An FPGA Implementation of High-Throughput Key-Value Store Using Bloom Filter

Jae Min Cho
Department of Electrical and Computer Engineering
Seoul National University
Seoul, South Korea
jaemincho@dal.snu.ac.kr

Kiyoung Choi
Department of Electrical and Computer Engineering
Seoul National University
Seoul, South Korea
kchoi@snu.ac.kr

Abstract—**This paper presents an efficient implementation of key-value store using Bloom filters on FPGA. Bloom filters are used to reduce the number of unnecessary accesses to the hash tables, thereby improving the performance. Additionally, for better hash table utilization, we use a modified cuckoo hashing algorithm for the implementation. They are implemented in FPGA to further improve the performance. Experimental results show significant performance improvement over existing approaches.**

Keywords—FPGA, Bloom filter, Key-value Store

1. Introduction

Key-value (KV) store has been used to implement data deduplication, on-line multi-player gaming, and Internet services. It is increasing its usage in most web-applications due to its scalability better than that of traditional relational databases. KV store is now taking a large portion of Non-SQL database [1]. It is used in large websites including Amazon [2], Facebook [3], and Twitter [4]. All these systems store ordered KV pairs and maintain distributed hash tables. Such a system has some important features. For example, retrieving data from the KV store can be much faster than retrieving it as a result of database queries or as a result of complex computations that require temporary storage. Moreover, due to the use of distributed hash tables, KV store has better scalability than traditional databases.

Due to large number of KV pairs, however, it requires a large set of hash tables and makes it difficult to keep the entire hash tables in a main memory. Thus, the hash tables are typically placed in a secondary storage. To maximize the benefit of fast data-retrieval, KV store tries to keep the data in the main memory as mush as possible to avoid slow I/O operations. Most of the web appliances, such as Memcache-D [5], Berkeley-DB [6], keep data exclusively in main memory.

Also, KV stores are generally network-enabled, permitting the sharing of information across the machine boundary and offering the functionality of large-scale distributed shared memory without the need for specialized hardware. This aspect is critical for large-scale web sites, where the sheer data size and number of queries on it far exceed the capacity of any single server. Such large-data workloads can be I/O intensive and have no specific access patterns. Thus large websites such as Facebook are using larger and larger clusters and scaled Memcache-D.

However, as these clusters grow larger, their operating cost becomes larger. One of the largest components of this cost is electric power consumed in processors, RAM, storage etc. And the cost of power servers of the data centers takes up to 50% of the three-year total ownership cost (TCO) [7].

One of the solutions could be using low-power CPUs or reconfigurable architectures. For the reconfigurable architectures such as FPGA, their relatively slow clock speed could downgrade performance. However, the time taken by KV store operations is typically very small and ignorable compared to the entire network latency. Moreover, FPGA implementations can be made even faster by exploiting parallelism, while maintaining low power consumption.

One advantage of implementing KV store operations in FPGA rather than software is the reduction of network layer stack. Since the network transport layer such as TCP or UDP is typically controlled by OS, it could be a bottleneck of the whole application. By implementing it into FPGA, however, we can have tightly integrated networking with better performance [8]. An existing approach that implements memcache-D in FPGA reduces power consumption by up to 9% compared to a standard server [9].

In this paper, we further improve the performance by using a Bloom filter in the KV store. Because the Bloom filter reduces the number of unnecessary accesses to the hash tables, it can reduce the amount of memory transactions. Bloom filters have been used in KV stores but their implementations are all in software [10][11][12]. We propose in this paper to implement them also in FPGA to further improve the performance. We also implement cuckoo hashing in FPGA to resolve possible collisions of hash function values in the hash tables.

