2015 IEEE International Parallel and Distributed Processing Symposium Workshop

High-throughput Online Hash Table on FPGA*

Da Tong, Shijie Zhou, Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: datong@usc.edu, shijiezh@usc.edu, prasanna@usc.edu

***Abstract*—Hash tables are widely used in many network applications such as packet classification, traffic classification, and heavy hitter detection, etc. In this paper, we present a pipelined architecture for high throughput online hash table on FPGA. The proposed architecture supports *search*, *insert*, and *delete* operations at line rate for the massive hash table which is stored in off-chip memory. We propose two hash table access schemes: (1) the first scheme assigns each hash entry multiple slots to reduce the hash collision rate; each slot can store the corresponding hash key of the hash entry; (2) the second scheme has a higher hash collision rate but a lower off-chip memory bandwidth requirement than the first scheme. Both schemes guarantee the line rate processing when using the memory devices with sufficient access bandwidth. We design an application specific data forwarding unit to deal with the potential data hazards. Our architecture ensures that no stalling is required to process any sequence of concurrent operations while tolerating large external memory access latency. On a state-of-the-art FPGA, the proposed architecture achieves 66-85 Gbps throughput while supporting a hash table of various number of entries with various key sizes for various DRAM access latency. Our design also shows good scalability in terms of throughput for various hash table configurations.**

***Keywords*-Hash table, Data forwarding, FPGA**

I. Introduction

Hash tables are commonly used whenever an item needs to be quickly retrieved from a set. Therefore hash tables have been applied to accelerate many network applications such as packet classification, traffic classification, and heavy hitter detection, etc [1], [2].

In recent years, 100 Gbps networking is becoming a standard. Both the research community and the industry are targeting 400 Gbps networks [3], [4]. State-of-the-art Field Programmable Gate Arrays (FPGAs) are promising platforms for high throughput implementation of hash tables [6], as they offer unprecedented logic density and very high on-chip memory bandwidth.

In this paper, we propose a high throughput online hash table on FPGA using external DRAM. The proposed architecture supports online operations including *search*, *insert*, and *delete* at line rate. There are three major design challenges. 1. Handling read-after-write data hazards due to the pipeline's processing latency and the DRAM's access latency without degrading the throughput. 2. Achieving line rate processing for DRAM devices with relatively low bandwidth. 3. Designing a pipelined architecture which integrates all the functions and achieves high throughput at the same time. In order to prevent the data hazard without stalling the pipeline, our proposed architecture uses efficient data forwarding to handle data hazards. We propose two hash table access schemes, first fit scheme and random fit scheme to work with relatively high and low bandwidth. Here relatively high/low bandwidth means the bandwidth is/is not high enough to retrieve all the slots of one hash table entry in one clock cycle. We summarize our contributions as follows:

- A pipelined architecture for online hash table. It guarantees line rate processing when working with DRAM devices of various memory access latency and bandwidth. The architecture sustains high throughput of 66-85 Gbps supporting 1-16 million entries, each entry having 2-8 slots with the key sizes ranging from 16 to 128 bits.
- An application specific data forwarding unit. It ensures the correctness of the architecture without stalling the pipeline regardless of the memory access latency.
- Two hash table access schemes. These schemes guarantee that we can perform one operation to the hash table per clock cycle for various DRAM access bandwidth.
- A high throughput multi-functional pipelined architecture. It supports all three typical operations to a hash table at line rate: *search*, *insert*, and *delete*.

The rest of the paper is organized as follows: Section II defines the problem we target. We introduce our architecture in Section IV. Section V evaluates the performance, and Section VI concludes this paper.

II. Problem Definition

The three basic operations of a hash table are:

- *Search*: The $search$ operation retrieves the $value$ associated with the input key if the input key exists in the hash table.
- *Insert*: The $insert$ operation first searches the input key in the hash table. If the key exists, it updates its associated $value$ with the input $value$. Otherwise, it

*This work is supported by U.S. National Science Foundation under grant CCF-1116781.

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.149

inserts a $key - value$ pair into the hash table if there is an open slot in the hash table entry.
- *Delete*: The $delete$ operation removes a $key - value$ pair from the hash table, and marks the corresponding slot as empty

Our goal is to design an online architecture which supports these operations. The input to the architecture is a sequence of $key - value - operation$ sets. Given fixed key and $value$ lengths, and the set of operations ($search$, $insert$, $delete$) , the architecture needs to be able to processes any arbitrary sequence of inputs. Online here means that the architecture is able to execute one operation to the hash table at line rate, i.e. one operation every clock cycle.

