A FLEXIBLE HASH TABLE DESIGN FOR 10GBPS KEY-VALUE STORES ON FPGAS

Zsolt István, Gustavo Alonso

Systems Group
Dept. of Computer Science
ETH Zürich
{zistvan, alonso}@inf.ethz.ch

Michaela Blott, Kees Vissers

Xilinx Inc.
Dublin, Ireland; San Jose, CA

{michaela.blott, kees.vissers}@xilinx.com

ABSTRACT

Common web infrastructure relies on distributed main memory key-value stores to reduce access load on databases, thereby improving both performance and scalability of web sites. As standard cloud servers provide sub-linear scalability and reduced power efficiency to these kinds of scale-out workloads, we have investigated a novel dataflow architecture for key-value stores with the aid of FPGAs which can deliver consistent 10Gbps throughput.

In this paper, we present the design of a novel hash table which forms the centre piece of this dataflow architecture. The fully pipelined design can sustain consistent 10Gbps line-rate performance by deploying a concurrent mechanism to handle hash collisions. We address problems such as support for a broad range of key sizes without stalling the pipeline through careful matching of lookup time with packet reception time. Finally, the design is based on a scalable architecture that can be easily parametrized to work with different memory types operating at different access speeds and latencies.

We deployed this hash table in a memcached prototype to index 2 million entries in 24GBytes of external DDR3 DRAM while sustaining 13 million requests per second for UDP binary encoded memcached packets which is the maximum packet rate that can be achieved with memcached on a 10Gbps link.

1. INTRODUCTION

Data centres grow each day both in sheer size and functional diversity driven by the emergence of new, connected mobile computing devices, an explosion in the volume of amorphous data captured and exchanged, and the rising popularity of web services. In particular the databases that store the actual content of web sites face scalability issues to sustain the expected growth rate. To address this challenge, it has become standard practice to use distributed in-memory key-value stores, such as memcached[1], as a caching layer between web servers and databases [1]. The most popular contents can be retrieved directly from one of these caches, thereby alleviating the access load on the database. The clients, which are typically the web servers, communicate with the key-value store over a simple protocol, which builds on variations of two commands: *set* and *get*. These allow for the storage and retrieval of arbitrary binary data by requesting them through their corresponding key. *Get* operations typically predominate in the application [2]. In the following we refer to these commands as *write* and *read*.

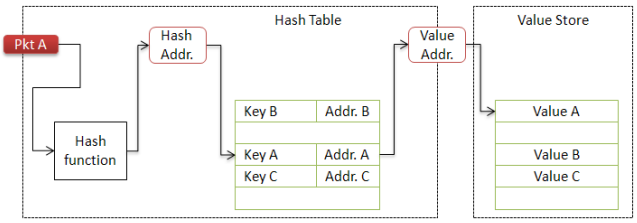

Fig. 1. Hash table and value store

As part of a broader research effort, we investigate a dataflow architecture for key-value stores that can sustain 10Gbps line-rate consistently and brings significant latency benefits through tight coupling of network interface, memory and compute resources [3]. We use FPGAs as implementation medium as the application demands a certain degree of programmability. This is to customize hash function, hash table parameters, communication protocols and memory and cache management strategies to individual use cases. At the heart of this key-value store is a hash table, which in essence determines the memory address of a value as a function of an incoming key. This is achieved by first applying the chosen hash function to the contents of a key to produce an address in the table. From this location then, a pointer to the address within the value storage area can be retrieved as is illustrated in Figure 1. By reducing a typically large input key space to a small index space, two or more different keys can potentially map to the same hash table address. Resolving these so-called hash collisions can in-

[1] http://www.memcached.org

978-1-4799-0004-6/13/$31.00

troduce uncertainty in regards to response time, which constitutes the main difficulty in achieving consistent line-rate performance. Additional challenges include supporting keys with a broad range of types and sizes while maximizing table occupancy and enabling a large storage space with millions of entries.

In this paper we present the chosen hash table architecture which sustains 10Gbps throughput independent of packet sizes through its pipelined structure. A novel parallel lookup technique is employed to handle flexible key sizes and collisions, while ensuring high memory utilization. The architecture of the hash table is parametrized and can in theory support millions of items. Our prototype implementation demonstrates 2 million entries with a maximum key size of 168Bytes. The hash table occupies less than 400MBytes which on our development platform leaves 23.6GBytes of DRAM for storing values. By using Bob Jenkins' lookup3 hash function [4], also present in the open source memcached software, we can demonstrate excellent hashing properties for different types of keys. Finally, the design can adjust to different memory access speeds and latencies thereby providing a certain amount of flexibility.

