Deterministic and Efficient Hash Table Lookup Using Discriminated Vectors

Dagang Li, Junmao Li
School of Electronics and Computer Engineering
Peking University
Shenzhen, China

Zheng Du
National Supercomputing Center in Shenzhen
Shenzhen, China

***Abstract*—Hash table is used in many areas of networking such as route lookup, packet classification, per-flow state management and network monitoring for its constant access time latency at moderate loads. However, collisions may become frequent at high loads in traditional hash tables, which may lead the access time complexity to be linear and intolerable to applications like high-speed route lookups. While some schemes were proposed to help resolve this problem and most of them may achieve *O*(1) average memory access per lookup, very few of them are able to cut down the access to a deterministic single one. In this paper, we design a structure called deterministic and efficient hash table (DEHT). In DEHT, a novel data structure on on-chip memory is built, with the help of which the off-chip memory access can be decreased to a single one at most per lookup even when the load of the hash table is very high. What's more, the on-chip data structure also plays a similar role as Bloom Filter to do membership screening, which can avoid most lookups of nonexistent items of the hash table visiting the off-chip memory. Through theoretical analysis and simulations, we show that our scheme is faster than other schemes in lookup operations; the usable load of the off-chip hash table, the memory efficiency and the false positive rate of the on-chip data structure are also favorable.**

Keywords—Network processing; Collison-free hashing; Cuckoo hashing; Hash table

I. INTRODUCTION

Hash table is prevalently used in various fields of networking such as route lookup, packet classification, per-flow state management and network monitoring as it can achieve *O*(1) latency for query, insert and delete operations at moderate loads. However, measures like chaining and linear probing will have to be taken to solve the collision problem at the cost of additional memory access when the load of the hash table increases. Thus, the lookup latency becomes uncertain, which poses threats to some time-critical systems. Specifically, some network packet processing modules are typical components in the data-path of a high-speed router which must be able to process packets at very high line speed. Non-determinism also influences multi-threaded systems because the slowest thread will be the bottleneck and determines the overall throughput. So, it is very vital to keep the lookup operations fast and deterministic while not hurting the usable load of the hash table at the same time. Several schemes [1][2][3][4] utilizing embedded memory technology were proposed to help solve this problem, but there are still some difficulties unsolved in these solutions. First, though some of them achieve *O*(1) average memory access per lookup, few of them are able to perform lookup operation with a single access to the off-chip memory. Second, not all of them support highly loaded hash table. Third, few of them are able to perform membership screening in the on-chip data structure.

Cuckoo hashing [5], a representative multi-choice hashing, is famous for its high usable load ratio of hash table. The load ratio of Cuckoo hash table can be up to 0.9 without collision with 3 independent hashing functions. When performing the lookup operations, Cuckoo hashing requires one memory access for each hash function respectively to determine which one of its candidate slots actually stores the element. If we can foresee which hash function corresponds to the actual slot where the element is stored in the Cuckoo hash table, the memory access will be cut down to a single one while keeping the load ratio unchanged.

More specifically, the deterministic and efficient hash table lookup can be designed as follows. First, due to its large memory requirement, we store the actual elements in a Cuckoo hash table in the slower off-chip DRAM. Second, we may design a compact data structure in the high-speed on-chip SRAM to assist the lookup operations of the hash table. With some computation based on the on-chip data structure we may determine which hash function is actually used to determine the location of the queried element in the hash table. The on-chip data structure should be compact, simple and fast and easy to be implemented. In this paper, we design such a deterministic and efficient hash table scheme called DEHT that can minimize access to the off-chip Cuckoo hash table.

In general, we make the following contributions:

- We designed a compact data structure in small high-speed on-chip memory (e.g. on-chip SRAMs) and a Cuckoo hash table in slow off-chip DRAMs. The lookup operation requires at most one access to the off-chip DRAMs even at high loads by combining these two data structures. This feature enables the lookup operations of the hash table take deterministic latency.
- The data structure we designed on-chip is also able to play a similar role in membership screening like the bloom filter. Though some other schemes may have a similar feature, the false positive rate (*fpr*) of our on-

This work was supported by Guangdong Science & Technology Project 2014B010117007, Shenzhen Basic Research Program JCYJ20140509093817684 & JCYJ20150626110611869.