III. Related Work

Hash table enables fast table lookup. Therefore it is widely used in network applications which normally performs rule set lookup or collects per packet/flow information. [2] propose an algorithm for fast packet classification using perfect hash functions. By applying the hash function, each packet classification requires only 2 memory accesses. As a result, [2] achieves a throughput of 150 million packets per second (MPPS). In Instvan $et\ al.$ [7], a hash-based traffic classification engine is proposed. Instvan $et\ al.$ convert the classic multi-feature decision-trees to a set of hash tables. Their hash-based approach demonstrates good performance with respect to both throughput and scalability on the state-of-the-art multicore platforms. The design achieves over $10\times$ improvement compared with the traffic classifiers based on classic decision trees and other techniques. Cormode $et\ al.$ [1] present a hash table based statistical summary technique for data streaming applications. The technique uses multiple hash functions and shows a great bounded accuracy and a low memory footprint. It can be applied to online heavy hitter detection for network flows and many other network applications.

There have been many FPGA implementations for high performance hash tables. Bando $et\ al.$ [8] present a parallel hash table based IP lookup technique with collision resolution. Theoretically, with the help of multiple external memory devices, the architecture can achieve a lookup rate of upto 250 MPPS. However, the architecture focused on the *lookup* operation for hash tables. The *delete* and *insert* operations were not emphasized. Istvan $et\ al.$ [6] present the design and the implementation of a pipelined hash table for an FPGA-based MemcacheD [9] server. Their design achieves 10 Gbps for a wide range of key sizes. However, their design stalls the pipeline whenever read-after-write data hazard occurs. Therefore, the performance is highly dependent on the sequences of keys and operations.

Our proposed architecture supports all typical operations to a hash table. Moreover, since it doesn't need stalling to handle the data hazard, high throughput is guaranteed for any input sequences of keys and operations.

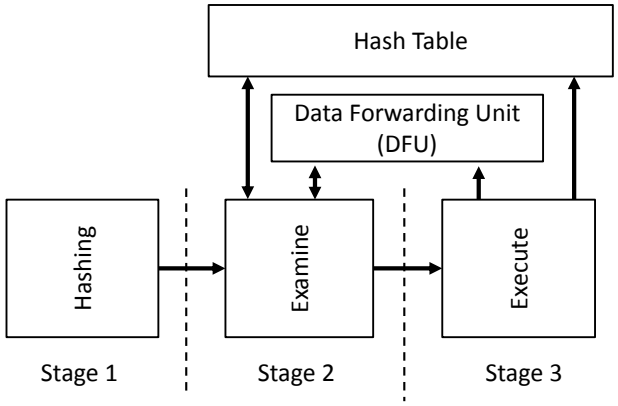


Figure 1: Pipelined architecture

IV. Architecture

A. Overall Architecture

In the proposed design, each hash table entry has multiple slots. Each slot stores one hash item (a $key - value$ pair). One valid bit is associated with each slot to indicate if this slot stores a valid hash item or is open for insertion. When an input key arrives, it is hashed to identify a hash entry and then compared with all the keys of the slots stored at the hash entry. We also check the validity of slots which is represented using a bitvector. Each bit in the bitvector corresponds to one slot. The intended operation is performed based on both the match of keys and the validity of the slots.

We map the operations into a 3 stage pipeline as illustrated in Figure 1.

- **Stage 1(Hashing):** The hash entry is computed
- **Stage 2(Examine):** The hash slots of the hash entry are retrieved and examined.
- **Stage 3(Execute):** The operation is performed according to the examination result from Stage 2.

The Data Forwarding Unit (DFU) forwards the necessary data into the Execute stage to prevent data hazard. It is discussed in detail in IV-C. The detailed operations at each stage of the pipelined architecture are shown in Algorithm 1.

