Shifting Hash Table: An Efficient Hash Table with Delicate Summary

Jie Jiang*, Yibo Yan‡, Mengyu Zhang*, Binchao Yin†, Yumeng Jiang*,
Tong Yang*, Xiaoming Li*, Tengjiao Wang*
* Department of Computer Science, Peking University, China
† Beijing University of Posts and Telecommunications, China
‡ School of Electronics and Computer Engineering (SECE), Peking University, China

***Abstract*—Hash tables have been broadly used in many security applications. These applications require fast query speed and high memory efficiency. However, the query speed degrades when hash collisions happen. The design goal of this paper is to achieve high load factor as well as fast query at the same time. In this paper, we propose a novel hashing scheme, namely the *Shifting Hash Table* (SHT), which consists of two parts. The first part is an enhanced version of the Bloom filter checking which subtable(s) may contain the incoming items, and the second part contains a cuckoo hashing based hash table which stores the key-value pairs. The key technique of this paper is that we divide items into two categories: *at-home* and *abroad*. We only insert the second kind of items (abroad) into the Bloom filter, thus the memory usage of the filter is significantly reduced. We conducted extensive experiments and the results show that SHT significantly outperforms the state-of-the-art. Specifically, SHT can query an item using on average less than 1.05 bucket probes and even using 1 bit per entry in the fast memory, and achieve a high load factor which is 95% at the same time.**

I. Introduction

As a basic data structure, the hash table is widely used in many security applications, such as authentication and identification in RFID [1], [2], malware detection in IoT [3], [4], etc. Many of these applications need high query speed and low memory usage of index structures at the same time. Owing to $O(1)$ query complexity, hash tables are widely used in such applications. Hash tables employ key-value structures and set up key-to-bucket maps using hash functions. Due to hash collisions, traditional hash tables cannot achieve a high load factor during construction, which means there are many empty buckets in the hash table. Therefore, we need offer sufficient memory to accommodate all the items, and this causes memory inefficiency.

Many open addressing methods are proposed to improve the load factors of hash tables, and these methods allow an item to be stored in multiple candidate positions. They use a probe sequence to lookup items. A typical scheme named Cuckoo hashing [5] has obtained wide acceptance in recent years. In cuckoo hashing, each item has d candidate buckets. When inserting a new item, the hash table can move items among their candidate buckets to make room for the new item. Through this way, cuckoo hashing can achieve a very high load factor (e.g., 95%) with high probabilities. In cuckoo hashing, querying an item needs probe at most d buckets. When the item is in the hash table, $\frac{d}{2}$ probes are needed on average if we do the probes sequentially. Although this overhead is low, it still causes unnecessary bucket probes in the memory, which will increase the query latency and occupy memory bandwidth.

To address this issue, one strategy is to use a small summary to determine where the item can be located before probing all the candidate buckets. This summary should be small enough to be stored in the fast memory (such as SRAM in ASIC/FPGA [6], CPU caches). The whole hash table is often too large to be accommodated in the fast memory, and it can only be stored in slow memory (such as DRAM [6]). Based on this strategy, a straightforward method is grouping the buckets as several subtables, and representing each subtable using a Bloom filter [7], [8]. The Bloom filter [9] is a compact data structure which is used to answer whether an item belongs to a set. The Bloom filter has false positives, which means it can report true even if the item does not belong to the set. Suppose there are d subtables, and every item has exactly one candidate bucket in each subtable, then we can build d Bloom filters as the summary. When querying an item, we query the d Bloom filters first, and only one subtable is reported when no false positives happen. In this case, the item will be found in the reported subtable. Because access to the fast memory is fast, the query performance of such kind of hash tables is determined by the time consumed in the slow memory part. Thus, we can use the number of bucket probes as the metric of the query performance. In such type of hash tables, because the false positive rates of Bloom filters are proportional to the number of items inserted, a key problem is how to reduce the number of items inserted into the summary. Peacock [7] reduces the number largely, but it uses multiple Bloom filters, which makes queries of the summary more complicate. To this end, we propose a novel hash table, namely the *Shifting Hash Table (SHT)*. SHT also consists of two parts: a summary in the fast memory and a cuckoo hashing based hash table in the slow memory. Our SHT algorithm has two key techniques as follows: First, in the hash table part, we classify the items into two categories: *at-home* and *abroad*, and only the “*abroad*” items are inserted into the summary. Furthermore, we add three rules to the eviction mechanism of original cuckoo hashing,

This work is supported by Primary Research & Development Plan of China (2016YFB1000304), NSFC (61672061).

which reduce the the number of abroad items sharply. This means that we only need a small summary in the fast memory. Second, in the summary part, we propose an enhanced Bloom filter in place of multiple Bloom filters to achieve fast query speed. There is only one Bloom filter in the fast memory and we represent the information of the subtable index(es) of an item as the offsets in the Bloom filter. Specifically, to represent that an item is in the sth subtable, we set the positions with an offset s to 1, which are *shifted* from their original positions. And the shifting information can be retrieved by reading consecutive bits.

