IEICE TRANS. INF. & SYST., VOL.E99–D, NO.3 MARCH 2016

LETTER

Hash Table with Expanded-Key for High-Speed Networking

Seon-Ho SHIN[†], **Jooyoung LEE**[††], **Jong-Hyun KIM**[††], **Ikkyun KIM**[††], ***Nonmembers***,
and* MyungKeun YOON**[†a)], ***Member

SUMMARY We design a new hash table for high-speed networking that reduces main memory accesses even when the ratio of inserted items to the table size is high, at which point previous schemes no longer work. This improvement comes from a new design of a summary, called *expanded keys*, exploiting recent multiple hash functions and Bloom filter theories.

key words: *hash table, Bloom filter, multiple hash functions*

1. Introduction

Networking applications process packets at an extremely fast line speed (40 Gbps for OC-768 and 16.4 Tbps in experimental systems) [1]. Since hash tables support INSERT, SEARCH, and DELETE operations with $O(1)$ time complexity, they are widely used in networking applications, including per-flow state measurement, traffic estimation, routing table lookup and packet classification [2].

In general, hash tables are maintained in main memory such as DRAM (also called off-chip memory). While hash functions can be computed rapidly, accessing DRAM is still relatively slow [1]–[3]. An additional DRAM access is required whenever a hash collision happens, which deteriorates the performance. Moreover, a hash table of small size causes a significant variation in the number of DRAM accesses due to hash collisions. The cost varies by a factor of four or more [2], which may not be acceptable for delay-sensitive applications.

Previous work tackles this problem by introducing an extra summary space in fast but small on-chip memory such as SRAM. This summary is looked up first and gives a hint on the optimal location in DRAM. For example, a fast hash table (FHT) [2], the state-of-the-art scheme, keeps a summary that is implemented as a counting Bloom filter [4]. The past research reduces the number of main memory accesses, but it requires the number of hash table slots to be 6∼10 times larger than the number of items stored [2], [5]. We observe that this space waste is too much, which motivates us to design a new summary structure for a hash table.

Our proposed scheme uses a counting Bloom filter as a summary in SRAM, which has been adopted in previous work [2], [5]. The new contribution of our proposed scheme is that hash collisions are resolved with multiple hash functions and the design of summary space is not restricted by the hash table size. Additionally, our proposed scheme does not require any extra reorganization step by inserting a new item optimally distributed over the hash table.

Although summary-based hash tables, including ours, are invented for high-speed networking, we stress that they can also be applied in any computing areas.

2. Related Work

Song *et al*. propose a fast hash table (FHT) that reduces a number of DRAM accesses [2]. A summary is maintained in fast on-chip memory as a counting Bloom filter. The size of counting Bloom filter should equal that of hash table. When a new item is inserted, FHT first looks up each of the item's hash index in the counting Bloom filter. The item is inserted into the hash slot matching to the smallest counter value of the counting Bloom filter. After the item is inserted, the counter increases by one. For SEARCH, the hash slot corresponding to the smallest counter is selected to look up. Of course, the smallest counter of an item may later becomes larger than the others. Then, any item not located at the smallest counter should be moved into the smallest counter position. This rearrangement process requires extra DRAM accesses. Actually, INSERT and DELETE operations of FHT require multiple DRAM accesses. To mitigate this weakness, FHT requires a large number of extra slots. According to the paper's experiments [2], the number of slots is ten times larger than the number of items.

Kirsch and Mitzenmacher provided another summary-based hash table [5]. They use a multilevel hash table with independent sub-tables. Each sub-table is associated with a unique hash function. The construction of a table has a strong skew property in that the first sub-table contains most of the items, the second sub-table contains most of the rest, and so on. The summary is located in SRAM, which consists of Multiple counting Bloom Filters (MBF), one per sub-table. This approach sequentially probes multilevel hash tables when a new item is inserted unlike FHT. Therefore, the chance to find an empty slot is increased, but the process requires extra delay time. But, this still requires a lot of extra slots; the length of the hash table is configured six times longer than the number of stored items [5]. Another serious problem with MBF is that an item may not

---

Manuscript received September 15, 2015.
Manuscript revised November 2, 2015.
Manuscript publicized December 11, 2015.