The rest of this paper is organized as follows: Section 2 discusses related work in hash tables on FPGAs. Section 3 provides the context of our work while Section 4 describes our dataflow architecture, detailing the implementation of the key modules. Prototype platform, experimental setup and results are presented in Section 5 and conclusions can be found in Section 6.

2. RELATED WORK

Hash tables provide an effective solution to the common search problem of retrieving a value based on a key. The main challenge presents itself in handling the case when multiple keys map to the same hash index (*hash collision*) while maintaining consistent throughput levels. This section looks at some of the most common techniques to deal with collisions in hardware. These stem mostly from the context of network processing, with one exception which relates to an alternative memcached implementation.

One well-known approach is referred to as *perfect hashing* which in essence relies on the idea of customizing the hash function itself such that for a previously known set of input keys, collisions can be completely avoided (e.g. [5]). The key benefit is that all read and write accesses to the hash table can be done in O(1) time. However, this scheme is not applicable in our use case as the incoming set of keys is dynamic and previously unknown.

Cuckoo hashing [6, 7] is a so called *open addressing* variant in which items are stored in one of two possible locations. Reads are always answered in constant time, however writes pay potentially the cost of collisions: If both locations are taken, a greedy algorithm is used to reorganize the table resulting in high response times. Although writes are much less common than read operations in typical memcached deployments, they can potentially block incoming read requests from accessing the hash table thereby jeopardizing line-rate performance. Consequently this approach becomes unsuitable given our design goals.

An alternative way of resolving collisions is through *chaining*: All keys that hash to the same index are stored in a dynamically allocated list, often referred to as *bucket*, which is linked to this index. The main advantage of chaining is that it provides symmetric read and write performance. Operations can be performed using a relatively constant number of memory accesses [6]. However, deviations from the median response time can be potentially too large which constitutes a challenge for pipelined hardware implementations. Often, the bucket size is limited to restrict the potential penalty. This approach is deployed in [8], which is the only other known memcached implementation on an FPGA. In contrast to our solution, Chalamalasetti et al use a centralized control architecture rather than our dataflow approach and aim for low power rather than performance. An important consequence of this architecture in regards to the hash implementation is that in case of a read, multiple locations have to be read and subsequently compared, yielding a much lower performance. (The presented prototype operated at 1Gbps line rate.) Furthermore, the chosen hash table does not support flexible key sizes.

As previously stated, chaining-inspired schemes are typically used in combination with an upper bound on the number of items that can map to the same hash index [9, 8], thereby all items in a bucket can be accessed in parallel to alleviate dependencies on memory latency. Our technique builds onto this approach whereby we extend it to support flexible key sizes, trading off storage efficiency for constant access time.

It is possible to reduce the expected number of items in buckets by deploying several hash functions and/or multiple possible locations for items [10, 11, 9]. We chose to rely on large bucket sizes, which equally reduces the probability of an overflow, to ensure a more uniform hash table utilization and simplify the circuit.

Sometimes, the above-mentioned techniques are combined with an *overflow buffer* which holds all items that cannot be stored because their respective buckets are full. Typically these are realized with small Content Addressable Memories (CAM) [9, 7]. In our case, however, the requirement to support flexible and potentially very large keys makes using CAMs as spill-over storage infeasible. As mentioned before, we simply support a larger bucket size per hash index. Since the key-value store acts in most common use cases as a cache for the database, read and write success do not have to be guaranteed. Typically, it is acceptable that

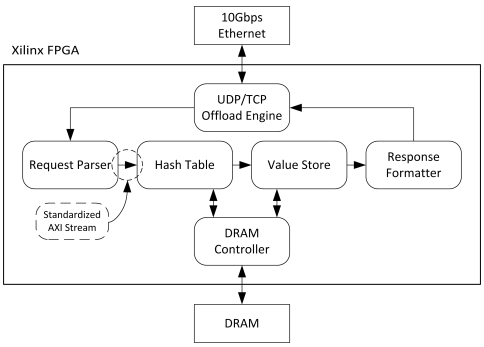

Fig. 2. FPGA-based memcached server architecture

for operations targeting buckets with too many collisions a cache miss or cache full is returned.