The key contributions of this paper are as follows:

- We propose a novel hash table, namely *the Shifting Hash Table (SHT)*, to achieve small fast memory usage, fast query speed and high load factor at the same time.
- We derive the formulas of the number of bucket probes of queries, and validate them using experiments.
- We carry out extensive experiments to compare our SHT with peacock hashing and BCHT, and results show that our SHT works much better. By using 1 bit per entry in the fast memory, SHT can query an item using on average less than 1.05 bucket probes when the load factor is 95%.

II. Background

In this section, we will introduce cuckoo hashing, Bloom filters and peacock hashing, which inspire our work. More related work please refer to [10]–[14].

A. Variants of Cuckoo Hashing

Cuckoo hashing is one efficient technique to solve collisions in the hash tables. The key idea of it is that each item has multiple candidate buckets in the hash table, and the existing items in the table are allowed to be moved recursively among these buckets to make room for new inserted items. The original cuckoo hashing [5] uses two hash functions which map each item to two buckets, and each bucket can accommodate one item. To achieve higher load factor, two variants of cuckoo hashing were proposed: *d-ary cuckoo hashing (d-ary)* [15] and *bucketized cuckoo hash tables (BCHT)* [16], [17]. The d-ary cuckoo hashing extends the number of hashing functions from 2 to d, and using d subtables. Each item has exactly one candidate bucket in each subtable. BCHT allows each bucket in the hash table to accommodate d items. Both of the two methods allow a hash table with $(1+\epsilon)n$ memory cells to accommodate n items when $d = O(log(1/\epsilon)))$.

In many memory architectures, BCHT has better query performance than d-ary, because sequential accesses perform better than random accesses in memory. In many hardwares, the processor has to fetch several bytes from memory at a time, e.g., most CPUs fetch a 64 bytes cache line from DRAM. This means there is less difference between probing one cell and probing four cells in terms of memory access if four cells can fit into a cache line. Furthermore, one can use one instruction to compare four keys by leveraging SIMD instructions [16] to accelerate this process.

Since memory accesses take up more time than key comparisons, we can use the number of bucket probes to measure the query speed. Like prior work [18], queries are classified as positive queries and negative queries. If a query lookups a key which is in the hash table, then it is a positive query, otherwise it is a negative query. For BCHT, positive queries will incur 1.5 bucket probes on average, while negative queries will incur 2 bucket probes. Although BCHT is more efficient than d-ary, it still causes many unnecessary bucket probes, which increase the query latency and occupy the limited memory bandwidth.

B. Bloom filters and Peacock Hashing

Fortunately, some small but fast memories are available in many devices, such as SRAM in FPGA and caches in CPU. The latency of accessing these memories is small, and this latency can be ignored compared with the slow memory, such as DRAM. But their sizes are often too small to accommodate the whole hash table. We can store a small summary of the hash table in the fast memory to filter out some buckets which cannot hold the items. Before querying the buckets in the slow memory, we can query this summary first to get a subset of candidate buckets. Then we only query the buckets in the subset to reduce the unnecessary bucket probes.

Due to the space limitation of the fast memory, deterministic algorithms cannot support constant time update and query in such a small space. We rescue to the probabilistic algorithms to design the summary, e.g., *Bloom filters*. The Bloom filter [9] is a compact data structure which supports fast membership queries with small space. A Bloom filter consists of an array of m bits, which is initialized to 0. The Bloom filter uses k independent hash functions, which map items to $\{0, 1, ..., m-1\}$ bit positions in the array. When inserting an item x, it calculates k positions using the hash functions and set all these positions to 1. To query an item x, get all the k bits which x is hashed to. And if these bits are all 1, then report x is in the set, otherwise report not. There are false positives in Bloom filters, which means an item may be actually not in the set even if the Bloom filter reports true.

