A Hash Table for Line-Rate Data Processing

ZSOLT ISTVÁN and GUSTAVO ALONSO, Systems Group,
Department of Computer Science, ETH Zürich
MICHAELA BLOTT, Xilinx Labs, Ireland
KEES VISSERS, Xilinx Labs, California

FPGA-based data processing is becoming increasingly relevant in data centers, as the transformation of existing applications into dataflow architectures can bring significant throughput and power benefits. Furthermore, a tighter integration of computing and network is appealing, as it overcomes traditional bottlenecks between CPUs and network interfaces, and dramatically reduces latency.

In this article, we present the design of a novel hash table, a fundamental building block used in many applications, to enable data processing on FPGAs close to the network. We present a fully pipelined design capable of sustaining consistent 10Gbps line-rate processing by deploying a concurrent mechanism to handle hash collisions. We address additional design challenges 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 parameterized to work with different memory types operating at different access speeds and latencies.

We have tested the proposed hash table in an FPGA-based memcached appliance implementing a main-memory key-value store in hardware. The hash table is used to index 2 million entries in 24GB of external DDR3 DRAM while sustaining 13 million requests per second, the maximum packet rate that can be achieved with UDP packets on a 10Gbps link for this application.

Categories and Subject Descriptors: B.6.1 [**Hardware**]: Logic Design—*Parallel circuits*; E.2.3 [**Data**]: Data Structures—*Hash-table representation*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: FPGAs, data structures, hash functions, parallel architecture

ACM Reference Format:
Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2015. A hash table for line-rate data processing. ACM Trans. Reconfig. Technol. Syst. 8, 2, Article 13 (March 2015), 15 pages.
DOI: http://dx.doi.org/10.1145/2629582

1. INTRODUCTION

Field-programmable gate array (FPGA)-based data processing is becoming increasingly relevant in data centers thanks to the inherent parallelism in these devices that allows for creating application-specific dataflow implementations. These implementations can bring significant performance benefits in the form of throughput and latency, as well as power reduction. Examples of such FPGA applications range from streaming query execution [Mueller et al. 2009] and pattern matching [Woods et al. 2010] to

This work was funded in part by the Enterprise Computing Center (ECC: http://www.ecc.ethz.ch).
Authors' addresses: Z. István and G. Alonso, Systems Group, Department of Computer Science, ETH Zurich, Zurich, Switzerland; emails: {zsolt.istvan, alonso}@inf.ethz.ch; M. Blott, Xilinx Labs, Citywest Business Campus, Dublin, Ireland; email: mblott@xilinx.com; K. Vissers, Xilinx Labs, San Jose, CA 95124, United States; email: kvissers@xilinx.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 1936-7406/2015/03-ART13 $15.00
DOI: http://dx.doi.org/10.1145/2629582

database-related tasks such as compression or encryption [Arasu et al. 2013; Francisco 2011] or SQL query execution [Dennl et al. 2013]. FPGAs are also used to speed up main-memory caching services by parsing packets and calculating hash values of keys [Lavasani et al. 2013; Convey 2013]. In this article, we focus on a widely used data structure relevant in this domain: hash tables. Hash tables are commonly used in software whenever an item from a set needs to be quickly retrieved; however, at this time, there is no large-scale line-rate FPGA implementation suitable for use in data centers.

There is a large body of prior work in the context of hash tables on FPGAs targeted at networking use cases, such as high-speed IP filtering or flow classification (e.g., Weaver et al. [2007] and Puš and Korenek [2009]). The key difference in data processing is that unlike in networking use cases, the hash tables need to hold more entries (millions), and key and value size are flexible and typically much larger as well. In this work, we rely on the parallelism of the FPGAs to provide a scalable, high-throughput, high-capacity hash table suitable for a wide range of data streaming applications. To achieve this, we use parallel lookups—that is, we compare a key to multiple locations in the hash table in parallel, and we decouple key and value storage, to achieve both very high flexibility in data sizes and high storage efficiency.

Development on FPGAs is a slower process than on CPUs, and code reuse is therefore of high importance. We aimed at an implementation that could ensure line-rate processing independent of the network interface, such as Ethernet or Infiniband, and in anticipation of future use cases we also wanted to support keys as large as hundreds of bytes and values ranging up to several megabytes without having to change the architecture. Finally, we aimed to provide an implementation independent of memory latency and access bandwidth to achieve a certain degree of portability between platforms and storage types (SRAM, DRAM, and flash).

The ideas presented in this article extend and build upon our previous work [Istvan et al. 2013]. The results have been evaluated as part of a larger project: a memcached appliance running on an FPGA [Blott et al. 2013]. Memcached is a main-memory cache for Web applications that exposes over the network an interface to store key-value pairs, similar to hash tables. By integrating our hash table into the dataflow architecture of this appliance, we make it possible to (1) scale the key-value store to millions of entries in DRAM and (2) provide enough throughput for a real-life 10Gbps data center application.

The rest of the article is organized as follows. Section 2 discusses related work in hash tables on FPGAs. Section 3 describes the two hash functions used in our work. A high-level overview of the implementation is given in Section 4, whereas Section 5 describes our parallel dataflow architecture in detail. Memcached, which is our main use case, and the evaluation results are presented in Section 6, and conclusions can be found in Section 7.