978-1-5090-1328-9/16/$31.00 ©2016 IEEE

chip structure is low enough compared to other schemes and more importantly the *fpr* will not impact the property of single access to off-chip memory.

The rest of this paper is organized as follow. We discuss the related work in Section II. The data structure and algorithm will be presented in Section III. And we make analysis of the performance of our design in Section IV. Simulations and the results are presented in Section V, while we conclude this paper at last in Section VI.

II. Related Work

Ideas that combine the use of on-chip and off-chip data structures to reduce off-chip memory access can be found in [1-7]. However, they cannot meet all the requirements of many networking applications that require fast and deterministic speed. Solution in [1] extends multiple hash technique and Bloom Filter to perform exact match. Each Element is stored into the shortest one of the off-chip linked lists, and on-chip Counting Bloom Filter is used as summary to indicate one linked list used for the search. Analysis of [1] shows that its expected external memory access can be $O(1)$. A scheme was proposed in [2] to improve the performance of method in [1]. By using off-chip multilevel hash table and an array of parallel Bloom filters, the space cost of summary and underlying hash table are both reduced compared to method in [1]. Though achieving $O(1)$ lookup speed, the latency is still not deterministic because they use chaining technique to solve the collision problems. Schemes in [3] combined on-chip summary vectors and Bloom filter to make sure only one single access to off-chip memory is needed. However, when inserting one element, some existing elements of the hash table may be moved to other slots. And the usable load ratio of the hash table can not be as high as in a standard Cuckoo hash table. A structure called FCHT based on on-chip Bloom filter and off-chip multi-choice hash table was proposed in [4]. Several Bloom filters corresponding to hash functions were built on-chip. When looking up an element, if one Bloom filter outputs a positive result, the system will use its corresponding hash function to calculate a location of the off-chip hash table. However, because of the false positive possibility of Bloom filter, when two or more on-chip Bloom filter output positive results, the off-chip memory accesses will increase. And this phenomenon will be more severe with higher number of inserted elements. Solutions in [6][7] have similar problems and can not meet all requirements that we listed above. Cuckoo hashing [5] is a simple multiple choice hashing scheme that allows elements to move to solve collision. Its space usage is similar to that of binary search trees. For insertion, each element is hashed to d possible buckets, and may kick existing elements away until every element is moved to an appropriate bucket. The insertion time complexity is $O(\log n)$ for $d=2$ and the upper bound of it have been proved more difficult for $d>=3$ [8]. The occupancy of Cuckoo hash table can be up to about half and 90% for d=2 and d=3 respectively.

III. DEHT: Algorithm and Data Structure

The data structure of DEHT is shown in Fig.1. For an off-chip Cuckoo hash table with k hash functions, there are $k+1$ discriminated vectors (DVs) on the on-chip memory which all

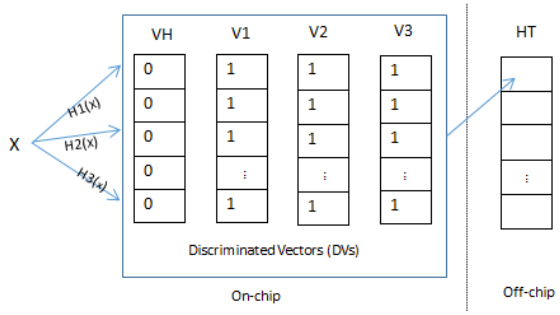

Fig.1.The data structure of DEHT

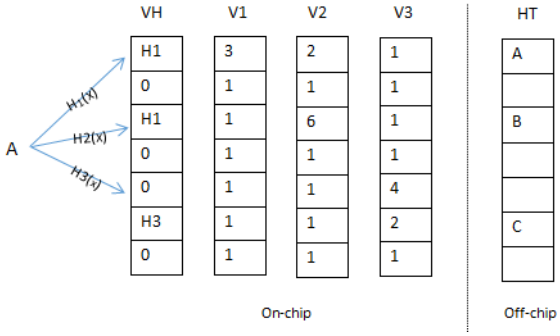

Fig.2.The lookup operations of DEHT

have the same length as the off-chip hash table. The first vector is called VH and the others are called V1, V2,…,Vk. The initial value of the VH is set to 0 while for all other DV vectors the initial values are set to 1. Basically the VH vector stores value indicates that who determines the location in the off-chip hash table and the other Vi vectors store values that can be interpreted as weights which determine if such location is valid. The DVs will be changed accordingly when insert and delete operations are performed. When looking up an element, some simple logic operations on slots of Vi will be executed and exactly one possible location in the off-chip hash table will be returned if the element has been inserted before, thus only one access to the off-chip memory is needed in lookup operation. For queries to nonexistent elements, most of them will not cause off-chip memory access because the DVs can also perform membership test like a Bloom Filter and screen them out. The algorithms of lookup, insert and delete operations will be described in detail below, where 3 hash functions are used as an example.