B. Hash Function

The class H_3 hash function [10], has been proved to be very effective in distributing keys evenly among hash table entries [11]. This leads to a low hash collision rate. The class H_3 hash functions are defined as follows [11]:

DEFINITION 1. *Let $K = \{0, 1, ...2^I - 1\}$ be a key set of I bits, and $V = \{0, 1, ...2^J - 1\}$ be a hash value set of J bits. Let Q be the set of all possible $I \times J$ boolean matrices. Given $q \in Q$ and $k \in K$, let q_n denote the n^{th} row of q, and k_n be the n^{th} bit of k, a hash function $h_q : K \to V$:*

$$h_q = (k_0 \cdot q_0) \oplus (k_1 \cdot q_1) \oplus ... \oplus (k_{I-1} \cdot q_{I-1})$$

Algorithm 1 Operations at each stage

Variables:

$HashFunction$ = The hash function used in the architecture.
$Input$ = Input $key - value$ pair.
$Operation$ = Operation performed on the input.
$HashAddress$ = Pointer to the hash table entry
I = Index of the I^{th} slot at $HashAddress$
$MatchSlot$ = The slot where the matching key resides
$MatchExists$ = The indicator of an existing match
$ValidVector$ = The bitvector recording validity of the slots

Stage 1: Hashing

1: $HashAddress$ = $HashFunction(Inpput.Key)$
2: Retrieve all the hash items at $HashAddress$

Stage 2: Examine

1: Initialize $MatchExists, ValidVectors$
2: **for all** Slots at $HashAddress$ **do**
3: **if** $HashAddress[I].Valid == 1$ **then**
4: $ValidVector[I] <= 1$
5: **if** $HashAddress[I].Key == Input.Key$ **then**
6: $MatchExists <= 1$
7: $MatchSlot <= I$
8: **end if**
9: **end if**
10: **end for**

Stage 3: Execute

1: **if** $MatchExists == 1$ **then**
2: **if** $Opereation == insert$ **then**
3: $HashAddress[I].Value == Input.Value$
4: **end if**
5: **if** $Opereation == delete$ **then**
6: $HashAddress[I].Valid <= 0$
7: /* for data forwarding */
8: $ValidVector[I] <= 0$
9: **end if**
10: **if** $Opereation == search$ **then**
11: Output $HashAddress[I].Value$
12: **end if**
13: **else**
14: **if** $Opereation == insert$ **then**
15: **for all** I at $HashAddress$ **do**
16: **if** $ValidVector[I]$ is the first zero in $ValidVector$ **then**
17: $HashAddress[I] <= Input$
18: /* for data forwarding */
19: $ValidVector[I] <= 1$
20: **end if**
21: **end for**
22: **end if**
23: **end if**

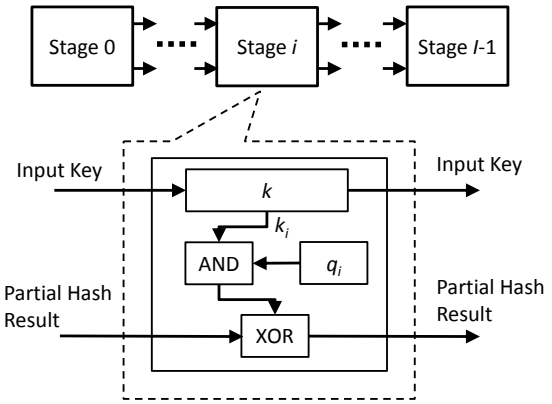


Figure 2: Pipelined Hash Computation

· denotes bitwise AND operation, ⊕ denotes bitwise XOR operation. The set $\{h_q|q \in Q\}$ is called class H_3.

By carefully choosing q, the optimal hash function of class H_3 for the target application can be produced. The H_3 is also suitable for high throughput hardware implementation as it only requires simple bitwise AND and XOR operations. According to Definition 1, to map a I-bit key to a J-bit hash value, we need $I \times J$ AND operations and I XOR operations. This is on the order of O($I \times J$). To improve the throughput of hardware implementation, we can pipeline the hash function as illustrated in Figure 2. We can use O(I) stages, each stage containing O(J) AND operations and O(1) XOR operations. Among these operations in each stage, AND operations can be performed in parallel. Therefore, O(1) latency in each stage can be achieved, which leads to a high throughput. The rows of the matrix q can be stored in distributed ram for fast access. The hash computing pipeline is before the memory access, therefore, it does not introduce any read-after-write hazards

C. Data Forwarding Unit

When $insert$ or $delete$ is performed, the hash table is updated. We assume that the latency of the Examine and the Execute stage is R and W cycles respectively (R and W can be any positive integer). Thus, an update takes $R+W$ cycles to complete. Since the architecture takes in one input every clock cycle, a read-after-write data hazard occurs if an input enters the pipeline before the previous update on the same hash item has been completed.

One trivial method to handle read-after-write hazard is to stall the pipeline for $R + W$ cycles. But this significantly deteriorates the throughput if many consecutive updates need to be performed on the same hash item. To handle the read-after-write data hazards without deteriorating the throughput, we need to provide the necessary up-to-date data to the Execute stage without stalling the pipeline.