2. Bloom filter

Bloom filters are used in various systems such as web caches[13], database systems[14], and peer-to-peer systems[15]. Their merit is in providing a space-efficient

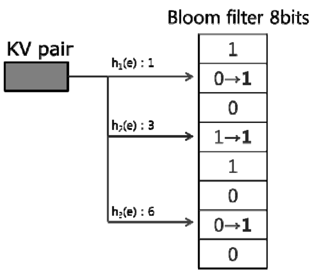

Fig. 1. Example of setting bits in a Bloom filter.

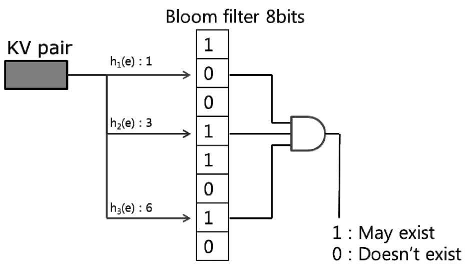

Fig. 2. Example of checking bits in a Bloom filter.

storage. However, one major shortcoming is that they have non-zero probability of false positive error on accessing queries. False positive error means that actual element is not in the set under search but the filter considers that the element is in the set. There are some other important properties in Bloom filter. There is linear relationship between the filter size and the number of elements that can be stored. And there is also relationship between false positive probability and the number of hash functions used in the filter.

A conventional Bloom filter is implemented by a single array of M bits, where M is the filter size. All the bits in the array are initialized to zero. The filter is also parameterized by a constant k that defines the number of hash functions used to change bits in the array. When inserting an element into the set, each of the k hash functions generates a hash value used as an array index. That is, for an element e, k indexes $h_1(e)$, $h_2(e)$, …, $h_k(e)$ are generated and the corresponding array positions are set to one. Fig. 1 shows an example of setting bits in a Bloom filter with an array of eight bits (M = 8) and three hash functions (k = 3).

Hash bits change only from zero to one, except for counting Bloom filter. When querying for an element, in order to get information about the existence of an element in the set, it is sufficient to see if all the bits in the array positions $h_1(e)$, $h_2(e)$, …, $h_k(e)$ are one. If one or more of these bits are not one, then the queried element is not present in the set. If all the indexed bits are one, then the element is considered to be in the set. But, there is a probability of false positive match. Fig. 2 shows using our running example how to check a Bloom filter for the existence of an element in the set.

A variant of Bloom filter [16], which we adopt in this paper, partitions the array of M bits into k slices with m = M/k bits each and assign a hash function to each of the slices. In this variant, each hash function produces an index over m bits in its respective slice. Therefore, each element is always

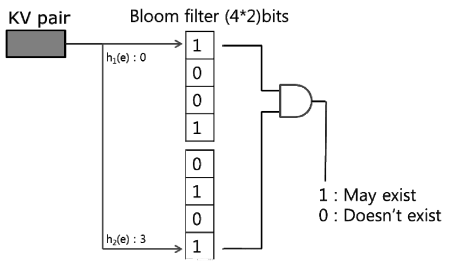

Fig. 3. Example of variant Bloom filter.

represented by exactly k bits, which results in a more robust filter (i.e., no element is specially sensitive to false positives) [16].

Fig. 3 is an example of the variant of Bloom filter. Let's assume an array of eight bits (M = 8) and two hash functions (k = 2). Then the number of slices (m) would be 4. Then $h_1(e)$ sets only the 1st slice, and $h_2(e)$ sets only the 2nd slice.

The false positive probability P of the scalable Bloom filter is given by

$$P = p^k$$

where p is the fill ratio between the number of set bits in the slice and the slice size m. The value of p increases linearly with the number of elements inserted into the set, and thus P also increase. In other words, to maintain P under a given upper bound, the number of elements in the set should not exeed a limit N. It has been shown in [16] that the maximum of N for a given P is obtained by

$$N \fallingdotseq \frac{M(ln2)^2}{|lnP|}$$

which occurs at p=1/2 or $P = p^k = (1/2)^k$, that is, $k=\log_2(1/P)$. Table 1 shows the numbers of N obtained by changing parameter P with fixed Bloom filter size of 4KB. For example, to maintain false positive error rate of 0.1%, we can store up to 2279 elements (or KV pairs).