A. Lookup

When looking up an item X, we first calculate its possible location of 3 candidate slots by hash functions *H1*, *H2* and *H3* and obtain locations *L1*, *L2* and *L3*. We compare the values at V1[*L1*], V2[*L2*], V3[*L3*] and choose the one with the minimum value, let's say it is Vi[*Li*]. Then we will check whether VH[*Li*] equals Hi or not. If so, we confirm that HT[*Li*] should be checked for *X*, or else *X* is not in the off-chip hash table at all.

Fig.2 gives an example of looking-up item *A* from a DEHT that has already 3 elements inserted including *A*, *B* and *C*. We first input *A* to the 3 hash functions *H1*, *H2*, *H3* and get 0, 2, 4 which are the indexes for the candidate slots. Then we fetch the corresponding values from V1, V2 and V3, i.e., V1[0], V2[2] and V3[4]. Among these 3 values V1[0] is the minimum, so we get index 0 and then we check the value at VH[0]. Because

VH[0] equals *H1* which is the label of hash function that corresponds to V1, so we should check HT[0] for the item *X*.

B. Insertion

We first get the three indexes *L1*, *L2*, *L3* and then check whether any of VH [*L1*], VH[*L2*], VH[*L3*] equals 0. VH[*Li*] equals 0 indicates that HT[*Li*] is empty. Given that VH[*Li*] is 0, we can insert the new element into HT[*Li*]. After that, we set VH[*Li*] with *Hi*, which means that HT[*Li*] stores an element that is mapped to this slot by hash function *Hi*. Assuming that the other two indexes are *Lm* and *Ln* ($m \neq n$, $1 \leq m, n \leq 3$), we should add Vi[*Li*] to Vm[*Lm*] and Vn[*Ln*] respectively in order to make sure that Vi[*Li*] is smaller than Vm[*Lm*] and Vn[*Ln*]. At last, we should check whether VH[*Lm*] equals *Hm* and whether VH[*Ln*] equals *Hn*. If VH[*Lm*] equals *Hm*, it means that an element was hashed to HT[*Lm*] by hash function *Hm* at an earlier time, so we should fetch HT[*Lm*] and use hash function *H1*, *H2*, *H3* to calculate and get indexes *Lm*, *Lx*, *Ly* ($x \neq y$, $1 \leq x, y \leq 3$). Then we should add value Vi[*Li*] also to Vx[*Lx*] and Vy[*Ly*]. This operation should be executed recursively, or else adding Vi[*Li*] to Vm[*Lm*] may increase Vm[*Lm*] higher than Vx[*Lx*] and/or Vy[*Ly*] and cause the looking-up for the element of HT[*Lm*] fail. If all the 3 candidate slots of the HT are occupied, we move away any one of the occupying item to another location according to Cuckoo hashing. When moving away an element, we may divide this action into two stages including deletion and insertion.

Fig.3 shows the process of inserting *A*, *B* and *C* into the DEHT that is initially empty. We first calculate 3 hash functions with input *A* and get the corresponding indexes 0, 2, 4. VH[0], VH[2] and VH[4] are all 0, which indicates that in off-chip hash tale HT[0], HT[2] and HT[4] are all empty. So we randomly select HT[0] to insert element *A*. Index 0 is obtained from the calculation of hash function *H1*. So we fetch the values of slot with index 0 from the vector that corresponds to hash function *H1*, i.e. vector V1. And we will add this value V1[0] to V2[2] and V3[4]. At last we set VH[0] with *H1*, which means HT[0] stores an element and this element was mapped to this location by hash function *H1*. Then we insert the element *B*. We input *B* to hash functions *H1*, *H2*, *H3* and get the subscripts 2, 0, 5. Because VH[2] equals 0, we insert *B* to HT[2] and add V1[2] to V2[0] and V3[5]. And we set VH[2] with *H1*. Inserting *C* is a bit more complicated. We get the indexes 0, 2 and 5 at this time. Only VH[5] is empty and we insert *C* to HT[5]. And then we add V3[5] to V1[0] and V2[2]. When adding V3[5] to V1[0], we find that VH[0] is *H1*, which means at an early time an element was stored in HT[0] mapping by hash function *H1* and the value of V1[0] was added to two other slots in V2 and V3 respectively. However, adding a value to V1[0] may lead V1[0] to be no-smaller than that of the other two slots mapped by function *H2* and *H3* in V2 and V3 respectively. So we make an auxiliary lookup of the element stored at HT[0], i.e., A, and get the mapping indexes 0, 2, 4. We then add V3[5] to V2[2] and V3[4], which ensures that V1[0] is still smaller than V2[2] and V3[4].