2. RELATED WORK

Hash tables provide an effective solution to the common search problem of retrieving a value that is associated with a key. The main challenge is in handling the case when multiple keys map to the same hash index (*hash collision*) while maintaining consistent throughput levels. In this section, we review popular ways of implementing hash tables on FPGAs. In general terms, we differentiate between the case in which the input keys are known beforehand and the hash collisions can be completely avoided, and the more general case in which little is known about the contents of the keys and the handling of hash collisions represents a major challenge.

One well-known approach for cases where all input keys are known beforehand is called *perfect hashing*. This method relies in essence on the idea of creating or customizing the hash function itself such that collisions can be completely avoided for

a previously known set of input keys. Using a perfect hash function to locate elements in a table has the advantage of guaranteed constant time read and write operations. However, finding such a function for a set of keys can be computationally complex on an FPGA (more so if the keys are long strings) and has to be repeated every time a key changes. Therefore, this scheme is particularly attractive in cases where the key set is near static, such as in the case of detecting patterns in an intrusion detection system (e.g., Sourdis et al. [2005]).

For many other use cases, the key set is unknown and changes dynamically over time. For these, a predefined hash function is used, and collisions will inevitably occur. A hash table with an *open addressing* scheme solves these collisions in the following way: when an item needs to be inserted into the table, its hash location is examined, and if it is already occupied by another item, the next locations are sequentially examined until an empty slot is found. The sequence in which the locations are scanned is determined by the specific type of open addressing scheme used (e.g., linear, quadratic). The main drawback of solving hash collisions this way is that both read and write operations take variable time to finish. Additionally, after a delete, housekeeping may be necessary to reorganize the table to fill in the "gaps."

Cuckoo hashing [Pagh and Rodler 2004; Kirsch et al. 2009] is an open addressing variant that uses two hash functions instead of just one. In this table, an item has two possible locations (the index provided by two different hash functions), and if the item resides within the table, it is guaranteed to be in either of the two. When performing a read access, only these two locations need to be examined. In case of a write, if both locations are taken by other items, a greedy algorithm is used to reorganize the table until it meets the preceding invariant for all items. In a cuckoo hash table, reads are always answered in constant time. However, writes pay the cost of collisions, resulting in unpredictable response times.

An alternative to the open addressing approach, often used in software implementations, is to store items that hash to the same address in a *bucket* belonging to that address. Inside the bucket, items are organized as a linked list of variable length. Although the disadvantage of variable-time reads and writes applies to this technique as well, its popularity in software can be explained by the fact that its performance degrades gracefully with increasing load factors. In other words, the table will function even if the number of keys is more than what was anticipated at the time of memory allocation. The price is that it will require more than one memory access per operation. When using a good hash function, the number of items in the buckets will be proportional to the load factor and the access time will be mostly constant. Deviations from the median access time can be potentially large, which constitutes a challenge for pipelined hardware implementations. Often, the bucket size is limited to restrict the potential penalty of the sequential search in the linked list inside the bucket [Chalamalasetti et al. 2013].

FPGA solutions can also leverage multiple off-chip memories to provide constant performance in the hash table even with collisions [Bando et al. 2009]. By comparing the input key in parallel to multiple entries in a hash table bucket, the decision-making process can be accelerated. Instead of parallelizing inside a bucket, it is also possible to parallelize at a higher granularity. This could be done by dividing the hash table into multiple physical parts, each of them addressed by a different hash function: accesses and comparisons could be carried out in parallel. The obvious drawback of any kind of parallel lookup is that it needs substantially larger memory bandwidth than the sequential lookup schemes—the trade-off is constant access time for higher memory bandwidth [Broder and Karlin 1990; Broder and Mitzenmacher 2001]. Here we adopt a similar approach. Instead of using multiple memory interfaces, we use only one wide memory interface; furthermore, a key difference in our work is that we support keys

of variable size. For this, we stripe the data structure over multiple entries. Thereby, the read access time is directly proportional to the size of the key, and with that the time to process a packet. In other words, this allows us to process keys at line rate even though they have variable access times (this is explained further in Section 5). In summary, unlike prior art, this scheme allows for parallel reads to multiple entries while handling flexible key sizes, trading off storage efficiency for flexibility.

Sometimes the previously mentioned techniques are combined with an *overflow buffer* that holds all items that cannot be stored because their respective buckets are full. Typically, these are realized with small content-addressable memories (CAMs) [Bando et al. 2009]. To combine this idea with cuckoo hashing, [Kirsch et al. 2009] modify the cuckoo hash table so that it only tries moving a single item when an insert fails, and upon failure stores the item in an overflow buffer. This works well for use cases with good hash functions and small table sizes, where the overflow buffer remains small and fits inside a CAM. Extending this idea to large hash tables (millions of entries) is unfortunately not feasible because CAMs are restricted in their maximum size. Furthermore, CAMs are expensive when used in conjunction with flexible key and value sizes.

For the chosen application domain, we did not see the need for such an overflow buffer, as it is not necessary to resolve hash collisions above a certain threshold. For instance, in case of main-memory caches, where no guarantee is offered in the first place on whether the data is to be found in memory, solving a collision for an insert operation at the expense of serving several reads in the meantime could significantly impact performance.

3. HASH FUNCTIONS

At the heart of every hash table is a hash function that influences the behavior of the data structure: if the hash function is poorly chosen, collisions will happen much more often than expected, creating significant access or update time penalties. For this reason, we considered multiple hash functions, from which we present two. One of these functions is relatively complex but provides an all-round reliable quality, whereas the other one trades off hashing quality for a more modest logic footprint. In this section, we discuss the two functions and their suitability for our target workloads. In the following section, we explain how the functions are combined with the rest of the hash table.