Table 1. Bloom filter parameters

P	0.1%	0.01%	0.001%	0.0001%
k	10	14	17	20
m	3227	2340	1927	1638
N	2279	1709	1367	1139

3. Cuckoo hashing

Cuckoo hashing is a scheme that resolves collisons of hash function values in a hash table with worst-case constant lookup time [17]. Thus, many of the KV store implementation uses cuckoo hashing to fully utilize hash table entries. But cuckoo hashing has some problem when implemented in hardware or FPGA, because it has a recovery process when there is no candidate slots. Before explaining the recovery process of the cuckoo hashing, we briefly explain how the cuckoo hashing works.

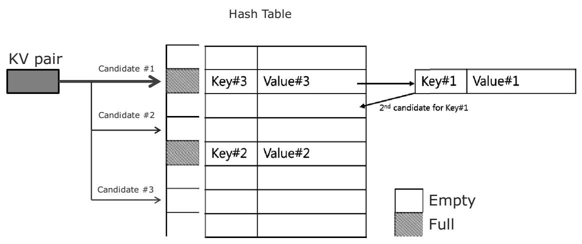

Fig. 4. Example of basic cuckoo hashing.

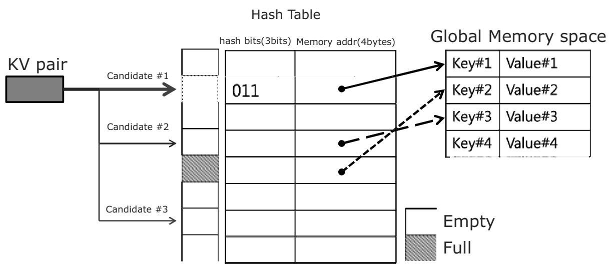

Fig. 5. Our proposed approach to cuckoo hashing implementation.

Fig. 4 shows an example of the basic cuckoo hashing for an insert operation. To insert a new KV pair into a hash table, multiple hash functions are used to calculate multiple candidate locations. In our example, we first check the hash table to see if the first candidate slot is empty. Since it is not empty, we kick out the existing KV pair and put the new KV pair into the first candidate slot. Then the KV pair kicked out from that location is placed into its own candidate location that is empty. It may cause a long sequence of replacement operations and even a loop, which can be resolved by rehashing when the fill ratio (load factor) is not high. since the first candidate slot is full but the second candidate slot is empty. The Bloom filter already calculated the hash values, and so we could use those values for the candidate slots. Usually the number of candidate slots are fixed as in our example, where we have three candidate slots.This basic cuckoo hashing implementation is used by various KV stores such as BloomStore[11], FlashStore[18], etc. However, maintaining the KV pair directly in the hash tables has some drawbacks. First, when the size of KV pairs varies, it is hard to be implemented in hardware. Secondly, when the hash tables are implemented in a distrbuted system, it could be a bottleneck to move around large sized hash tables. A solution to these problems could be letting the hash table provide only the addresses of the KV pairs. This leads to reduced usage of SRAM cache size, easy implemention in hardware, and better scalability of the distributed system. Fig. 5 shows the implementation of our proposed approach.

In order to separate KV pairs from the hash table, we need to have some signature in the hash table. The basic cuckoo hashing uses the key as a signature as shown in Fig. 4. In our approach, however, since the KV pairs are no longer in the table, we need another form of signature to see if we have found the matching KV pair. For this, we use the calculated hash values. Fig. 5 shows how to put the signature into the hash table. We use the hash value of the next candidate

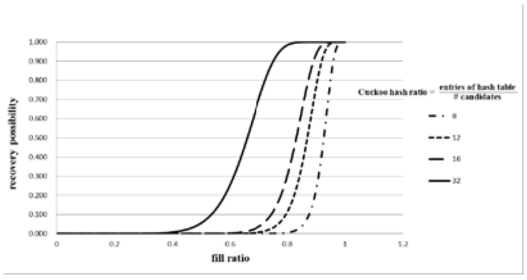

Fig. 6. Recovery possibility as the fill ratio in the hash table.