[†] The authors are with Kookmin University, Seoul, Korea.
[††] The authors are with Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea.
a) E-mail: mkyoon@kookmin.ac.kr

DOI: 10.1587/transinf.2015EDL8201

be inserted into a hash table if all of the available slots are already occupied, which is called a *crisis*. When a new INSERT causes a crisis, the hash table fails to store that item. The interested reader is referred to [5] for more information.

A summary-based hash table includes a counting Bloom filter that supports a DELETE operation [4]. When a counter reaches its maximum value, for example 15 for a 4-bit counter, additional INSERT would trigger an overflow and continuous DELETE operations may cause the filter to lose information. However, the authors of the counting Bloom filter proved that 4-bit counters are generally large enough to keep the overflow probability minuscule [4]. Including this paper, summary-based hash tables [2], [5] utilizes a counting Bloom filter of 4-bit counters. The issues around the optimal configuration of the counting Bloom filter is beyond the scope of these papers.

A Bloomier filter returns the value for a queried item [6]. When the set of items and the corresponding values are static, there are linear space solutions. However, this approach is not fit for networking applications or any others where INSERT or DELETE is required during the operation of the filter.

Broder and Mitzenmacher provided d-left scheme [7]. They use multiple hash functions and multiple-choice to evenly distribute items into a hash table. Recently, Kanizo *et al.* propose how to maximize the throughput of d-left hash tables by using a bipartite graph model [8]. These schemes require d lookups for DRAM in parallel to reduce the delay time for memory accesses, but it imposes costs of other resources, such as pin count in hardware [5].

3. Fast and Compact Hash Table (FCH)

3.1 Definitions and Architecture

Let $E = \{e_1, e_2, \ldots, e_n\}$ be the set of items to be stored in a hash table. The key value of e_i is denoted as y_i, and therefore $y_i = e_i.key$. Let m be the number of slots in the hash table. The i^{th} slot in the table is denoted as $T[i]$, $1 \leq i \leq m$. We use multiple hash functions. Let k be the number of hash functions, and this pool of hash functions is denoted as $H = \{h_1, h_2, \ldots, h_k\}$. We term h_i the i^{th} hash function and i is the id, or label of the hash function. We assume that the hash functions are independent each other. Inserting e_i into the hash table, we compute k hash functions, from $h_1(y_i)$ to $h_k(y_i)$, and one slot is chosen for INSERT.

Suppose that the j^{th} hash function is chosen for inserting e_i. Then, we insert e_i into the slot of $T[h_j(y_i)]$. We term such an h_j an *assigned hash function* (A-function) for e_i, and denote its label j as $a(y_i)$. We define an expanded key for y_i as '$y_i||a(y_i)$' where $h_{a(y_i)}$ is the A-function and $||$ is a string concatenation operation. Note that each key and A-function should be extended to a fixed length of digits. For example, the length of the key and A-function is 32 and 4, respectively, the length of the expanded key is fixed at 36. We use $x(y_i)$ to denote y_i's expanded key, and therefore $x(y_i) = y_i||a(y_i)$.

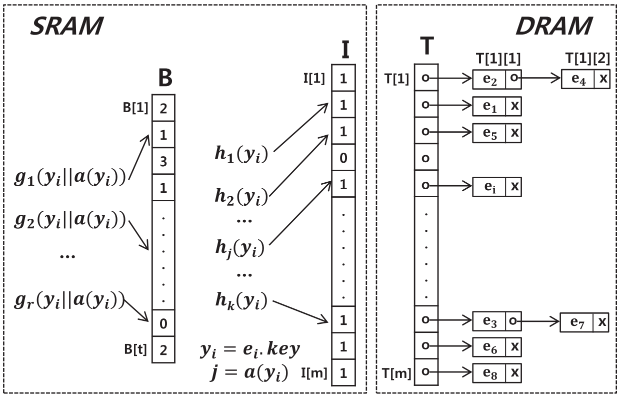

Fig. 1 Architecture of fast and compact hash table (FCH).