3. KEY-VALUE STORES IN FPGAS

As previously mentioned, this work is part of an ongoing research effort [3] to explore dataflow architectures for key-value stores with the aim of handling consistently 10Gbps throughput while reducing latency and power consumption. Currently, key-value store implementations are almost exclusively utilizing x86 servers at their basis, although it is well-established that they are not optimized for this kind of workload. High interrupt rates from the TCP/IP stack cause a high cycle per instruction (CPI) count. 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. To the best of our knowledge, Wiggins and Langston present in [12] the most fine-tuned implementation available in literature, demonstrating 3.15 million requests per second (MRPS) with a median round-trip latency around 200microseconds (us) on a dual-socket Xeon E5 processor.

In our efforts, we aim to show how through exploitation of instruction- and task-level parallelism in the dataflow architecture, the throughput can be significantly scaled and higher power efficiency achieved. Furthermore, through the tight integration of network, memory and compute we demonstrate that the latency can be reduced by two orders of magnitude. The proposed fully pipelined dataflow architecture is shown in Figure 2. In this design, packets are received from the network and processed by *Ethernet* and *TCP/IP* offload engines. Then they are parsed and passed to the *hash table*, which finds the location of a reserved memory slot as a function of the incoming key. The *value store* then reads or writes the respective values from the specified location in memory. Finally the *response formatter* creates a response packet and returns it to the *TCP/IP* offload engine. All blocks communicate over a standardized streaming interface to facilitate a modular design and an iterative implementation process. The hash table forms a fundamental part of this design and is described in more detail in the following section.

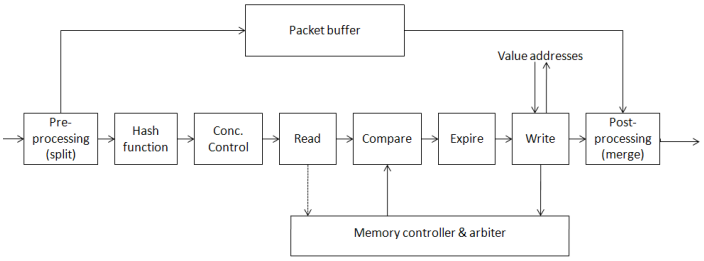

Fig. 3. Hash table pipeline structure

4. IMPLEMENTATION

The main design goal for the hash table implementation, as part of a key-value store, was to ensure line-rate processing for all operations. This presents challenges in regards to collision handling as was discussed in the previous sections. In addition, the fact that key-value stores need to handle keys and values of a large range of sizes and of different types had to be taken into consideration. Per memcached protocol specification, keys can be as large as 250Bytes, and values are allowed to reach 1MByte in size. Finally, we aimed to provide an implementation which is independent of memory latency and access bandwidth to achieve a certain degree of portability. In the following paragraphs we describe the key implementation choices which allowed us to fulfil these requirements.

4.1. Pipeline Structure

To achieve the aforementioned design goals, a pipelined architecture was chosen. As such, data enters and leaves the hash table through flow-control enabled streaming interfaces based on the AXI-Streaming standard[2]. Keys and values are transmitted over these interfaces in parallel in 64bit data buses. The meta-information associated with each request, such as operation code, key and value length, are conveyed over a third parallel channel.

As illustrated in Figure 3, the actual hash table logic sits between the *pre-processor* and the *post-processor*. The preprocessor extracts the key and relevant metadata from the stream and stores the rest of the packet in a packet buffer, and the post-processor merges the results from the hash table back into the packet. First, the *hash function* calculates

[2]Specification at: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html`

the address of the key in the hash table. The *concurrency control unit* then ensures that there are no read-after-write hazards in the pipeline by delaying conflicting keys. This is an artefact of handling multiple requests concurrently. The *read unit* issues the read commands to the memory and the *comparator* compares the input key with the data coming from memory. The *expiration unit* invalidates expired keys based on an internal counter which keeps time in seconds. Finally, the *write unit* is responsible for updating the hash table for all writing operations such as *inserts* and *deletes*. In addition, the write unit outputs the location of the value in memory and its length for successful operations, or an error code otherwise.

To achieve a decent memory usage efficiency, addresses are dynamically allocated for the value store in different block sizes. In our implementation, the external value store management logic communicates with the write unit through a simple queuing interface which provides free addresses for different block sizes in parallel. Deleted addresses are returned in a similar fashion to the external logic. Depending on the operation, the write unit fetches or pushes the value store addresses from their respective queues and updates the keys' data in the hash table accordingly.