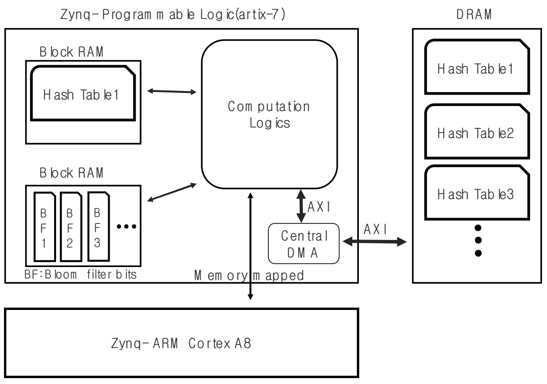

Fig. 7. Our proposed implementation in Zynq.

location as the signature. In our example, the hash value for the next location (Candidate #2) is 3 (011), it is used as the signature to be put into the hash table (at the first candidate location). The signature calculation wraps around (i.e., the first hash value becomes the third signature).

When we lookup the proposed hash table, there is a possibility of alias. This alias possibility is inversely proportional to the number of hash bits. We consider this collison penalty in the experiment result. Also there is a possiblity that there are no candidate slots in the hash table, which requires recovery (rehash). The recovery can be done easily by software but it may not be easy for the hardware approach. In that case, we take the software approach for the recovery, which takes long time. So we need to have a very low recovery possibility to implement it in hardware.

Fig. 6 shows the relationship between fill ratio of cuckoo hash table and recovery possibility. The fill ratio is defined as the number of elements stored in the hash table divided by the number of entries in the hash table. The figure shows that if cuckoo hash ratio increases, then we could store more elements while maintaining the same recovery possibility. But increasing cuckoo hash ratio means adding additional logic resources.

4. Experimental results

Our proposed design was implemented in Xilinx Zynq-7020 FPGA [19]. It had two ARM Cortex-A9 processor cores and FPGA fabric and we used 1GHz clock and 100MHz clock, respectively. Fig. 7 shows the diagram of our implementation. BFs denotes Bloom filter bits, all of which fit in one block RAM in the FPGA. Bloom filter summarizes the contents of

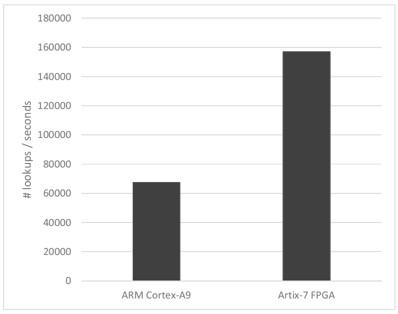

Fig. 8. Performance evaluation on Zynq.

the corresponding hash tables and this allows us to reduce the number of DRAM accesses.

Inside Computation Logics in Fig. 7, there are numerous functional modules such as hash value calculation logic, Bloom filter comparsion logic, block RAM replacement logic, modified cuckoo hashing logic, etc. And thus we can take full advantage of parallel operations inorder to maximize the performance. For the implementation, we used 13% of the BRAMs and 14% of the LUTs in Zynq FPGA. We used hash table size of 4KB, which has capacity of 512 entries.

We compared our proposed design with a software implementation on the ARM Cortex-A9 processor cores in Zynq by using workloads obtained from 100,000 uniform random generated KV pairs. For the sizes of each key and value, we used 32bytes and 64byte, respectively.

Fig. 8 Show the result of comparing software only implemenatation and FPGA implemenatation. For the DRAM access, the ARM processors and FPGA have relatively the same performance because FPGA uses Central DMA logic. So, this result shows us that FPGA uses its parallel operations such as Bloom filter calculations, hash value calculations, and hash table lookups.