C. Deletion

Deletion of the DEHT is actually the inverse operation of

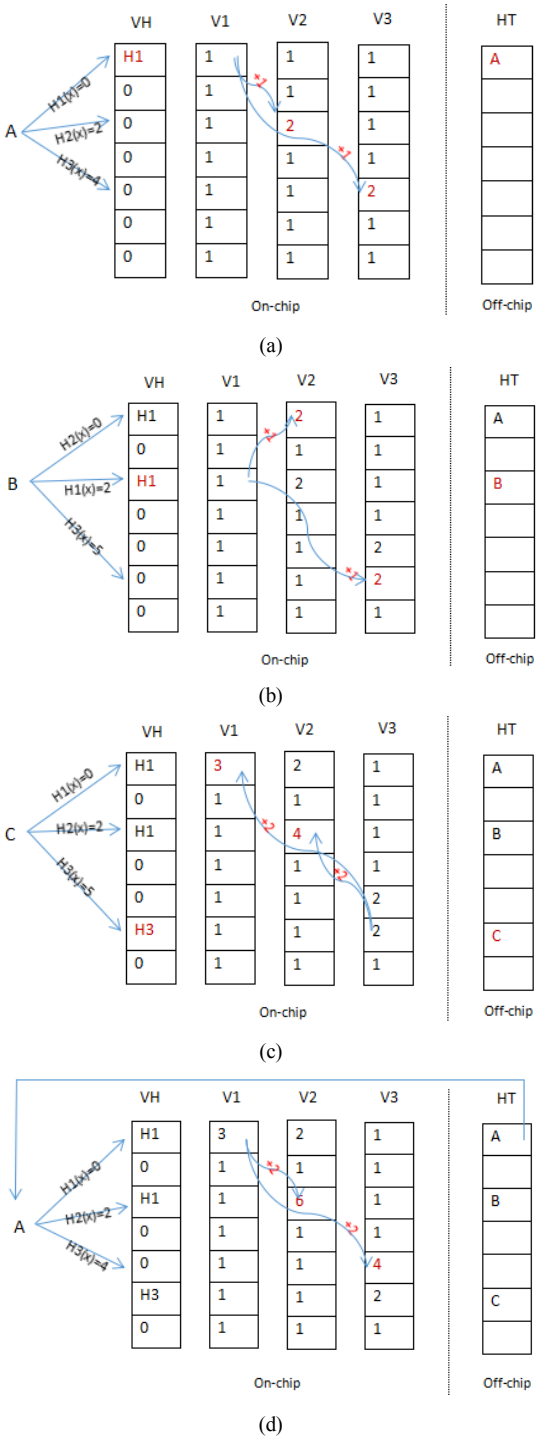

Fig.3.The insertion operations of DEHT

insertion. When deleting an item *X*, we first perform the lookup operation to get the actual location *Li* ($1 \leqslant i \leqslant 3$) from *L1*, *L2*, *L3* of HT that item *X* may lie in. If HT[*Li*] does not contain *X*, the operation should terminate since we get a false positive here. Otherwise we set VH[*Li*] with 0 to indicate that HT[*Li*] now becomes empty. Assuming that the other two indexes calculated are *Lm* and *Ln* ($m \neq n$, $1 \leqslant m, n \leqslant 3$). Then we subtract the value of Vi[*Li*] from Vm[*Lm*] and Vn[*Ln*] respectively. A similar recursive operation as the one used in

insertion should also be used here: if VH[*Lm*] equals *Hm*, HT[*Lm*] should be fetched to calculate its corresponding indexes *Lm*, *Lx*, *Ly* ($x \neq y$, $1 \leqslant x, y \leqslant 3$), and then Vi[*Li*] should be subtracted from Vx[*Lx*] and Vy[*Ly*] because the values of Vx[*Lx*] and Vy[*Ly*] were increased before by Vi[*Li*] from the location of Vi[*Li*].