4.2. Parallel Lookup with Flexible Keys

In the hash table we handle collisions with a variant of the *chaining* method explained earlier. A list of constant length is pre-allocated at each hash table address and for every incoming key, the entire list, or bucket, is retrieved from the table and compared to the respective key in parallel. We basically trade-off *probability of collisions versus memory bandwidth*. With increasing bucket size, the probability of collisions can be reduced. However, this is at the expense of memory bandwidth. In essence, the available memory bandwidth acts as an upper bound for the bucket size, because it increases linearly with the number of parallel items.

If every hash table location would correspond to only one memory burst access, in the following referred to as *memory line*, then the maximum key size would depend on the width of the memory interface. To support flexible key sizes, we stripe keys over multiple memory lines. Thereby, the retrieval and comparison of longer keys takes more memory accesses and with that a longer time. To ensure line-rate operation, we match the read access bandwidth with the incoming packet bandwidth. We guarantee that the time it takes to retrieve keys from memory is smaller than the amount of time it takes to transfer their corresponding packets over the network. This layout is shown in Figure 4. A bucket is spread over one memory line, whereby each hash item can span multiple lines. A hash item is composed of 1) a fixed size header, which contains the length of the key, its expiration time and the pointer to the value store with the value length, and 2) the key itself.

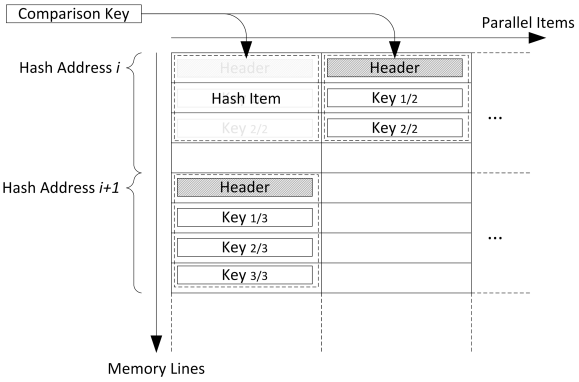

Fig. 4. Hash table layout in memory

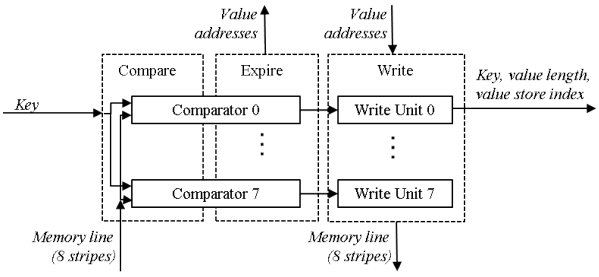

Fig. 5. Parallel operation of compare, expire and write units

Although, this method of striping the keys over a number of lines provides flexibility, it also leads to reduced storage efficiency for small keys. However, since we use DRAM for the hash table, we believe the density requirements to be less critical. With less than 400MByte we can support 2 million entries and 23.6GByte of storage.

The main use-case of the hash table is caching, therefore if a key is not found within a bucket, it is declared a miss and no further address lines are accessed. This way we can guarantee constant access time to the hash table regardless of its fill rate or contents, whereby we maximize the probability of a hit in the table by using a large bucket size, 8 to be specific. In the next subsection we show that with a good choice of hash function, and by not utilizing the hash table to its full capacity, the number of lost items can be kept at a minimum.

4.3. Flexible Handling of Keys and Memory

Implementing a hash table which offers flexibility both in the number of parallel items and in the size of the keys poses several challenges. The *compare-expire-write* stages of the pipeline need to be able to handle keys arriving over multiple clock cycles, which have to be matched in parallel to multiple keys that reside in a number of memory lines. On our platform a memory line is 384Bytes in total, which cor-

responds to a burst length of 8 on a 384bit wide memory interface. As illustrated in Figure 5, *compare* and *expire unit* are split into parallel components to support the processing of multiple hash items in parallel. The input key is then broadcast to all *comparators* which merge the compare and expire logic. The memory line is equally divided into the various hash items whereby each one is routed to a different comparator. We refer to the part of a memory line that belongs to one hash item in the following as a *stripe*. This way the copies of the key can be compared concurrently to the stripes of the memory. Each comparator produces three result bits which represent whether the stripe matches the input key, whether it holds an expired key, or whether it is free. These result bits are forwarded to the *write unit*.