To achieve this goal, we design a Data Forwarding

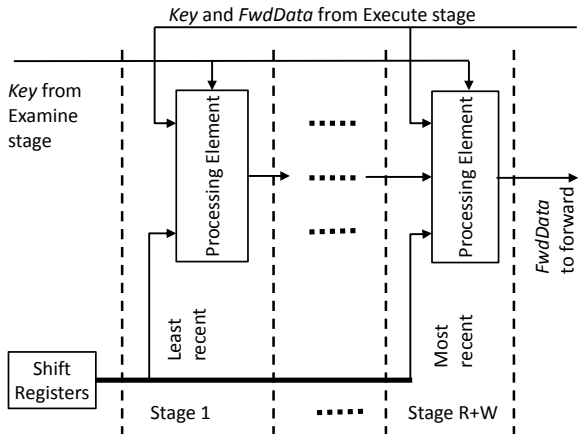


Figure 3: Pipelined data forwarding unit

Unit (DFU). According to Algorithm 1, the necessary data contains the $ValidVector$, $MatchSlot$, $MatchExist$, and $Operation$ of the hash table entry being accessed. We denote this set of data as $FwdData$. Since it takes $R+W$ cycles to complete an update, we need to keep track of the $R+W$ operations previous to the current operation. If the updates on the current input key are observed in these operations, we forward the most recent $FwdData$ to the Execute stage to replace the outdated data from the Examine stage. Assuming that the memory can support a read and a write operations in each cycle, an architecture that ensures that the throughput is not adversely affected by the read-after-write data hazards is shown in Figure 3.

The DFU has shift registers of size $R+W$ to store the keys and their associated $FwdData$. During each clock cycle, the $FwdData$ and the key at the Execute stage are pushed into the shift registers and the oldest element is removed.

To detect if a data hazard occurs, the input key to the Execute stage is compared with all the keys stored in the shift register. If one or multiple matching keys are found, the DFU forwards the $FwdData$ of the most recent matching key.

The pipelined architecture gives higher priority to the more recent keys and $FwdData$s. Each stage processes the following data:

- One pair of key and $FwdData$ from the shift register. The more recent pairs are processed at the stages closer to the output of the pipeline.
- The key and $FwdData$ at the Execute stage, because when a key enters the pipelined DFU, the partial results and the key being processed by the DFU are not available in the shift register.
- The key and $FwdData$ from the previous stage

This data allocation ensures, at any stage, the data being processed is always more recent than the data from an earlier stage. Therefore, at each stage, by locating the most recent matching key among the 3 inputs, the processing element can always output the $FwdData$ of the most recent matching key by that stage. Thus, if matching keys exists, the output of the last stage is the most recent $FwdData$ of that key. The operations of the processing elements are shown in Algorithm 2.

Algorithm 2 Operations of the processing element of the pipelined DFU

Variables:
key_{EX}/key_{Shift} = key from the Execute stage/shift registers.
$FwdData_{EX}/FwdData_{Shift}/FwdData_{Prev}$ = $FwdData$ from the Execute stage/shift registers/forwarded from the pevious stage.
$keyIn$ = Input key.

```
1: if FwdData_Prev is from the shift registers then
2:   if key_EX == key_In || key_Shift == key_In then
3:     if Exactly one of key_EX, key_Shift matches key_In then
4:       Output the partial result of the matching key
5:     else
6:       Output FwdData_EX /*- - FwdData_EX has the highest priority - -*/
7:     end if
8:   else
9:     Output FwdData_Prev
10:  end if
11: else
12:   if key_EX == key_In then
13:     Output FwdData_EX
14:   else
15:     Output FwdData_Prev
16:   end if
17: end if
```

D. Supporting Low Memory Bandwidth

The operations shown in Algorithm 1 use the first fit hash table access scheme. That is: we examine the slots one after another and always perform the operations to the first slot that meets the requirements. (For example, when performing insertion, if there are multiple empty slots, we insert the new $key-value$ pair to the first one we encounter). This scheme guarantees that no existing keys can be missed in a search operation and no duplicated data can exist in the hash table. The hash collision only occurs when all the slots of a hash entry are occupied and a new key needs to be inserted in the hash entry. Since the H_3 hash function has a low hash collision rate, the chance that multiple keys are mapped to the same hash entry is little, resulting in a low collision rate for the first fit hash table access scheme. However, since this scheme needs the data from all the slots of the accessed hash