For collision resolution, we use a chaining scheme by placing all the items that hash to the same slot into the same linked list. Therefore, $T[i]$ contains a pointer to the head of the list of all stored items that hash to i. For simplicity, we use $T[i][j]$ to denote the j^{th} item in $T[i]$, $1 \leq j \leq n$, although T is not a real two-dimensional array. We define the length of $T[i]$ as the number of items in that linked list, and denote it as $T[i].length$. If there is no item in $T[i]$, we say $T[i]$ is empty. We define the load factor, α, for T as n/m where n is the number of items stored in the hash table.

The architecture of FCH is shown in Fig. 1. It consists of a hash table and a summary. The hash table, $T[m]$, is built in DRAM, and the summary is located in SRAM. Each slot of the hash table has a pointer to a linked list. The summary is implemented with two arrays, a counting Bloom filter of size t, and a bit array of size m, denoted as $B[t]$ and $I[m]$, each cell of which is initialized to zero. I is an indicator bit array; if $I[i]$ is zero, it means that $T[i]$ is empty. B records expanded keys. These arrays are compact and can be placed in SRAM.

The signatures of expanded keys are programmed into B, the counting Bloom filter, which is a hash-based data structure that stores a set of items compactly [4]. It is based on a traditional Bloom filter, but can support the DELETE operation the Bloom filter cannot provide. Here, an item is an expanded key. The i^{th} counter is denoted as $B[i]$, and all counters are initialized to zero. Each counter requires only few bits, for example four bits per counter, the soundness of which is proved in [4]. We use c to denote the number of bits per counter. The counting Bloom filter computes r hash functions on each item, and this pool of hash functions is denoted as $G = \{g_1, g_2, \ldots, g_r\}$. Each hash function returns an address of array B. Note that the hash function pools of H and G are different, and they are used for hash table T and counting Bloom filter B, respectively.

3.2 Operations

When inserting e_i, multiple hash functions are computed from $h_1(y_i)$ to $h_k(y_i)$ to find an empty slot. If multiple empty slots are found, we choose the hash function of the smallest

```
$INSERT(e_i)$
1.  for $j := 1$ to $k$ do
2.      $ind := h_j(y_i)$
3.      if $I[ind] = 0$ then
4.          $T[ind][1] := e_i$
5.          $I[ind] := 1$
6.          goto line 9
7.  if $T[ind][1] \neq e_i$ then
8.      insert $e_i$ at the tail of $T[ind]$
9.  for $s := 1$ to $r$ do
10.     $B[g_s(y_i||j)] := min(B[g_s(y_i||j)] + 1, 2^c - 1)$
```

Fig. 2 INSERT for the fast and compact hash table (FCH).

```
$SEARCH(e_i)$
1.  for $j := 1$ to $k$ do
2.      for $s := 1$ to $r$ do
3.          $ind = g_s(y_i||j)$
4.          if $B[ind] = 0$ then
5.              go to line 1
6.      $ind = h_j(y_i)$
7.      if $I[ind] = 0$ do
8.          go to line 1
9.      if $T[ind][1] = e_i$
10.         return $ind$
11.     if $j = k$ then
12.         traverse $T[ind]$.
13.         if $e_i$ is found, return the address
14. return UNSUCCESSFUL_SEARCH
```

Fig. 3 SEARCH for the fast and compact hash table (FCH).

label value for tie-breaking. Figure 2 shows the INSERT algorithm for FCH. Note that e_i is inserted into the linked list of the last hash index, which can work the same as any fair load balancing for appending a new item.

When e_i is searched for, we first look up the summary where the expanded key may have been inserted. If this value is obtained, we know the A-function for y_i. Then, applying the A-function to y_i leads us to the exact slot in the hash table. Therefore, we can search for the item with a small number of DRAM accesses. Figure 3 shows the SEARCH algorithm of FCH.

When e_i is deleted, we first perform SEARCH. Then, e_i is removed from the linked list, and the counters are decremented in the summary as follows:

$$B[g_s(x(y_i))] := max(B[g_s(x(y_i)\|j)] - 1, 0), 1 \leq s \leq r.$$

Comparing our proposed scheme with previous work, we summarize the reason why the number of DRAM accesses could be reduced with FCH. Note that extra DRAM accesses are required whenever a collision happens in hash tables. In the previous work, collisions are resolved by increasing the size of hash table, and therefore this approach does not work with a large load factor. The design of summary space is also restricted by that of hash table; for example, the counting Bloom filter size should equal the hash table size [2]. We stress that our proposed scheme resolves the hash collision problem by mapping every inserted item with one of multiple hash functions. This mapping is based on the policy that every item is optimally distributed over the hash table. Additionally, the design of summary space is not restricted by the hash table size in FCH.