Similarly to compare and expire, the write unit is split into *stripe writers*. For read operations, the stripe writer outputs the header of the key which contains the value address and value length in bytes. This is then merged back into the original packet. For write operations, the first stripe writer with a free slot fetches a value store pointer from one of the address queues, and then writes the key and its header to the actual hash table residing in memory. For delete operations the pointer to the value is pushed into the deleted address queue, and the hash table entry is erased by validating the corresponding flag in the hash table. The value pointers of expired keys are handled in the same fashion. When performing any write or delete operations, all addresses belonging to expired keys within the accessed line are pushed into the deleted queue, and the hash table line is updated with freed entries.

4.4. Hashing

We chose to implement the Bob Jenkins' hash function lookup3 [4] which is also used in the open source software version of memcached and is well-known to work effectively over a broad range of key types. This function processes variable sized keys iteratively in 96bit chunks and each chunk is split into three 32bit numbers which are added to a set of state variables. Before the next chunk is read, these state variables are mixed using addition, subtraction and XOR operations. Due to the inherent feedback loop, the hash function cannot be easily pipelined. Our implementation can consume a new 96bit word every 6 clock cycles. To achieve line-rate throughput, 64bits have to be processed every cycle. Since keys need to be padded to multiples of 96bits, six copies of this function are deployed in parallel inside the *hash unit*, and a round-robin scheme is utilized both for distributing the keys and for collecting the hash values.

We evaluated the software version of the hash function on synthetic keys based on the most common memcached use cases described in [2]. The aim was to see to what level the hash table could be filled before the hash function produces too many collisions. Figure 6 shows that by increasing

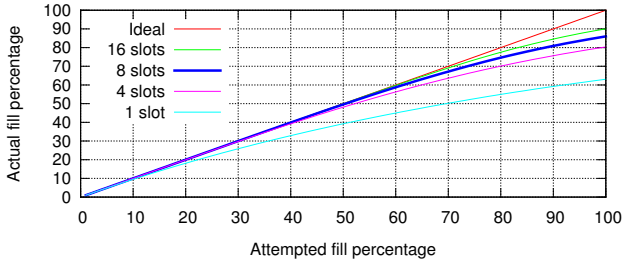

Fig. 6. Percentage of successfully stored items in the hash table as function of parallel lookup for the lookup3 function

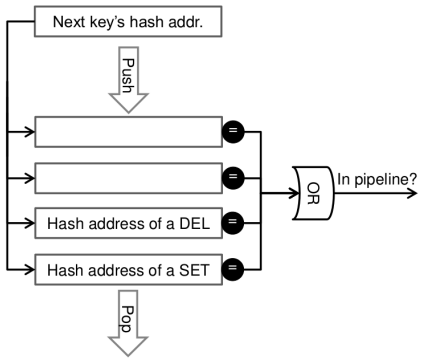

Fig. 7. Concurrency control unit

the number of parallel items per address line, the effective utilization of the hash table can be greatly increased. For instance, with 8 parallel slots per memory line, the hash table can be filled to 50% while keeping the percentage of lost items well below 1%. As previously stated, maximizing this parallelism makes sense, however the available memory bandwidth defines the limits.

4.5. Address Conflicts and Concurrency Control

In order to protect against read-after-write hazards in the pipeline we adopted an approach in which keys are blocked before the read unit while a writing operation on the corresponding hash address resides within one of the subsequent pipeline stages. The previously mentioned concurrency control unit is in place between the hash and read units to handle these conflicts.

For the implementation, we use a queue-like structure as shown in Figure 7. Write and delete operations push their addresses into this data structure when entering the read phase, and pop their addresses upon leaving the write phase. For each new incoming read operation, the latest hash address is compared to all hash addresses that reside within this queue structure to eliminate potential hazards. The pipeline before the concurrency control unit is temporarily stalled in case of a match. The number of keys which can

concurrently reside within the pipeline's critical section depends on the latency of the memory. Therefore, we have parametrized this component such that it can be easily adjusted to match different access latencies of different memory types on different platforms. On our platform, for instance, the average memory latency allows for a maximum of 50 keys in this section. Of course, in real-world workloads most of these would be read operations, making read-after-write conflicts rare. Furthermore, the pipeline is overprovisioned and a buffer introduced to minimize the effect of temporarily blocking the entrance to the next pipeline stage. In the evaluation, we show that already for a minimal working set of 500 out of 1 million entries with typical read-write distributions [2], the effects are not observable.