The *Lookup3* hash function [Jenkins 2006] is a popular choice for open-source software projects because it is proven to hash well a broad range of key types with a reduced chance of collisions. This is due to a property called the *Avalanche effect*, which results in different hash values for keys that differ even in single bits only. This hash function processes variable-size keys iteratively in 96-bit chunks. Each chunk is split into three 32-bit numbers that are added to a set of three state variables. Before the next chunk is processed, these state variables are mixed using addition, subtraction, and XOR operations. Due to the feedback loop nature of the algorithm, the hash function cannot be easily pipelined: in our implementation, the mixing step is performed in six clock cycles. As a consequence, the hash function can consume a new 96-bit input every six cycles; however, in Section 4, we show how the overall hashing throughput can be increased by means of replication. Given the complexity of the algorithm, it is not surprising that it requires a significant number of logic gates in exchange for its high-quality output. Table I summarizes the resource requirements of this component.

We have also implemented a second hash function, which we call the *multiplicative hash function*; this is an adaptation of the multiplication-based function in Kernighan et al. [1988]. Even though this second implementation is not guaranteed to work well on arbitrary input data, it produces good hash values on average for long string inputs, such as one would encounter with database query caching. It is computationally

Table I. Resource Consumption of the Two Implemented Hash Functions

Name	Registers	LUTs	Slices	DSP48s	Hash Performance
Lookup3	352	1,253	369	0	96 bits/6 cycles
Multiplicative	104	79	51	2	64 bits/8 cycles

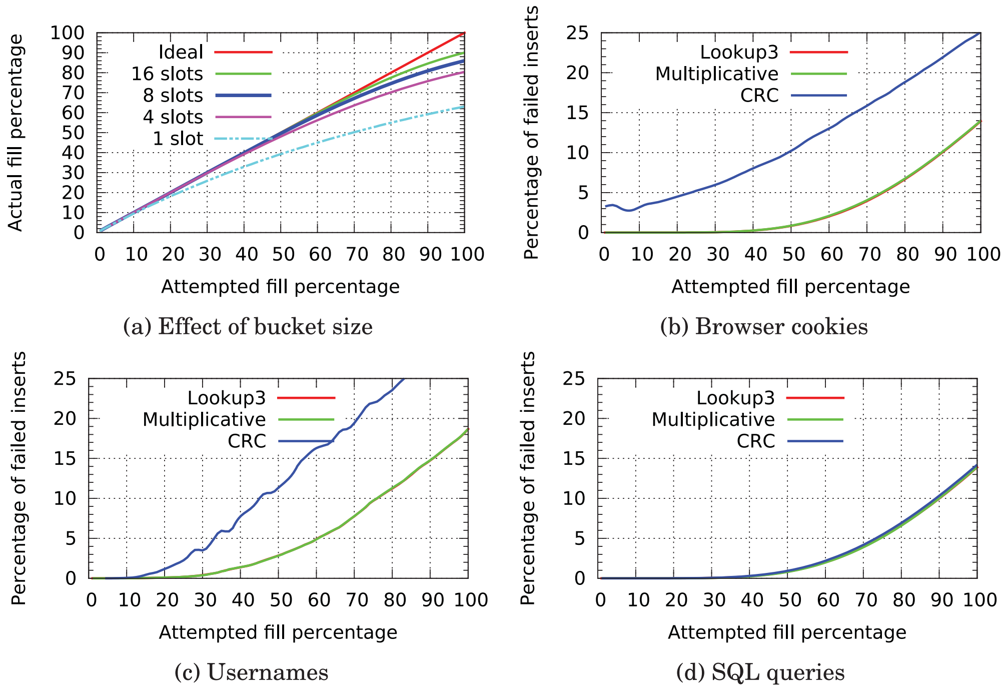

Fig. 1. Percentage of failed inserts with different hash functions.

much less complex and therefore requires less logic resources. The key differentiator to Lookup3 is in the mixing step, which relies only on multiplication and addition. The implementation works with 16-bit-wide input words. To calculate the hash values, it adds each input word to the hash and multiplies the result with a “magic number.”[1] As Table I shows, the multiplicative hash function has a much smaller logic footprint than the Lookup3, and it can also take advantage of DSPs. For projects in which chip real estate is not a limiting factor, the better hash function should be used to ensure the best possible behavior regardless of the input key distribution; however, if the resource consumption has to be minimized, the multiplicative function could provide an effective alternative.

As a first evaluation of the Lookup3 function, we looked at the interaction between bucket size and number of hash collisions. In Figure 1(a), we hashed a set of random keys with the Lookup3 function and tried to insert them into hash tables with different bucket size limits while keeping the total table capacity constant. We found that by increasing the bucket size limit, the hash table can be filled to a high extent before encountering too many lost items. Virtually no collisions occur when fill levels remain below 50%. Then we observed the effect of different key distributions on the number of collisions. For this, we derived three sets of keys based on the most common memcached use cases described in [Atikoglu et al. 2012] (Figure 1(b) through (d). These are user

[1]Traditionally a large prime number.