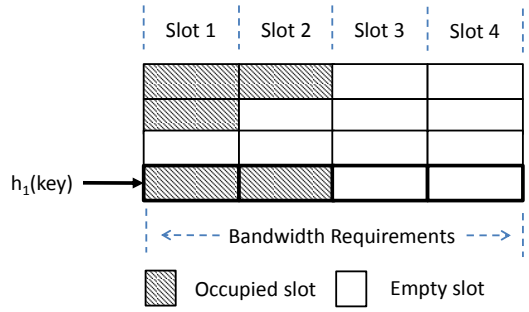
entry to decide which is the right slot to operate. Since we need to perform line rate processing, all the slot data needs to be retrieved in one memory access. This requires a high memory bandwidth. If the DRAM device cannot provide such a high bandwidth, the performance of the architecture will be significantly degraded.

To boost the throughput while working with limited memory bandwidth, we designed a random fit hash table access scheme. In addition to the hash function deciding which hash table entry to access, we use a second hash function to decide which slot in the hash table entry to perform the operations. Figure 4 shows a comparison between the first fit scheme and the random fit scheme along with their bandwidth consumption. We can see that for the same hash table configuration, the random fit scheme needs much less bandwidth than the first fit scheme per access. This means more accesses per unit time and it leads to a better throughput. As long as the bandwidth is enough to bring one slot data per memory access, the line rate processing can be guaranteed. This is a much lower requirement than bringing all the slots in one memory access.

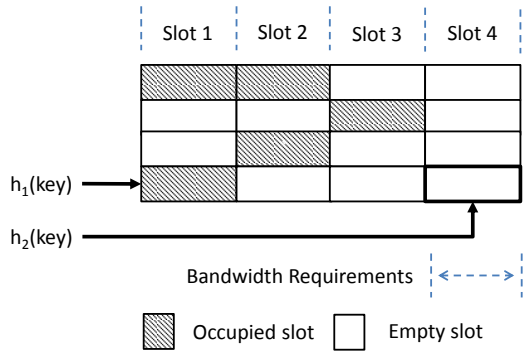
Like the first fit scheme, the random fit scheme can also guarantee no missed search for any existing keys and no duplicated data in the hash table. This is because a given key is always hashed to the same slot in the same entry. If a key exists in the hash table, it can only exist in that slot leaving no chances for missed search or duplicated data.

In addition to lower bandwidth requirement, the random access scheme also needs less complex logic than the first fit scheme. When implementing first access scheme we need a module to decide the right slot to perform the operations. This module is not necessary when implementing random access scheme. Less complex logic results in less logic slices consumption and a better routing on FPGA. This further enhances the throughput of the architecture with the random access scheme, as demonstrated in Section V.

Compared with the first fit access scheme, the random fit scheme has a larger hash collision rate. It is possible that an insertion is assigned to a slot which has been occupied by a different key. At this point even if there are other empty slots in the hash entry, this insertion can not take place and the input key is discarded. This results in that a lower utilization of the hash table than the first access scheme. But the collision only occurs when two different keys are mapped to the same slot of the same entry, namely having hash collisions for both hash functions. Considering the low collision rate of the H_3 hash functions, the collision rate of the random fit scheme is still very low. Moreover, in a normal network traffic where packets are from various users, the probability of a collision is even lower. Therefore we can achieve a great improvement in throughput with a small trade off in hash table utilization when working with limited memory access bandwidth.



(a) First Fit Scheme



(b) Random Fit Scheme

Figure 4: Two Hash Table Access schemes

E. Supporting Multiple Operations

Figure 5 shows the architecture of the execution stage to support multiple functions at a high throughput. We assign one processing element to update each slot. In each processing element, updated slot data for all three operations are generated in parallel. Then we select the appropriate updated slot data for forwarding based on the operation to the input key. A slot selection module is designed to generate a one-hot slot selection signal to the second multiplexers in the processing elements. According to the comparison results and the valid vector from the examine stage, it sends a "1" to the processing element for the target slot, indicating that the processing element forwards the updated slot data. To all the other processing elements, the slot selection module sends a "0" to let them forward the original slot data. The outputs of these processing elements are written back to the external DRAM.

The number of the processing elements is decided by the hash table access schemes. If we use first fit access scheme, we need as many processing element as the number of slots per hash table entry. If we use random access, we only need one processing element and the slot selection module can also be eliminated.