4.6. Memory Independence

The hash table implementation described in this paper can be adjusted to support different types of memories and memory interfaces. As described in the previous section, the compare-expire-write units can be adjusted to match the available memory bandwidth and interface width by splitting memory lines into a parametrizable number parts which are then handled in parallel. Also, the concurrency control unit is adjustable to memory latency. Furthermore, the design is parametrized to support different maximum key sizes. This does not require modifications to any of the pipeline stages apart from the buffers. With that, we achieve an independence of the memory interface and type, which creates a certain degree of portability.

4.7. Memory Layout

Our chosen platform offers one single DDR3 DRAM interface that has to be shared between the value store and the hash table. Fundamentally, this implies almost a complete random access pattern in address sequences resulting in a relatively poor access performance. This is further aggravated by the fact that the DRAM needs to be read and written to update value store and hash table depending on the given nature of the received operation, thereby introducing inefficiencies due to the associated data bus turnaround. The only possible optimizations were ensuring that hash items and values reside within a single row and value store and hash table do not interrupt each other while operating within a suitable access sequence. The average data bus utilization was measured to be around 20%.

5. EVALUATION

5.1. Platform and Experimental Setup

The hash table has been implemented on an FPGA-based network adapter. The heart of the platform is a Xilinx® Virtex6® SX475T chip, which is interconnected to a 10Gbps Ethernet interface and 24GBs of DDR3 SDRAM (Figure 8). The memory is accessed in 384bit words at 300MHz with a burst size of 8. The FPGA board sits in a Maxeler workstation with an Intel i7 quad-core processor and 16GBs of memory. The FPGA and the host communicate through PCIe® gen2 x8. The hash table was evaluated both in simulation and in hardware as part of our memcached prototype, whereby we relied on a Spirent C-1 network tester appliance for performance testing. Performance numbers present were measured with the UDP-based binary memcached protocol.

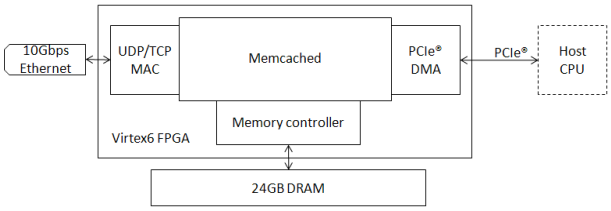

Fig. 8. Memcached deployed on a Maxeler Workstation

For experiments we deployed a hash table with a capacity of 2 million items and a maximum key size of 168Bytes. This key size was chosen so that every hash table entry spans at most 8 memory lines. The table described above occupied less than 400MBytes in DRAM, and was used to address the remaining 23.6GBytes of memory allocated for the value store.

5.2. Performance

This section shows that the hash table can meet the line-rate requirement regardless of key size for both read and write operations. Given our limit for maximum key size, we issued read and write commands to random addresses with key sizes ranging from 6 to 168Bytes in simulation with a value size of 1Byte. Similarly, we ran hardware experiments in which we measured the maximum throughput of the hash table as part of the memcached prototype. Requests were sent over UDP in the binary memcached format, yielding packet sizes between 96 and 258Bytes. Given the minimal packet size of 96Bytes, a maximum packet rate of 13MRPS can be achieved, then the network is fully saturated. As shown in Figure 9, the hash table on its own is overprovisioned and can handle throughput beyond 10Gbps, servicing a maximum packet rate of over 31MRPS. This overprovisioning helps to accommodate for additional overhead associated with resolving address conflicts.

Figure 9 also illustrates the average latency introduced by the hash table on our evaluation platform. This number is composed of a constant offset and a component which increases linearly with the key size. The constant part accounts for roughly 90 cycles of which 60 are a direct result of the memory subsystem. Given a clock period of 6.4ns,

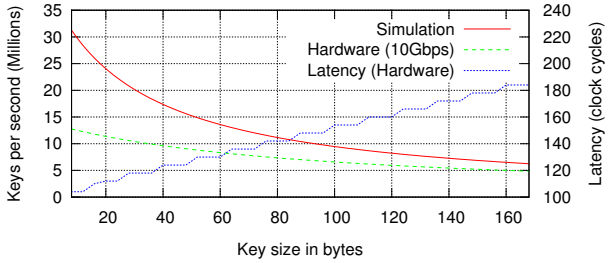

Fig. 9. Measured latency and maximum read and write performance of the hash table as function of key size