We also had an experiment to examine the relationship between number of candidate locations and fill ratio in the hash table. As shown in Fig. 9, if the number of candidate locations is 32 which corresponds to cuckoo hash ratio of 8, it could fill the hash table by up to 90% without recovery process. This corresponds to the theorical result of Fig. 6.

5. Conclusion

In this paper, we designed a KV store based on Bloom filter and cuckoo hashing, and implemented it in FPGA. Our main contribution in this work is that we are the first to implement the Bloom filter based KV store in hardware and proposed modified cuckoo hashing that fits well into hardware. Our proposed approach shows about 2.3x higher speed compared to software only approach. This is mainly due to exploiting the parallelism through hardware implementation. Our future work would be measuring the actual power consumption dynamically while the application is running.

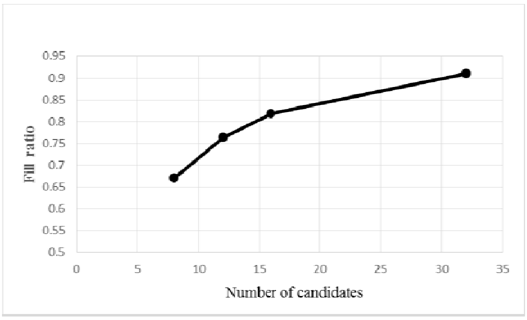

Fig. 9. Fill ratio as the number of candidates.

6. Reference

[1] non sql. https://en.wikipedia.org/wiki/NoSQL

[2] Amazon, http://www.amazon.com

[3] Facebook, http://www.facebook.com

[4] Twitter, http://www.twitter.com

[5] Memcache-D, http://www.memcached.org

[6] Berkeley DB, http://en.wikipedia.org/wiki/Berkeley_DB

[7] C. Patel and A. Shah., “Cost model for planning, Development and operation of a data center. *Hewlett*-Packard Laboratories Technical Report,” 2005

[8] M. Berezecki and K. Steele, “Many core key-value store,” Green Computing Conference and Workshops (IGCC), 2011 International

[9] S. Chalamalasetti, K. Lim, M. Margala, “An FPGA Memcached appliance,” international symposium on Field programmable gate arrays, 2013

[10] Broder, A., Mitzenmacher, M. “Network applications of Bloom filters: a survey,” Allerton 2002

[11] G. Lu, Y. Nam, and D. Du, “BloomStore: Bloom-Filter based memory-efficient key-value store for indexing of data deduplication on Flash,” Mass Storage Systems and Technologies, 2012

[12] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, “The Bloomier filter: an efficient data structure for static support lookup tables,” Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, 2004

[13] L. Fan, P. Cao, J. Almeida, A. Z. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” IEEE/ACM Trans. Netw. 8, 2000, pp. 281–293

[14] L. F. Mackert, G. M. Lohman, “R* optimizer validation and performance evaluation for distributed queries,” Proceedings of the Twelfth International Conference on Very Large DataBases(VLDB), Morgan Kaufmann Publishers Inc., USA, 1986, pp. 149–159

[15] P. Reynolds and A. Vahdat, “Efficient peer-to-peer keyword searching,” M. Endler, D. C. Schmidt (Eds.), Middleware, Vol. 2672 of Lecture Notes in Computer Science, Springer, 2003, pp. 21–40

[16] P. Almeida, C. Baquero, and D. Hutchison, “Scalable Bloom Filters,” Information Processing Letters Vol. 101 Issue 6, Mar 2007

[17] Pagh and Rodler, “Cuckoo Hashing,” ALGORITHMS: Journal of Algorithms, vol. 51, 2004

[18] B. Debnath, S. Sengupta, and J. Li, “FlashStore: high throughput persistent key-value store,” VLDB Endowment VLDB Endowment Hompage archive, Sep 2010

[19] Xilinx Zynq, http://www.xilinx.com/products/silicon-devices/soc