4. Experiments

We evaluate FCH through experiments using real Internet traffic traces, collected from a campus network. In our experiments, we compare FCH with two existing summary-based hash tables, FHT [2] and MBF [5]. Our evaluation is based on the number of DRAM accesses as in [2], [5]. For a fair comparison, we allocate the same number of slots and items to different hash tables in the experiments. The number of items is fixed, and α is changed from 0.1 to 1.0 while the memory size is accordingly changed.

Instead of implementing FCH, FHT, and MBF in hardware, we compare the number of DRAM accesses. This experimental comparison has been adopted in previous work [2], [5]. Hardware implementation is not always possible and the main cost is caused by DRAM accesses rather than SRAM. During the experiments, we observe that FCH requires less SRAM accesses, compared with FHT and MBF. The reason is that extra SRAM accesses are required when a hash table collision happens [5] or a new inserted item triggers table reorganization [2]. Note that FCH causes less collisions and no reorganization step is required.

The key value for hashing is defined as a combination of source and destination IP addresses from the traffic traces. We set n to 10,000 as in [5] and [2]. For our experiments, we use the first 10,000 distinct keys from the traffic traces. We insert these 10,000 keys into three hash tables, FCH, FHT, and MBF, respectively, and count the number of DRAM accesses. Then, each count is divided by n, which makes the average number of DRAM accesses per INSERT. Next, we search these 10,000 keys from each table, and compute the average number of DRAM accesses per successful SEARCH. The same process is repeated for DELETE. For unsuccessful SEARCH, we randomly generate 10,000 keys that the traffic traces do not include. We try to search each table for these 10,000 keys.

We obtain interesting results from the experiments; FCH saves more DRAM accesses than other hash tables over any α. FHT requires DRAM accesses more than 8 times FCH, for INSERT and DELETE operations. For SEARCH, FHT reduces the cost to the level of FCH only when α becomes as small as 0.1. In contrast, FCH keeps the costs close to an optimal point even with large α of 0.8~1.0. Figure 4 shows the average numbers of DRAM accesses for INSERT, successful/unsuccessful SEARCH, and DELETE, respectively, from left to right. Each plot compares FCH and FHT as α changes. We use $k = 4$ for FHT, and $k = 4$, $r = 4$, and 64 KB is allocated equally to both FHT and FCH for fair comparison. We observe that FCH outperforms FHT at any different parameter configuration and memory size.

The plots reveal that FHT requires at least 8 DRAM accesses for INSERT and DELETE while FCH keeps the cost to the lowest point. For successful and unsuccessful SEARCH, FCH always requires less DRAM accesses than FHT. The difference becomes large as α increases. These

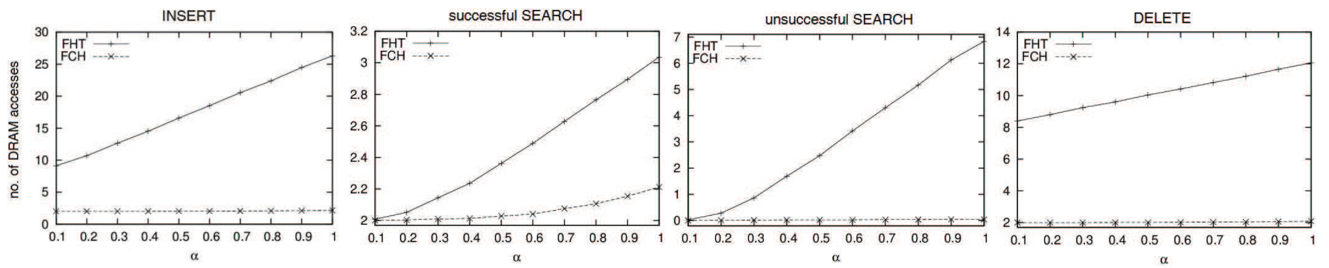

Fig. 4 FCH v.s. FHT. The plots show the average numbers of DRAM accesses for INSERT, successful SEARCH, unsuccessful SEARCH, and DELETE, respectively, from left to right. Each plot compares FCH and FHT as α changes.