The Bloom filter can record whether an item is in a subtable. When a query incurs a false positive, the Bloom filter will report the item is in this subtable, and it will lead to an unnecessary bucket probe. The false positive rate of a Bloom filter is proportional to the number of items inserted into it, so an important problem is how to reduce this number. Peacock hashing [7] is one of the algorithms that use Bloom filters as their summary. Peacock hash uses a hierarchy of hash table segments (a main table and several backup tables) whose sizes form a geometric decreasing sequence. The main table is the largest subtable, and it will be much larger than the sum of the sizes of the backup tables. Peacock builds a summary only for items in the backup tables: each backup table has a corresponding Bloom filter in the fast memory to represent the items in it, so the number of items in the Bloom filter can be significantly reduced.

But Peacock has three shortcomings: 1) Due to the lack of the mechanism of moving existing items, peacock cannot

achieve a high load factor (about 90% even use the bucketized techniques, as shown in section VI). 2) It has to query multiple Bloom filters in the fast memory, and these accesses cannot be merged by some techniques like [8] because of the different sizes of the Bloom filters. 3) When the hash table is full, it is non-trivial to add a subtable directly because of the unequal sizes of its subtables. These shortcomings motivate our design.

III. Related Work

A large amount of literature focused on dealing with collisions in hash tables. In this section, we focus on cuckoo hashing and other work using open address schemes.

Cuckoo hashing is proposed in [5], and be generalized to d-ary [15] and BCHT [17] to achieve high load factors. Except for the two techniques, [19] demonstrates that the insertion failure probability can be dramatically reduced by using a constant-sized stash. [20] proposed partial key cuckoo hashing which does not need the entire key to get hash positions. This technique is useful when the key length is variable and only part of the key can be stored in the index data structure. [21] extends this idea to build a filter which supports approximate membership query. [22] focus on how to support concurrent insertions and queries. [23] designs an on-chip data structure called discriminated vectors to do membership screening. Rectangular Hash Table [24] utilizes a bitmap to determine which key-value pair among buckets will most quickly find a new empty bucket, and then kicks that key value pair out. SmartCuckoo [25] can predetermine the occurrence of the endless loop during item insertion. Recently, Horton Table [18] is proposed to reduce the bucket probes, and their method is similar to us. Horton table is also based on BCHT, and each item has a primary bucket and several backup buckets. If an item is not in its primary bucket, a 3-bit remap entry for this item will be stored in its primary bucket, so the candidate buckets can be known by probing the primary bucket. Horton table does not need a summary in fast memory. The shortcoming of Horton table is that one must probe the primary bucket before probing other buckets, while in SHT these probes can be done in parallel.

Besides cuckoo hashing, some other open addressing schemes are proposed to solve collisions. Hopscotch hashing [26] moves items in a sequential range to achieve high load factors, and this technique has better cache locality during hash table operations. Path hashing [27] organizes the buckets as a binary tree to solve collisions without the help of moving items in the hash table, which is better for non-volatile memory technologies (NVMs). RwHash [28] dynamically moves items within a bucket when updating based on cache mechanism and update sequence, which significantly reduces collisions.

Fast Hash Table (FHT) [6] is the first scheme which uses a small summary in fast memory to reduce the memory accesses in slow memory. Segmented hashing [8] uses multiple subtables and builds one Bloom filter for each subtable. Both of these two algorithms need large fast memory to build the summary. Peacock [7] reduces the summary size largely, and the detail is introduced in section II.

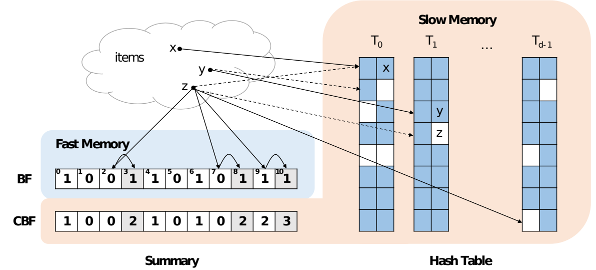

Fig. 1. The architecture of SHT

IV. Shifting Hash Tables

In this section, we will introduce our proposed algorithm, the *Shifting Hash Table (SHT)*. At a high level, SHT is based on cuckoo hashing, and uses an enhanced Bloom filter to reduce bucket probes. The structure of SHT is shown in Fig. 1. SHT consists of two parts: the hash table part and the summary part. We will show the design details and interactions of these two parts in the following subsections.