	Flip-flops	LUTs	BRAMs
Pipeline w/o Hash	13233 (2%)	11477 (4%)	43 (4%)
Hash Unit	5169 (1%)	16518 (5%)	24 (2%)
Total	18402 (3%)	27995 (9%)	67 (6%)

Table 1. The hash table on a Virtex6 SX475T chip

this constant latency adds up to 0.58μs. The hash function constitutes the variable part of the latency, which grows in steps with key size because the key has to be padded to multiples of 96bit words for hashing.

The numbers measured in Figure 9 correspond to workloads without address conflicts. In the presence of such conflicts, the concurrency control unit introduces backpressure in the pipeline, lowering the maximum achievable performance. On our platform for instance, the line-rate processing goal may not be achieved for all key sizes if more than 5% of all operations are stalled despite the over-provisioned throughput. We argue that for a hash table with 1 million entries this is unlikely to happen. For this we assume that in large hash tables there is a subset of addresses which are frequently accessed with a uniform random distribution which we refer to in the following as the *working set*. Further, M stands for the maximum number of items in the critical section (this is a function of the memory latency), and N for the randomly accessed address lines in the hash table, the size of the working set so to speak. Finally, the fraction of *sets* and *deletes* in the operation mix is S. With this, the probability of having a write operation in the critical section that conflicts with the current item in the concurrency control unit can be expressed as: $P_{col} = 1 - (1 - \frac{1}{N} * S)^M$. Based on this formula, and assuming common operation mixtures described in [2], we can derive that for only a minimum of $N = 500$ out of 1 million entries, the expected address conflict probability stays under 5% for most workloads and with that the line-rate performance can be met.

5.3. Resource Consumption

Table 1 shows the resource consumption of the presented setup. Overall, the hash table uses only a small fraction of the chip, with the actual hash unit being its largest contributor. As explained in the previous sections, the size of the hash table is independent of the number of items stored in DRAM and has no effect on resource requirements. However the memory latency determines the size of the concurrency control unit. Additionally, the number of BRAMs used for buffering depends on both the memory latency and the maximum allowed key and value sizes.

6. CONCLUSION

In this paper we present the design and the implementation of a hash table for an FPGA-based memcached server. We provide 10Gbps line-rate performance by pipelining the hash table and deploying a parallel lookup technique in conjunction with a fixed bucket size. By choosing a large bucket size, we minimize the probability of cache misses/full events at the expense of memory density. Given the usage of DRAM we believe this to be an acceptable compromise. Further, we handle a large range of key sizes by striping keys over multiple memory lines and thereby matching memory read access time with corresponding packet budgets. Furthermore, we utilize a strong hash function to provide constant behaviour regardless of the key contents. Finally, the hash table is parametrizable in regards to memory latency and memory access bandwidth, thereby making the implementation more portable.

7. ACKNOWLEDGEMENTS

The authors would like to thank Louis Woods and Jens Teubner from the Systems Group, and Oliver Pell, Rob Dimond and Andrew McCaffrey from Maxeler Technologies for their continued support in this project.

8. REFERENCES

[1] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, no. 124, pp. 72–74, 2004.

[2] 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*. ACM, 2012, pp. 53–64.

[3] M. Blott, K. Karras, L. Liu, Z. Istvan, J. Baer, and K. Vissers, "Achieving 10gbps line-rate key-value stores with fpgas," in *HotCloud'13. The 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, 2013.

[4] B. Jenkins, "Function for producing 32bit hashes for hash table lookup," http://burtleburtle.net/bob/c/lookup3.c, 2006.

[5] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspec-

tion," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 644–647.

[6] R. Pagh and F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[7] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2009.

[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*. ACM, 2013, pp. 245–254.

[9] M. Bando, N. S. Artan, and H. J. Chao, "Flashlook: 100-gbps hash-tuned route lookup architecture," in *High Performance Switching and Routing, 2009. HPSR 2009. International Conference on*. IEEE, 2009, pp. 1–8.

[10] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008, pp. 101–105.

[11] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM journal on computing*, vol. 29, no. 1, pp. 180–200, 1999.

[12] A. Wiggins and J. Langston, "Enhancing the scalability of memcached," http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached, 2012.