D. Complexity Analysis

The lookup operation needs to compare 3 values and output the minimum one on-chip. And then a value from the off-chip hash table will be fetch accordingly. So the total complexity of lookup is $O(1)$ and the off-chip memory access is 1 at most.

As for the delete operation, the complexity of off-chip access is equal to that of an ordinary Cuckoo hashing, i.e., $O(1)$. For the on-chip part, we need to subtract a value from two locations of the on-chip DVs and each location has probability of 0.25 (situation of 3 hash functions) to evoke a recursion. So the time frequency of the delete operations on-chip can be obtained by

$$t = 2 + 2 * \frac{1}{4} t,$$

where we get t=4. Under the general situation of k hash functions it is

$$t = 2 + \frac{2}{k+1} t,$$

where we get

$$t = 2 + \frac{4}{k-1}.$$

Because k is a fixed number ranging from 2 to 4 in most situations, so the on-chip complexity is also $O(1)$.

When inserting an element, if any of the candidate slots of the hash table is empty, the on-chip time complexity $O(1)$ similar to that of delete operation. However, if all candidates slots are occupied, one of them need to be moved to another slot to make room for the current inserted one and the relocation may be recursive. Relocation may be divided into two stages including deletion and insertion. Because the on-chip time complexity of the insertion and deletion are both $O(1)$ for every relocation, so the time complexities of on-chip and the whole insert operation are equal to the ordinary Cuckoo hashing.

IV. PERFORMANCE ANALYSIS

A. Memory Efficiency Analysis

What we are concerned about is how many elements can be store in this hash table. We define the load factor f as n/m where n is the number of stored elements in a hash table of size m. And we define the insertion failure rate *ifr* as k/n where k is the number of failures to insert an element into any slot of the hash table allowing moving away elements.

We perform experiments to see the insertion failure rate of our DEHT respectively with 2 and 3 hash functions. And we compare the results with the ordinary Cuckoo hash. The results

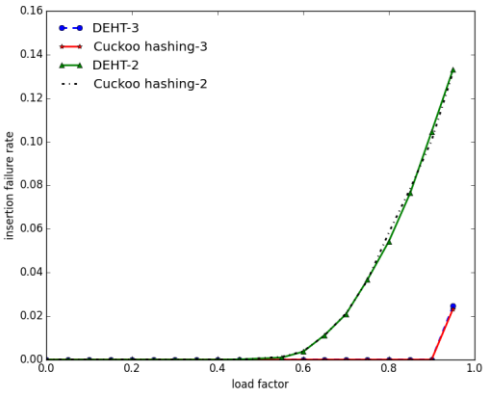

Fig.4. Insertion failure rate of different load factor by experiment

of Fig.4 show that our DEHT has nearly the same performance in usable load ratio of hash table with Cuckoo hashing. When the number of the hash function is 2, the off-chip hash table of size m of DEHT can store about $0.5m$ elements. And the number can be up to $0.9m$ when there are three hash functions. If more than $0.9m$ elements are going to be inserted, some of them will not find any candidate empty slot at any situations.

Now let's look into the memory cost of the on-chip Discriminated Vectors. In the situation of 3 hash functions, there are four vectors VH, V1, V2 and V3 on chip. In every slot of VH there are 4 possible values, so 2 bits are enough for the slots of VH. We conducted an experiment of inserting 30 millions elements into the hash table to see the frequency distribution of values in V1, V2 and V3, and the result can be seen in Fig.5. We find that the cumulative frequency of the values 1-4 and 1-8 are 0.93 and 0.99 respectively and the vast majority of them are very small integers. Methods in order to save memory cost can be applied in such situation [9-12]. They reduce the space requirement at the cost of additional computation and the shuffling of memory, while still keeping worst-case time bounds unchanged on various primitive operations. Method in [12] exploits the idea of hierarchical structure to compress a great deal of wasted space corresponding to zero counters. An original vector is divided into several hierarchical sub-vectors. The sub-vector of the first level has a same length as the original vector, and higher level sub-vectors can be dozens of time shorter than the level below it. In this structure all the slots now only cost very little memory, so the total memory cost is greatly reduced. For example when each slot costs 2 bits and the length of sub-vectors is 10 times smaller up each level, let n denote the total length of the vector, the memory cost of each of V1, V2 and V3 will be

$$Cost(n) = 2n + 2n * \frac{1}{10} + 2n * \frac{1}{100} + \ldots = \frac{20}{9} n.$$

