Vivado®HLS. High Level Synthesis tools show promising results within other application domains to reduce development, verification and performance optimization efforts [6]. As part of our ongoing research efforts, we plan to evaluate the effectiveness of these tools within this application context. Additionally, we believe that the chosen architecture can scale to higher performance points. As part of our future work, we will investigate the opportunities with current devices (e.g. Kintex®) at higher line rates, aiming at 80Gbps.

[4]These numbers do not include inefficiencies in the power supply.

[5]`http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/`

6 Acknowledgments

The authors would like to thank Oliver Pell, Rob Dimond and Andrew McCaffrey from Maxeler Technologies for their continued support in this project.

References

[1] ARVIND, AND NIKHIL, R. Executing a program on the mit tagged-token dataflow architecture. *Computers, IEEE Transactions on 39*, 3 (March 1990), 300–318.

[2] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev. 40*, 1 (June 2012), 53–64.

[3] BANDO, M., ARTAN, N., AND CHAO, H. Flashlook: 100-gbps hash-tuned route lookup architecture. In *High Performance Switching and Routing, 2009. HPSR 2009. International Conference on* (June 2009), pp. 1 –8.

[4] BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M., AND STEELE, K. Power and performance evaluation of memcached on the tilepro64 architecture. In *Green Computing Conference and Workshops (IGCC), 2011 International* (July 2011), pp. 1 – 8.

[5] CHALAMALASETTI, S. R., LIM, K., WRIGHT, M., AUYOUNG, A., RANGANATHAN, P., AND MARGALA, M. An fpga memcached appliance. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (New York, NY, USA, 2013), FPGA '13, ACM, pp. 245–254.

[6] CONG, J., LIU, B., NEUENDORFFER, S., NOGUERA, J., VISSERS, K., AND ZHANG, Z. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 30*, 4 (April 2011), 473 –491.

[7] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *SIGARCH Comput. Archit. News 40*, 1 (Mar. 2012), 37–48.

[8] HETHERINGTON, T. H., ROGERS, T. G., HSU, L., O'CONNOR, M., AND AAMODT, T. M. 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* (Washington, DC, USA, 2012), ISPASS '12, IEEE Computer Society, pp. 88–98.

[9] ISTVAN, Z. Hash Table for Large Key-Value Stores on FPGAs . Master's thesis, ETH Zurich, Dept. of Computer Science, Systems Group, Switzerland, 2013.

[10] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., UR RAHMAN, M. W., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance rdma capable interconnects. *2012 41st International Conference on Parallel Processing 0* (2011), 743–752.

[11] LANG, W., PATEL, J. M., AND SHANKAR, S. Wimpy node clusters: what about non-wimpy workloads? In *DaMoN* (2010), pp. 47–55.

[12] MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78 –117.

[13] WIGGINS, A., AND LANGSTON, J. Enhancing the scalability of memcached. In *Intel Software Network* (2012).