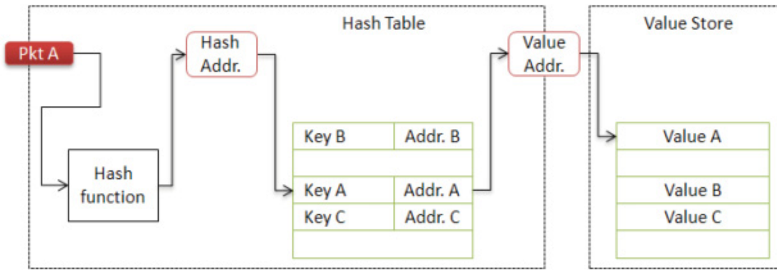

Fig. 2. Hash table and value store.

names generated from a dictionary of common first and last names, MD5 hashes of SQL-query strings, and a concatenation of four numeric IDs, respectively. To measure the number of collisions for the different keys, we fixed the bucket size at eight items and the table capacity at 1 million. In Figure 1(b) through (d), we compare the Lookup3 and the multiplicative function, and with a simple cyclic redundancy check (CRC) algorithm for reference. A simple CRC module might be tempting for use on FPGAs because of its very high throughput and minimal resource requirements, but it is prone to producing collisions, as can be seen in the case of user names and browser cookies. Lookup3 performs well overall, as the number of hash collisions is very small. Surprisingly, for these three synthetic workloads, the multiplicative hash function behaved almost indistinguishably from Lookup3; however, with real-world data, we expect it to be less robust. When compared with the CRC results, both hash functions are well worth their higher cost in real estate within the FPGA.

4. IMPLEMENTATION AT A GLANCE

Our hash table supports the three basic operations of lookup tables: *get*, *insert*, and *delete*. The *get* operation retrieves the value associated with the input key, if present in the hash table. The *insert* operation inserts a key-value pair into the hash table if there is enough space and the key is not already in the table. If the key is already present in the table, then it simply replaces its associated value. Finally, the *delete* operation removes a key-value pair from the hash table and marks its location empty.

To maximize the flexibility in data storage, we choose a decoupled design (Figure 2) for the hash table, which means that the keys and values are stored in separate memory areas. The benefit of this separation is that different memory management techniques can be used for keys and values, which might be orders of magnitude larger than the keys. The cost of decoupling is additional latency and circuit complexity, but as we show in Section 6, the final footprint of the hash table is small enough that it makes this trade-off worthwhile. For use cases where minimizing latency is the final goal, the hash table can be used with fixed-size values (after all, the value address pointer is a value itself). In the following, when we refer to the hash table, we speak about the part that stores the keys and the fixed-size pointers to the value store. The logic to implementing the value store, which writes to and reads from the addresses provided by the hash table, is described in more detail in Blott et al. [2013].

As illustrated in Figure 3, data enters and leaves the hash table's pipeline through flow control–enabled streaming interfaces based on the AXI-Streaming standard.[2] Keys and values are transmitted over these interfaces in parallel in 64-bit data buses. The metainformation associated with each request, such as operation code, key, and value

[2]Specification at http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

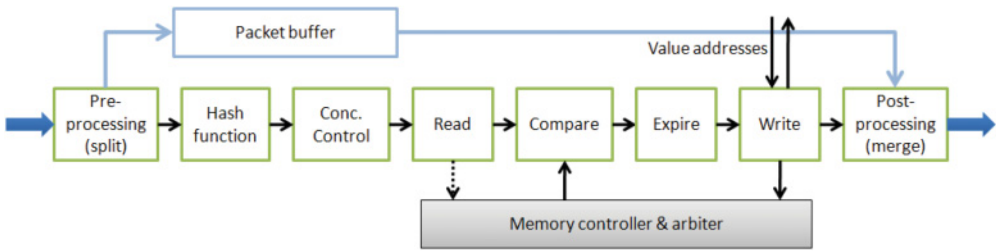

Fig. 3. Hash table pipeline structure.

length, are conveyed over a third parallel channel. The actual hash table logic sits between the *preprocessor* and the *postprocessor*. The preprocessor extracts the key and relevant metadata from the stream and stores the rest of the packet in a packet buffer, and the postprocessor merges the results from the hash table back into the packet. First, the *hash function* calculates 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 artifact 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 optional *expiration unit* invalidates expired keys based on an internal counter that keeps time in seconds. Finally, the *write unit* is responsible for updating the hash table for all writing operations, such as *inserts* and *deletes*. When done, the write unit outputs the location of the value in memory and its length for successful operations or, in case the operation does not succeed, the corresponding error code.

The *compare-expire-write* logic is designed to be able to match the available memory bandwidth and interface width on different types of memories. This is achieved by parameterization: memory lines are split into a parameterizable number of parts, which are then handled in parallel, and the design is further parameterized to support different maximum key sizes. The data structures in the concurrency control unit can also be adjusted to the memory latency. With that, we achieve independence of memory interface and type, thereby creating some degree of portability.

To achieve better 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 that 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. By making the address generation task external to the main pipeline logic, we aim to provide more portability to our implementation: on every platform, a suitable address generation scheme can be chosen.

Since neither of the two hash functions presented can process incoming data at 10Gbps, we replicate these functions and deploy in parallel either (1) eight instances of the Multiplicative hash function or (2) eight instances of the Lookup3 hash function (even if the latter function has a higher overall throughput than the multiplicative function, we still needed to deploy eight copies to account for quantization effects). Work is distributed and collected in a round-robin manner as illustrated in Figure 4.