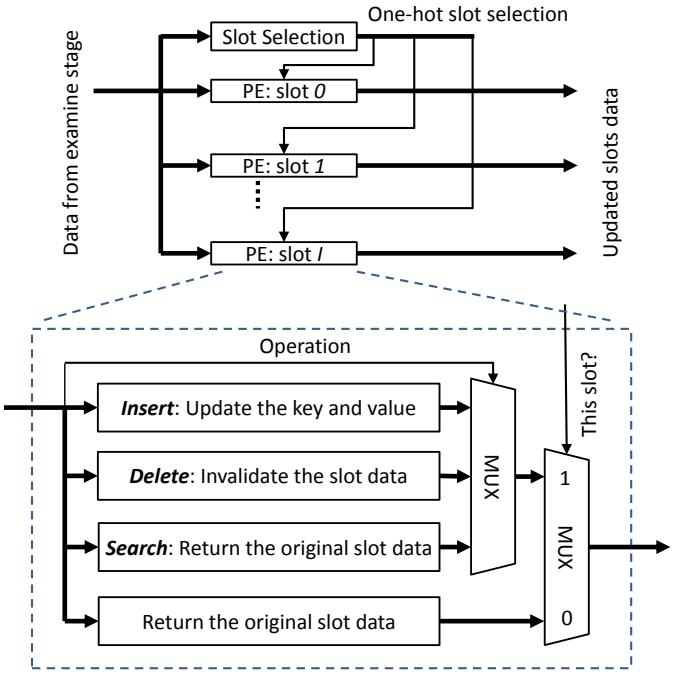
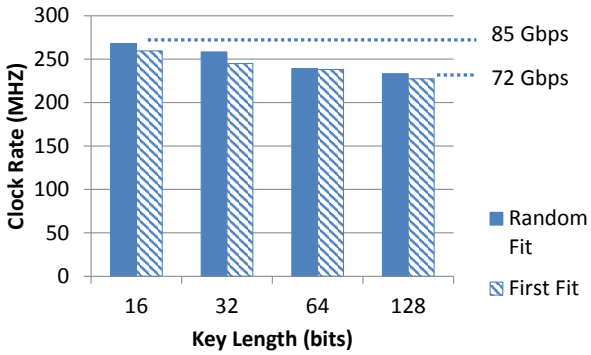


Figure 5: Supporting Multiple Operations



(1M hash table entries, 4 slots per entry)

Figure 6: Performance of baseline hash table configurations

Table I: Bandwidth requirement for baseline configuration

Key length (bits)	16	32	64	128
First Fit (Gbps)	25.94	40.18	69.54	127.63
Random Fit (Gbps)	6.70	10.59	17.46	31.95

V. Evaluation

A. Experimental Setup

We implement the proposed design on FPGA. Our target device is Xilinx Virtex 7 XC7VX1140T with -2 speed grade. All reported results are post place and route results using Xilinx Vivado 2014.3. We evaluate the performance by varying the DRAM access bandwidth and latency value to demonstrate that our architecture achieves high throughput for various DRAM devices.

The key sizes we use in our experiments are 16, 32, 64 and 128 bits. These numbers cover most key sizes in network applications (e.g. Port numbers, IPv4 and IPv6 address, 5 field packet classification and traffic classification, etc) and they also cover a sufficiently wide range to test the scalability of our architecture. We test the logic resource consumption and the throughput of our architecture for various key sizes and vary the number of hash table entries and slots per entry to test the scalability of our architecture.

Logic resources on FPGA are organized as slices. So the number of occupied slices reflects the logic resource consumption of our architecture. In our experiments, the utilization of slices never exceeds 4% of the total available number. Therefore in the following sections, we focus on reporting the throughput of our architecture using various configurations.

When computing the throughput, we use the minimum IP packet size, 40 bytes. Our architecture can take in one packet per clock cycle. Therefore, the throughput can be computed by multiplying the clock rate with the minimum packet size. Since our architecture is designed as a general architecture to work with various DRAM devices, we also show the minimum bandwidth requirement for our architecture to operate at its highest possible clock rate. To calculate the bandwidth requirement, we multiply the operation clock rate with the amount of data exchanged between FPGA and the DRAM device.