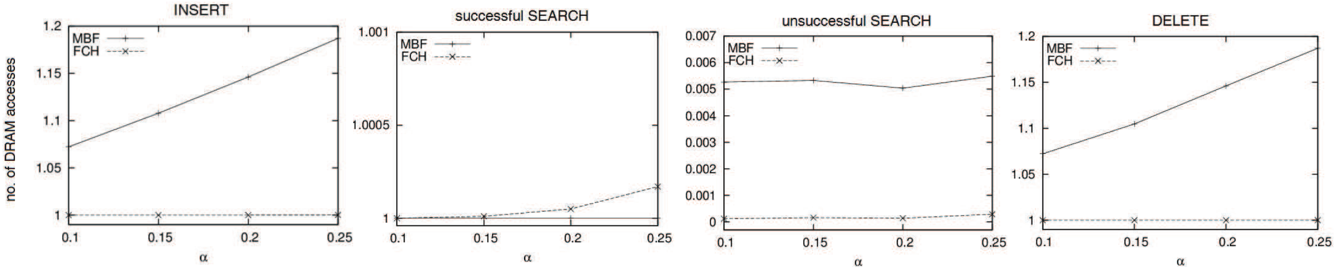

Fig. 5 FCH v.s. MBF. The plots show the average numbers of DRAM accesses for INSERT, successful SEARCH, unsuccessful SEARCH, and DELETE, respectively, from left to right. Each plot compares FCH and MBF as α changes.

plots reveal that FHT reduces SEARCH costs alone only when α is very small.

Figure 5 shows the average numbers of DRAM accesses for INSERT, successful SEARCH, unsuccessful SEARCH, and DELETE, respectively, from left to right. Each plot compares FCH and MBF as α changes. Both schemes are assigned 99 KB for the summary space. FCH performs better than MBF except for unsuccessful SEARCH, but the difference is not significant. Actually, both schemes keep the costs to optimal points. During the experiments, a crisis happens with MBF when $\alpha > 0.2$. We observe that FCH outperforms MBF at any different parameter configuration and memory size.

5. Conclusions

This paper proposes a new hash table that is able to reduce the number of DRAM accesses for all the dictionary operations of INSERT, SEARCH, and DELETE, over a wide range of load factor. The proposed scheme successfully reduces DRAM accesses while keeping the hash table compact, which has not been achieved by previous work.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (B0101-15-0300, The Development of Cyber Blackbox and Integrated Security Analysis Technology for Proactive and Reactive Cyber Incident Response).

References

[1] T. Li, S. Chen, and Y. Ling, “Fast and compact per-flow traffic measurement through randomized counter sharing,” Proc. IEEE INFOCOM ’11, pp.1799–1807, April 2011.

[2] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, “Fast hash table lookup using extended Bloom filter: An aid to network processing,” Proc. ACM SIGCOMM, pp.181–192, Aug. 2005.

[3] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, “Space-code Bloom filter for efficient per-flow traffic measurement,” Proc. IEEE INFOCOM 2004, pp.1762–1773, March 2004.

[4] L. Fan, P. Cao, J. Almeida, and A.Z. Broder, “Summary cache: A scalable wide-area Web cache sharing protocol,” IEEE/ACM Trans. Netw., vol.8, no.3, pp.281–293, June 2000.

[5] A. Kirsch and M. Mitzenmacher, “Simple summaries for hashing with choices,” IEEE/ACM Trans. Netw., vol.16, no.1, pp.218–231, Feb. 2008.

[6] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The Bloomier filter: An efficient data structure for static support lookup tables,” Proc. Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp.30–39, 2004.

[7] A. Broder and M. Mitzenmacher, “Using multiple hash functions to improve IP lookups,” Proc. IEEE INFOCOM ’01, pp.1454–1463, 2001.

[8] Y. Kanizo, D. Hay, and I. Keslassy, “Maximizing the throughput of hash tables in network devices with combined SRAM/DRAM memory,” IEEE Trans. Parallel Distrib. Syst., vol.26, no.3, pp.796–809, 2015.