5. PARALLEL HASH TABLE

In the hash table, we handle collisions with a variant of the *chaining* method explained in Section 2. A list of constant length is preallocated at each hash table address, and

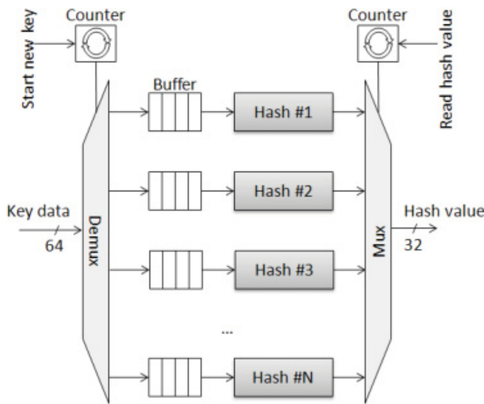

Fig. 4. Parallel hash unit.

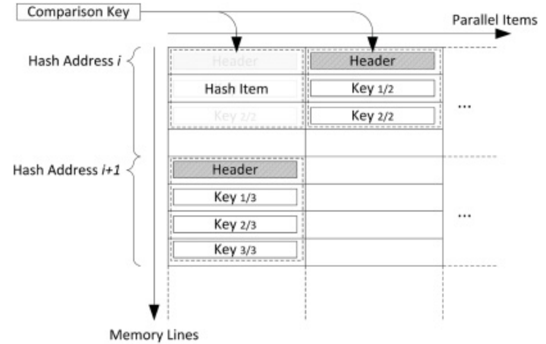

Fig. 5. Hash table layout in memory.

for every incoming key the entire list[3] is retrieved from the table and compared to the respective key in parallel. This design is essentially a trade-off between *probability of collisions and memory bandwidth*. With increasing bucket size, the probability of collisions can be reduced, but at the same time, the necessary memory bandwidth grows and at some point will act as an upper bound to the number of parallel items.

If every hash table location would correspond to only one *memory line* (which is the equivalent of one memory access burst), then the maximum key size would be limited by 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 5. A bucket is spread "horizontally" over a memory line, whereby each hash item can span multiple lines "vertically." 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.

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 want to use DRAM for the hash table, we believe that the density requirements are less critical. On our development platform with 24GB of main memory, for instance, with less than 400MB we can support 2 million entries and 23.6GB of value storage.

If a key is not found within a bucket, we declare it a miss and no further address lines are accessed. We believe that this behavior is acceptable for our target applications, where the key-value store acts as a cache and cache misses are dealt with on a system level. This way, we can guarantee constant access time for a given key size 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—size 8 to be specific. For other applications and smaller table sizes, an overflow CAM could be inserted as discussed in Section 2. In Section 3, we show that with a good choice of hash function, and by utilizing the hash table to some fixed percentage of its capacity, the number of lost items can be kept at a minimum.

[3]For the rest of the article, we will refer interchangeably to these fixed size lists as buckets.

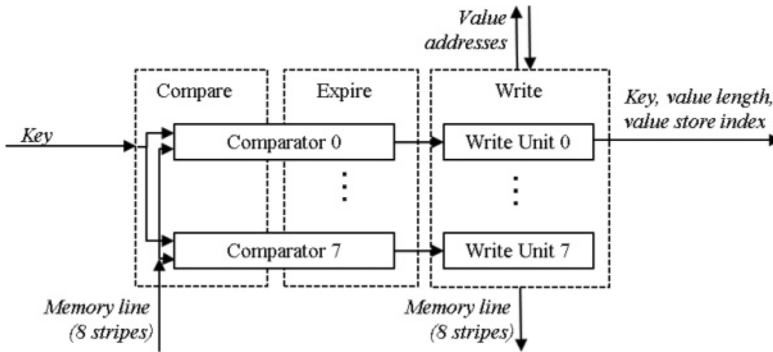

Fig. 6. Parallel operation of compare, expire, and write units.

5.1. Flexible Handling of Keys and Memory

Implementing a hash table that 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. These keys have to be matched in parallel to multiple keys that reside in a number of memory lines. On our platform, a memory line is 384 bytes in total, which corresponds to a burst length of 8 on the 384-bit-wide memory interface. As illustrated in Figure 6, the *compare* and *expire unit*s are split into parallel components to support the processing of multiple hash items in parallel. The input key is then broadcast to all *comparators* that 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 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 representing whether the stripe matches the input key, whether the key it holds has expired, 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 does not modify the memory and outputs only the header of the key that 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.

A concurrency control unit is required to protect against read-after-write hazards in the pipeline, as a write to a specific memory address may not complete before a read to the same address has been issued. This unit effectively blocks keys from entering the read unit while a writing operation on the corresponding hash address is still in progress. As such, the concurrency control unit is located between the hash and read units to handle these conflicts. For the implementation of this unit, 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 that can concurrently reside within the pipeline's critical section depends on the latency of the memory. We have therefore

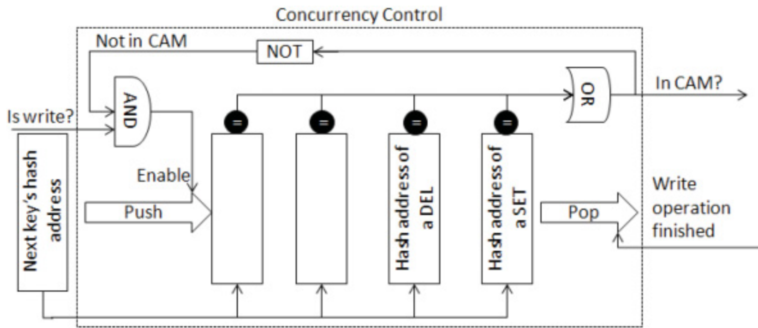

Fig. 7. Concurrency control unit.