So the total memory cost of the on-chip Discriminated Vectors is the sum of V1, V2, V3 and VH which is

$$Cost(n) = 3 * \frac{20}{9} n + 2n = \frac{26}{3} n.$$

So if the off-chip hash table has 1 million slots, we need just only 1 MB memory on chip for all the DVs.

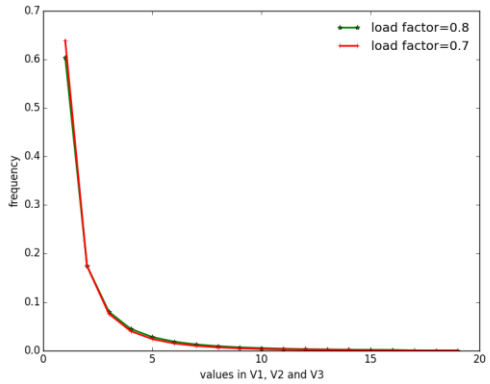

Fig.5. Frequency distribution of numbers in on-chip structures by experiment

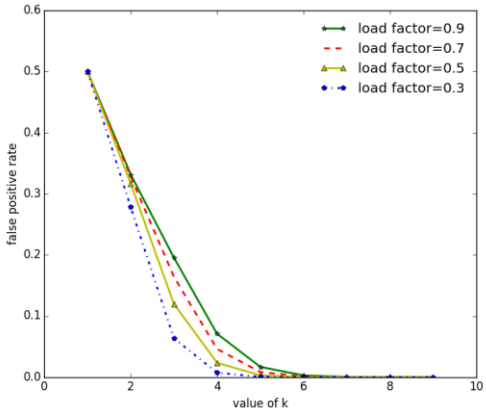

Fig.6. False positive rate of different k by theoretical analysis

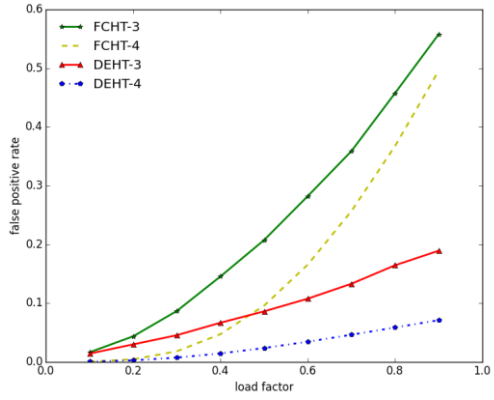

Fig.7. False positive rate of different k by experiment

B. False Positive Rate Analysis

Remember that when performing lookup operations the on-chip Discriminated Vectors can distinguish most nonexistent items from the existent ones and avoid unnecessary off-chip memory access. However, there is still a small probability of mistaking a nonexistent item for an existent one which is called false positive rate, i.e. *fpr*. Given that n elements have been inserted into the hash table whose length is m with k hash functions. Remember that every time an element is inserted, the total number of slots whose value are increased in V1, V2,…,Vk is

$$tot = 2 + \frac{4}{k-1}.$$

So after inserting n elements into the hash table with length of m, the probability that a slot can still maintain its initial value of 1 in V1, V2, …, Vk is

$$p = \left(1 - \frac{tot}{km}\right)^n = \left(1 - \frac{2k+2}{(k-1)km}\right)^n \approx e^{-\frac{(2k+2)n}{(k-1)km}}.$$

When looking up a nonexistent item, we get the indexes $L1$, $L2$, … , Lk by hash functions and let's say the minimum value is Vm[Lm] from V1[$L1$], V2[$L2$], …, Vk[Lk]. If k-1 or k values of V1[$L1$], V2[$L2$], …, Vk[Lk] are more than 1 and VH[Lm] equals Hm, we will visit the off-chip HT[Lm]. So the *fpr* can be

$$fpr = \frac{1}{k+1}\left[(1-p)^k + kp(1-p)^{k-1}\right].$$

The curve of the *fpr* with different k can be seen in Fig.6. We can find that no matter with which load factor (n/m) the *fpr* will always decrease quickly as k increases. And in the case of 3 hash functions or k=3, the *fpr* is always smaller than 0.18 which are trivial compared to other methods. We will compare our results with other methods in section V.