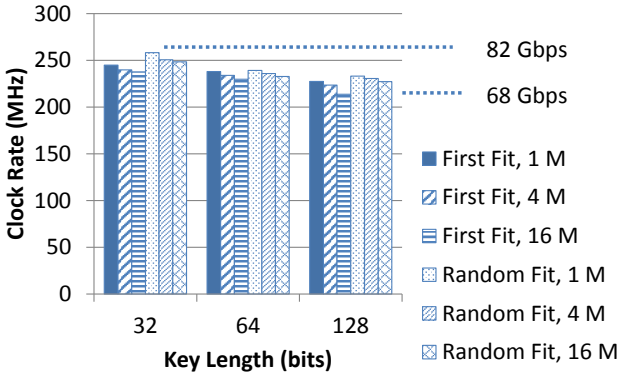
B. Typical Hash Table Configuration

Figure 6 shows the clock rate and the throughput of our architecture for a baseline hash table configuration. The architecture achieves very steady performance for all tested hash key lengths. The throughput lies between 72 - 85 Gbps. We can tell from the figure that as we have discussed in Section IV-D, due to less complex logic, the design using random fit scheme achieves a higher throughput than the one using first fit scheme. Table I shows the memory bandwidth requirement for the corresponding configuration. As discussed in Section IV-D, the random fit scheme needs much less memory bandwidth than the first fit scheme.

C. Scalability

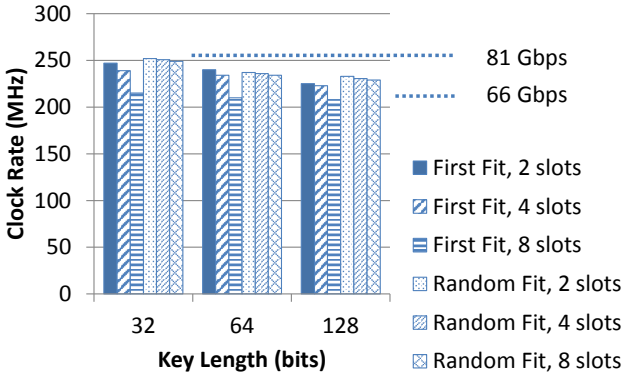
Section V-C and V-D test the scalability of our architecture. Since key size of 16 bits is too small to effectively demonstrate the scalability, we remove it from the experiments in these two sections.

Figure 7 shows the performance of our architecture for various numbers of hash table entries. We observe that for both random fit scheme and first fit scheme, the clock rate does not vary much as the number of hash table entries grows significantly from 1 M to 16 M. This is because only 4 more bits in the hash value are required to increase the number of hash table entries from 1 M to 16 M. This 4-bit



Memory Bandwidth Reqirement (Gbps):
First Fit: 38 - 124, Random Fit: 10 - 31
(4 slots per entry)

Figure 7: Various Number of Hash Table Entries



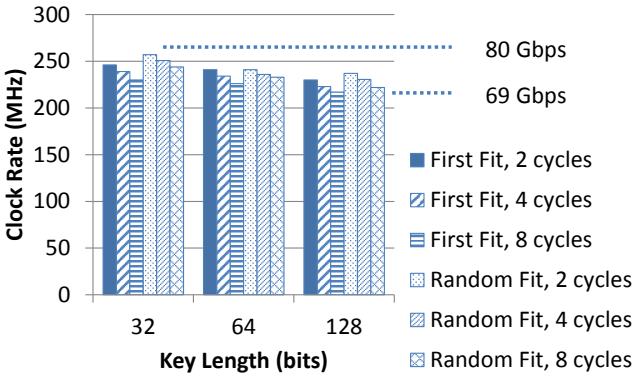
Memory Bandwidth Reqirement (Gbps):
First Fit: 17 - 252, Random Fit: 10 - 31
(4 M hash table entries)

Figure 8: Various Number of Slots per Entry

lays little impact on the logic consumption and the routing of the implementation. Therefore, the architecture demonstrates very good scalability when the number of hash table entries varies.

Figure 8 shows the performance of our architecture for various numbers of slots per hash table entry. We observe that the two hash table access schemes show different performance with respect to scalability in the experiment. For the first fit scheme: although the architecture still achieves a high clock rate, the clock rate drops significantly when we increase the number of slots per entry from 2 to 8. For the random fit scheme: the clock rate varies little as the number of slots grows. Such difference with respect to scalability is due to:

- When we use first fit scheme, we need to process all the slots in parallel. The logic consumption is proportional to the number of slots per hash table entry. When the number of slots grows, along with the increasing logic consumption, the pressure on routing also increases. Therefore the clock rate drops fast.
- When we use random fit scheme, given a fixed number of entries in the hash table, the number of slots per entry only affects the hash value width of the second hash function. When the number of slots increases from 2 to 8, the hash value width only varies by 3 bits, which has little effect on the implementation. Therefore, the random access scheme shows a better scalability than the first fit scheme for various numbers of slots per hash table entry.