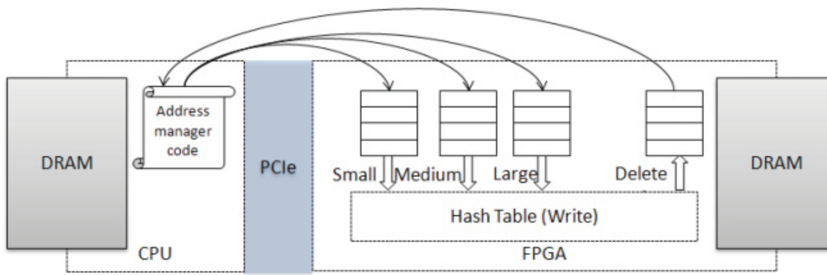

Fig. 8. The queues holding block addresses stretch from the CPU to the FPGA.

parameterized 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, 64 entries can accommodate for the worst case in memory subsystem latency. 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 is introduced to minimize the effect of temporarily blocking the entrance to the next pipeline stage. In Section 6.1, we show that already for a minimal working set of 500 out of 1 million entries with a read-biased access pattern, the effects of the related stalls are not observable.

5.2. Memory Management

To maximize the applicability of the hash table and support use cases with a wide range of value sizes, we decoupled the value store from the hash table, whereby the hash table returns pointers into the value store rather than the actual values themselves. For this reason, we need memory management logic that allocates memory dynamically in predefined block sizes (similar to the "slab memory management" technique used in software applications such as memcached [Memcached 2013]), which can easily be accessed by the hash table when new entries have to be inserted or old ones removed. In our implementation, the task of generating and storing value store pointers is carried out by a CPU communicating with the FPGA using PCIe. Figure 8 shows how this system looks, with three queues (for small-, medium-, and large-size memory blocks) fed from the CPU, and with a fourth queue that returns the deleted addresses to the CPU. These addresses are 32-bit numbers, with the most significant 2 bits encoding their size type. As a result, the system can potentially store as much as 2^{30} key-value pairs. Relying on PCIe for communication between the hash table and the memory management logic is not a limiting factor, because in our platform, the PCIe connection

can handle 2GB/s bidirectional traffic, which translates to a throughput of 500 million addresses per second.

The most important advantage of including the CPU in the memory management task is that it enables experimenting with dynamic allocation schemes: the full list of addresses for each block size need not be generated up front but instead can be created in a demand-driven way. One caveat of this solution, though, is that the queues extending from the CPU to the FPGA have to be relatively large (tens of thousands of entries) to make sure that insert operations can be performed even if the CPU temporarily becomes unavailable to move addresses between the queues. The code that we have implemented to run on the CPU is relatively simple. It relies on four threads, with each one responsible for a queue. These threads are of two types: those pushing the free addresses to the FPGA and those transferring the deleted ones to the CPU. On startup, an initial division of the address space is performed and the three queues are created with the addresses. The deleted addresses are distributed into the three blocking queues by the deletion thread, and from these queues they are further moved to the FPGA with the help of the three worker threads.

We reduce communication costs between the FPGA and the CPU with the help of *address reuse* inside the hash table. The idea is that the hash table will try to reuse the value store addresses in the memory line where the key belongs instead of fetching a new one from the queues. When performing an insert, a new address is fetched when (1) the key is already in the table, only if the address associated with the key points to a block too small to store the value, or (2) when the key is not in the table, only if no pointers from expired items in the same line can be reused. The types of value storage blocks are determined based on the two most significant bits of the pointers.

5.3. Adding Expiration Time to Items

In scenarios when the key-value store is used as a cache, as is the case of memcached deployments, it is useful to represent the concept of data freshness to the users. Key-value pairs can be augmented with an expiration time, expressed in seconds, after which they are evicted from the cache, forcing the application to retrieve fresher information.

On our FPGA, items that expire are treated as if a *delete* command would have been issued on them—we call this behavior *delete-on-expire*. A very important design decision was to choose when to perform the deletion of expired items: on every access to a hash table bucket or just on opportunistic occasions. We chose to perform the *delete-on-expire* only together with an *insert* or *delete* operation—*reads* just treat the expired items as empty slots. As a result, the description of value pointer management in Section 5.1 can be extended to to expired items—that is, the 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. The reason for performing these steps only together with *insert*s and *delete*s is twofold: (1) if reading the table could have side effects, then the stalling behavior of the pipeline would need to be extended to *read* commands as well, resulting in very high performance penalties, and (2) empty space needs to be physically reclaimed only when a new item is inserted, and in all other cases it is safe to ignore the expired items in logic.