V. Experimental Analysis

A. Experiment Settings

We evaluate the performance of DEHT under different configurations and compare it with FCHT [4] because the targets of FCHT are very similar to our DEHT. DEHT and FCHT both aim at achieving deterministic lookups with multiple hash functions. A class of universal hash functions described in [13] is suitable for hardware implementation [14]. For any item X with b-bits representation as

$$X = < x1, x2, x3, \ldots, xb >$$

the i^{th} hash function over X, $h_i(x)$ is calculate as:

$$h_i(x) = (d_{i1} \times x_1) \oplus (d_{i2} \times x_2) \oplus \ldots \oplus (d_{ib} \times x_b)$$

where '×' is simple multiplication operator and '⊕' is bitwise XOR operator.

B. Experiment Results

The false positive rate of DEHT and FCHT with 3 and 4 hash functions can be seen in Fig.7. The overall *fpr* increases with the load factor becoming larger. However, the growth rate of DEHT is much smaller than that of FCHT. With 3 hash function, the *fpr* of FCHT is up to 0.3 and 0.5 when the load factor is 0.6 and 0.9 respectively. Remember that the FCHT needs to access off-chip memory once for every one positive result outputs by the on-chip Bloom Filters. So this high *fpr* may result in more than one on-chip Bloom Filters outputting positive results and more than one memory access. In contrast, the *fpr* of our DEHT with 3 hash functions is 0.1 and 0.18 when the load factor is 0.6 and 0.9 respectively. However, because the characteristics of the Discriminated Vectors in DEHT are different from the Bloom Filters in FCHT, the *fpr* will not influence the off-chip memory access which will keep at most one access all the time. The rise of the *fpr* may increase

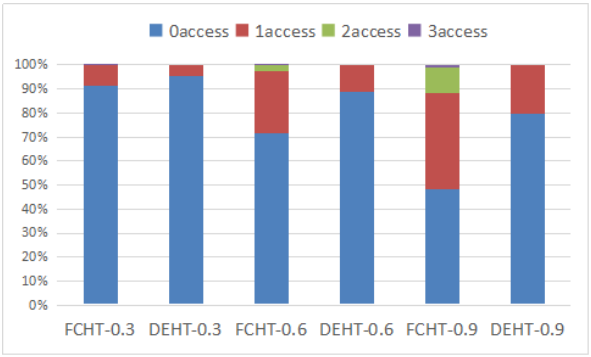

Fig.8.Off-chip memory accesses of DEHT and FCHT under various load factors with random inputs.

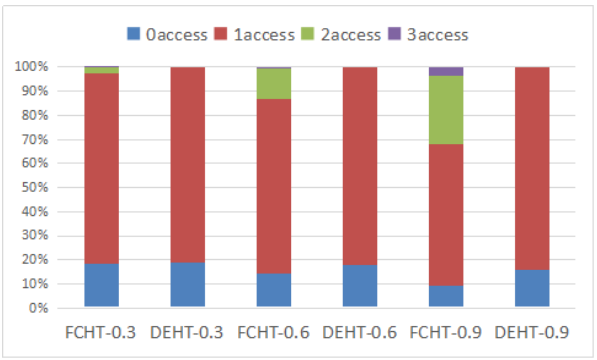

Fig.9. Off-chip memory accesses of DEHT and FCHT under various load factors with inputs of 80% true elements.

the possibility of DEHT to visit the off-chip memory when looking up a nonexistent item. However, the *fpr* of DEHT can still keep low and will not hurt the performance even at high loads.