Memory Bandwidth Reqirement (Gbps):
First Fit: 37 - 126, Random Fit: 10 - 30
(4 M hash table entries, 4 slots per entry)

Figure 9: Various Memory Access Latency

D. DRAM Access Latency

Depending on the DRAM devices and memory interface, DRAM access may take various number of clock cycles. As discussed in Section IV-C, the memory access latency affects the number of stages of the data forwarding unit. Figure 9 shows the clock rate of our proposed architecture for various memory access latency. Since our DFU is fully pipelined, increasing the number of stages from 2 to 8 doesn't affect the performance by a lot. As a result, both architectures demonstrate good scalability when working with various memory access latency.

E. DRAM Devices

As the results shown in Section V-B, V-C, and V-D our architecture can work at a very high clock rate. DRAM provides very large storage capacity. However, due to frequent refreshing and the row activation delay [13], a direct mapping of our hash table to a normal DRAM device (e.g. DDR3 DRAM) cannot keep up with the working frequency of our proposed architecture. In [13], Qu *et al.* proposed a data replication technique and boost up the DRAM access rate to 200 M accesses per second. The technique replicates

the data across all the DRAM banks and access the banks in a round robin manner, so that the row activation delay can be overlapped with the fetching delay. Therefore, it can only accelerate the *search* operations, because *delete* and *insert* need to be performed to all the banks in the DRAM. This technique can be applied when our architecture is used to implement a static hash table (which means normally only $search$ operations are performed, for example an IP lookup engine without dynamic update).

In [6], the experiments are conducted on a Maxeler Workstation. The target platform is equipped with a 24 GB DDR3 DRAM which can be accessed in 384-bit words at 300 MHz with a burst size of 8. This platform provides a memory bandwidth of 115 Gbps. Comparing with the bandwidth requirement shown in Section V-B, V-C and V-D, this bandwidth is large enough to support our experimented configurations.

VI. Conclusion

In this paper, we proposed a pipelined architecture for a high throughput online hash table on FPGA. It can be applied to accelerate various network applications. Our architecture supported fixed-length keys and values using external DRAM. It sustained 72-85 Gbps throughput for typical hash table configurations due to the careful design in hash function, hash table access, and data forwarding unit. Based on the DRAM bandwidth requirement, we proposed two hash table access schemes, first fit scheme and random fit scheme. The first fit scheme examined multiple slots of the corresponding hash entry, therefore having a high memory bandwidth requirement; the random fit scheme reduced the memory bandwidth requirement and logic resources by examining only one slot of the corresponding hash entry, but the hash collision rate is higher than first fit scheme. Both schemes demonstrated good scalability when the size of hash table increased.

As future work, we will extend our architecture to support larger scale hash table with broader range of key and value sizes. The scope of our work will be expanded to big data application under the data center context, for example, to accelerate MemcacheD [9] system.

References

[1] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.jalgor.2003.12.001

[2] V. Puš and J. Korenek, "Fast and scalable packet classification using perfect hash functions," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 229–236. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508163

[3] "FP3: Breakthrough 400G network processor," http://www.alcatel-lucent.com/fp3/.

[4] M. Attig and G. Brebner, "400 gb/s programmable packet parsing on a single fpga," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, Oct 2011, pp. 12–23.

[5] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 323–336, Aug. 2002.

[6] Z. Istvan, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–8.

[7] Y. Qu and V. Prasanna, "Compact hash tables for high-performance traffic classification on multi-core processors," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, Oct 2014, pp. 17–24.

[8] M. Bando, N. S. Artan, and H. J. Chao, "Flashlook: 100-gbps hash-tuned route lookup architecture," in *Proceedings of the 15th International Conference on High Performance Switching and Routing*, ser. HPSR'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 14–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1715730.1715733

[9] "Free and open source, high-performance, distributed memory object caching system," 2013.

[10] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '77. New York, NY, USA: ACM, 1977, pp. 106–112. [Online]. Available: http://doi.acm.org/10.1145/800105.803400

[11] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *Computers, IEEE Transactions on*, vol. 46, no. 12, pp. 1378–1381, Dec 1997.

[12] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 323–336, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/964725.633056

[13] Y. Qu and V. Prasanna, "High-performance pipelined architecture for tree-based ip lookup engine on fpga," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 114–123.

[14] W. Jiang and V. Prasanna, "Data structure optimization for power- efficient ip lookup architectures," *Computers, IEEE Transactions on*, vol. 62, no. 11, pp. 2169–2182, Nov 2013.