6. MEMCACHED USE -CASE

Memcached [2013] is an open-source in-memory key-value store. It has a simple interface that allows the storage and retrieval of arbitrary binary data based on a given key. Most often, memcached is used to speed up Web applications by acting as a fast write-through cache for the databases backing these applications [Fitzpatrick 2004]. This means that the result of a query is first checked in the cache, and only if not found is the database queried for the result. When a value is changed, the cache and

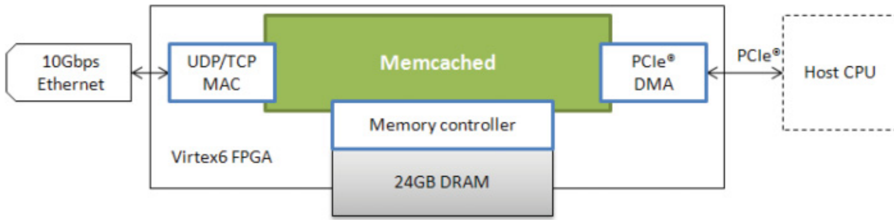

Fig. 9. Memcached deployed on a Maxeler workstation.

the database are both updated so that subsequent requests can be answered without accessing the database again. To achieve fast response times, memcached internally relies on a large hash table to store the key-value pairs, and it uses the Lookup3 hash function to index the table. Per the memcached protocol specification, keys can be as large as 250 bytes, and values are allowed to reach 1MB in size. In case the hash table grows too large, a least recently used (LRU) policy is used to remove items and free up space. The clients communicate with memcached over a simple protocol that builds on variations of three basic commands: *get*, *store*, and *delete*, which are semantically equivalent with the hash table operations. From these three, *get* operations typically predominate in most real-world workloads [Atikoglu et al. 2012].

There is an ongoing effort on implementing memcached as an FPGA appliance [Blott et al. 2013] with the goal of improving performance and reducing the power footprint at the same time. Figure 9 depicts this data appliance running on an FPGA-based network adapter. The heart of the platform is a Xilinx® Virtex6® SX475T chip, which is connected to a 10Gbps Ethernet interface and 24GB of DDR3 SDRAM. The memory is accessed in 384-bit 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 16GB 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 were measured with the UDP-based binary memcached protocol.

Our prototype platform offers a single DDR3 DRAM interface that has to be shared between the value store and the hash table. Fundamentally, this implies an almost completely random read pattern in address sequences, resulting in a relatively poor access performance. This is further aggravated by the fact that the DRAM also needs to be written in a random fashion, to update the value store and the hash table, introducing inefficiencies due to the associated data bus turnaround. For most workloads considered, the average data bus utilization was around 20%.

In our experiments, we deployed a hash table with the Lookup3 function, a capacity of 2 million items, and a maximum key size of 168 bytes, which accommodates many common workloads of memcached [Atikoglu et al. 2012]. The table described earlier occupied less than 400MB in DRAM and was used to address the remaining 23.6GB of memory allocated to the value store.

6.1. 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 168 bytes. We performed this experiment first in simulation, with a value size of 1 byte, to measure the maximum throughput the hash table's pipeline is capable of. Similarly, we ran hardware experiments in which we measured the maximum throughput

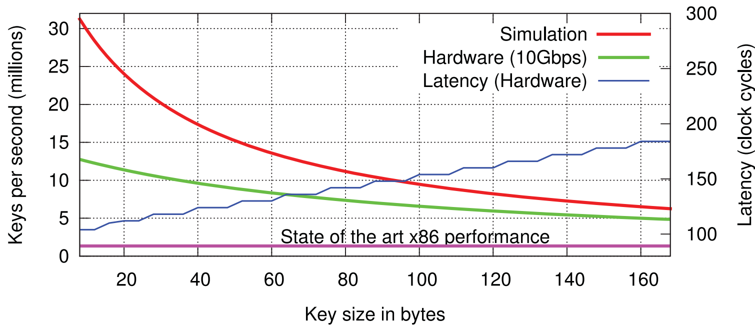

Fig. 10. Measured latency and maximum read and write performance of the hash table as function of key size in simulation and on the FPGA.

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 258 bytes. Given the minimal packet size of 96 bytes, a maximum packet rate of 13 million requests per second (MRPS) can be achieved before the network is fully saturated. As shown in Figure 10, the hash table is overprovisioned and can handle throughput beyond 10Gbps, servicing a maximum packet rate of more than 31MRPS. This overprovisioning helps to accommodate for additional overhead associated with resolving address conflicts. For comparison, we also plotted in the figure the state of the art in memcached performance on server-grade CPUs [Wiggins and Langston 2012]. The FPGA-based solution outperforms a single-socket, eight-core CPU by a factor of 10.

Figure 10 also illustrates the average latency introduced by the hash table on our evaluation platform. This number is composed of a constant part and a part that increases linearly with the key size. The constant part of the latency is roughly 90 cycles, of which 60 are a direct result of the memory subsystem. Given a clock period of 6.4ns, 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 96-bit words for hashing.

The numbers measured in Figure 10 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 throughput. 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 overprovisioned 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 that are frequently accessed with a uniform random distribution. We refer to this set 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—that is, the size of the working set. 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 waiting in the concurrency control unit can be expressed as $P_{col} = 1 - (1 - \frac{1}{N} * S)^M$. Based on this formula, and assuming the common operation mixtures described in Atikoglu et al. [2012], we can derive that starting from $N = 500$ 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. For larger values of N, the probability of stalls will shrink further, ensuring sufficient throughout with high probability.