The comparisons of off-chip memory access are in Fig.8 and Fig.9. We build a hash table of size 30 million and test the lookup operations with load factor of 0.3, 0.6 and 0.9 respectively. Fig.8 shows the results when performing lookups on a random querying dataset and most of the items of the dataset are not in the hash table. We can see from the result that our DEHT is able to distinguish most of the nonexistent items from existent ones and no off-chip memory access is needed for these lookups. And with the load factor rising, the percentage of 0 off-memory access decreases because of the *fpr* ascending. However, this phenomenon is much more obvious with FCHT than DEHT because the rising trend of *fpr* of FCHT is much steeper than DEHT. And when the load factor is 0.9, nearly 10 percent of lookups of FCHT need 2 off-chip memory access and 2 percent of lookups need even 3 off-chip memory access while all lookups of DEHT need at most 1 access. Next we use a querying dataset that has 80 percent of elements existing in the hash table, which is more common in network applications. The result of this querying dataset is in Fig.9. We can see similar result in Fig.9 with Fig.8 except that the percentage of 0 off-memory access becomes much less because the number of nonexistent item of querying is much smaller. There are 12 percent and 28 percent of lookups of FCHT taking 2 off-chip memory accesses with 0.6 and 0.9 load factor respectively. And nearly 4 percent of lookups of FCHT need 3 off-chip memory access with 0.9 load factor. In contrast, the performance of DEHT keeps stable and always need 1 access at most.

VI. Conclusion

In this paper a deterministic and efficient hash table based on Cuckoo hash is proposed. We designed an on-chip data structure called Discriminated Vectors to reduce the access of off-chip hash table to a single one at most when performing lookup operations while keeping the usable load ratio of hash table as high as Cuckoo hashing. The Discriminated Vectors also play a similar role as Bloom Filter in membership screening, which avoids most unnecessary off-chip memory access for looking-up nonexistent items in the hash table. Operations of lookup, insertion and deletion of this data structure are elaborated. We make theoretical analysis and evaluate this structure in experiments. The results show that the memory efficiency is favorable and the false positive rate is low. We ensure deterministic off-chip memory access per lookup even when the hash table is at high loads, which is unattainable for other methods.

References

[1] Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended Bloom filter: an aid to network processing. Acm Sigcomm, 2005, 35(4):181--192.

[2] Kirsch A, Mitzenmacher M. Simple Summaries for Hashing With Choices. IEEE/ACM Transactions on Networking, 2008, 16(1):218-231.

[3] D. Li, P. Chen, Optimized Hash Lookup for Bloom Filter Based Packet Routing, 16th International Conference on Network-Based Information Systems (NBiS) (2013) 31-37.

[4] Huang K, Xie G, Li R, et al. Fast and deterministic hash table lookup using discriminative bloom filters. Journal of Network & Computer Applications, 2013, 36(2):657-666.

[5] Pagh R, Rodler F F. Cuckoo hashing. Journal of Algorithms, 2004, 51(2):122-144.

[6] Kirsch A, Mitzenmacher M. The power of one move: hashing schemes for hardware. In: IEEE INFOCOM; 2008b. p. 106–10.

[7] Kumar S, Turner J, Crowley P. Peacock Hashing: Deterministic and Updatable Hashing for High Performance Networking. Proceedings - IEEE INFOCOM, 2008:101-105.

[8] Frieze A, Melsted P, Mitzenmacher M. An Analysis of Random-Walk Cuckoo Hashing, Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques. Springer Berlin Heidelberg, 2009:490-503.

[9] Bonomi F, Mitzenmacher M, Panigrah R, et al. Beyond Bloom filters: From approximate membership checks to approximate state machines. SIGCOMM '06, 2006, 36(4):315-326.

[10] Bonomi F, Mitzenmacher M, Panigrahy R, et al. An Improved Construction for Counting Bloom Filters// Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings. 2006:684--695.

[11] HUA, N., ZHAO, H., LIN, B., AND XU, J. Rank-indexed hashing: A compact construction of bloom filters and variants. In IEEE International Conference on Network Protocols (ICNP 2008) (Oct. 2008), pp. 73–82.

[12] Ficara D, Giordano S, Procissi G, et al. MultiLayer Compressed Counting Bloom Filters// Infocom Conference on Computer Communications IEEE. IEEE, 2008:311-315.

[13] Carter J L, Wegman M N. Universal classes of hash functions. Journal of Computer & System Sciences, 1979, 18(2):143-154.

[14] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In Proc. of Int. Conf. on Computing and Information, pages 1621–1636, 1994.