Table II. The Hash Table on a Virtex6 SX475T Chip

	Flip-Flops	LUTs	BRAMs
Hash Table Pipeline w/o Hash Unit	13,233 (2%)	11,477 (4%)	43 (4%)
Hash Unit (Lookup3)	5,169 (1%)	16,518 (5%)	24 (2%)
Hash Table Total	18,402 (3%)	27,995 (9%)	67 (6%)

6.2. Resource Consumption

Table II shows the resource consumption of the hash table module. Overall, the hash table uses only a small fraction of the chip, with the actual hash unit being its largest contributor. As explained in 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.

7. CONCLUSION

In this article, we present the design and the implementation of a hash table for line-rate data processing on an FPGA. We achieve 10Gbps line-rate performance by pipelining the hash table and deploying a parallel lookup technique in conjunction with a fixed bucket size. By separating the keys and their corresponding values to different memory regions, we can use large bucket sizes, which minimizes the probability of hash collisions, without sacrificing memory utilization. We handle a large range of key sizes by striping keys over multiple memory lines, thereby matching memory read access time with corresponding packet budgets. Furthermore, we utilize a strong hash function to provide constant overhead regardless of the key contents. Finally, the hash table is parameterizable in regard to memory latency and memory access bandwidth, which makes the implementation more portable.

This design has been used in an FPGA-based memcached appliance [Blott et al. 2013], in which it demonstrated 10Gbps throughput while managing 24GB of DRAM holding millions of key-value pairs. Additionally, the same design has been used in IBEX, a smart database storage engine running on an FPGA, to perform a Group By aggregation of database records [Woods et al. 2014].

ACKNOWLEDGMENTS

The authors would like to thank Kimon Karras and Ling Liu from Xilinx Labs and Louis Woods and Jens Teubner from the Systems Group for their support in this project.

REFERENCES

Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. 2013. Orthogonal security with Cipherbase. In *Proceedings of the 6th Conference on Innovative Data Systems Research (CIDR)*.

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, 53–64.

Masanori Bando, N. Sertac Artan, and H. Jonathan Chao. 2009. Flashlook: 100-Gbps hash-tuned route lookup architecture. In *Proceedings of the International Conference on High Performance Switching and Routing*. IEEE, Los Alamitos, CA, 1–8.

Michaela Blott, Kimon Karras, Ling Liu, Zsolt Istvan, Jeremia Baer, and Kees Vissers. 2013. Achieving 10Gbps line-rate key-value stores with FPGAs. In *Proceedings of HotCloud'13: The 5th USENIX Workshop on Hot Topics in Cloud Computing*.

Andrei Broder and Michael Mitzenmacher. 2001. Using multiple hash functions to improve IP lookups. In *Proceedings of INFOCOM 2001: The 20th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, Los Alamitos, CA, 1454–1463.

Andrei Z. Broder and Anna R. Karlin. 1990. Multilevel adaptive hashing. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*. 43–53.

Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. 2013. An FPGA memcached appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 245–254.

Convey. 2013. *Ramping Up Web Server Memcached Capabilities with Hybrid-Core Computing*. White Paper. Retrieved March 2, 2015, from http://www.conveycomputer.com/files/6113/7998/5068/CONV-13-047_MCD_whit epaper.pdf.

Christopher Dennl, Daniel Ziener, and Jürgen Teich. 2013. Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration. In *Proceedings of the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Los Alamitos, CA, 25–28.

Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124, 72–74.

Phil Francisco. 2011. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbook*.

Zsolt Istvan, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2013. A flexible hash table design for 10Gbps key-value stores on FPGAs. In *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Los Alamitos, CA, 1–8.

Bob Jenkins. 2006. Function for Producing 32bit Hashes for Hash Table Lookup. Retrieved March 2, 2015, from http://burtleburtle.net/bob/c/lookup3.c.

Brian W. Kernighan, Dennis M. Ritchie, and Per Ejeklint. 1988. *The C Programming Language*, Vol. 2. Prentice Hall, Englewood Cliffs, NJ.

Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing* 39, 4, 1543–1561.

Maysam Lavasani, Hari Angepat, and Derek Chiou. 2013. An FPGA-based in-line accelerator for memcached. *IEEE Computer Architecture Letters* 2, 1.

Memcached. 2013. Free and Open Source, High-Performance, Distributed Memory Object Caching System. Available at http://www.memcached.org/.

Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: A query compiler for FPGAs. *Proceedings of the VLDB Endowment* 2, 1, 229–240.

Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2, 122–144.

Viktor Puš and Jan Korenek. 2009. Fast and scalable packet classification using perfect hash functions. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 229–236.

Ioannis Sourdis, Dionisios Pnevmatikatos, Stephan Wong, and Stamatis Vassiliadis. 2005. A reconfigurable perfect-hashing scheme for packet inspection. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, Los Alamitos, CA, 644–647.

Nicholas Weaver, Vern Paxson, and Jose M. Gonzalez. 2007. The Shunt: An FPGA-based accelerator for network intrusion prevention. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 199–206.

Alex Wiggins and Jimmy Langston. 2012. Enhancing the Scalability of Memcached. Retrieved March 2, 2015, from http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached.

Louis Woods, Zsolt Istvan, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced SQL off-loading. *Proceedings of the VLDB Endowment* 7, 11, 963–974.

Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex event detection at wire speed with FPGAs. *Proceedings of the VLDB Endowment* 3, 1 2, 660–669.

Received December 2013; revised April 2014; accepted April 2014