A. The Design of the Hash Table

The hash table of SHT is based on d-ary cuckoo hashing which contains d subtables $(T_0, T_1, ..., T_{d-1})$. Each subtable T_i has a hash function h_i corresponding to it to locate a candidate bucket for an item. The main difference is that we classify the items in the hash table into two categories: *at-home* items and *abroad* items. Besides the group of hash functions $h_i(.)(0 \leqslant i \leq d)$ corresponding to the subtables, we use another hash function $g : U \mapsto [0, d)$ to decide a *home* subtable for x. Although an item has one candidate bucket in each of the d subtables, it is stored in only one of the buckets at the same time. The item in its *home* subtable is called an *at-home* item, otherwise it is an *abroad* item. The candidate bucket in x's home subtable is called x's *home bucket*, which is calculated by $h_{g(x)}(x)$. In Fig. 1, solid lines point to items' home buckets, and dashed lines point to items' other candidate buckets. In Fig. 1, both of x and y are at-home items, and z is an abroad item because its home subtable is T_{d-1} rather than T_1.

To keep the number of at-home items as large as possible, we add three rules on the eviction mechanism of the original cuckoo hashing: 1) SHT always tries to insert an item to its home bucket first. 2) SHT never allows an abroad item to evict an at-home item. And 3) when there are both an abroad item and an at-home item can be evicted to make room for the inserted item, the abroad item should be evicted first. These rules will limit the efficiency of eviction degrade the load factor. Therefore, we also apply the bucketized technique on the hash table: The hash table in SHT contains d bucket arrays, and each bucket consists of w cells. As shown in our

experiments (Section VI), SHT can easily achieve a high load factor (e.g., 95%) with small d and w, e.g. $d = 8$ and $w = 8$.

B. The Design of the Summary

We leverage an enhanced Bloom filter (BF) to summarize the abroad items. By building one Bloom filter for each subtable, it is easy to find out whether an item is in a subtable or not, but this causes several Bloom filter queries during one query. Thus, we enable a Bloom filter to report which subtable(s) may accommodate the item through modification. Assume that the item x is in the subtable T_s, we set the position offset by s to 1, instead of setting the hashed position to 1. For example, in Fig. 1, there are 3 hash functions for the BF, and the item z is in the subtable T_1, where s is 1. Assuming hashed values of x are 2, 7, and 9, we set the position 2+1, 7+1, and 9+1 to 1. Correspondingly, the query operation should get the consecutive d bits at each hash position and do bitwise *AND* operations to get candidate subtables may contain a target item. Note that the BF only records location information of abroad items.

The summary should support the insertion, query, and deletion operations, while the Bloom filter do not support deletions. Similar to peacock, we use a counting Bloom filter (CBF) to address this problem [29]. The CBF uses counters instead of bits to support deletions. Because the CBF occupies much memory, we put it in the slow memory as a mirror of the BF in the fast memory. During deletion, we first decrease the counters of CBF, and then reset the corresponding bit in the Bloom filter to 0 only when one counter in the CBF is 0. Notice that the CBF will only be accessed during insertion, so the query speed will not be affected.

C. SHT Operations

Combining these two parts, the insertion and query procedure of SHT are shown as follows.

Insertion: When inserting an item x, SHT first calculates $g(x)$ to locate its home subtable $T_{g(x)}$. Then, SHT tries to insert it into its home bucket B_x in the $T_{g(x)}$. If x is already in the B_x, or the B_x has at least one empty cell, the B_x will accommodate x and end insertion. Otherwise, SHT will evict an abroad item x' in the B_x to make room for x. If all the items in the B_x are at-home, SHT will not accommodate x in the B_x. After x' is evicted or x is not inserted into the B_x, SHT will accommodate the x' or x in its other candidate buckets as cuckoo hashing does. While evicting items from buckets to make room for new items, SHT always follows the aforementioned three rules. Correspondingly, SHT updates the BF and the CBF as mentioned above each time it evicts an item or accommodate an item. Note that, during the insertion, if an abroad item is evicted, it will be deleted from the CBF, and if any counter(s) of the CBF comes to 0, the corresponding bit(s) in the BF will be reset to 0.

Query: When querying an item, we should query the BF first and get bit vectors. Then SHT queries the subtable(s) determined through bitwise AND operations with the bit vectors. If the item is not found, SHT probes the item's home bucket. The CBF is not accessed during this process.

D. Extensions

Using multiple subtables can accommodate more items than preallocated memory cells by dynamically adding a subtable. This technique also applies to our SHT algorithm. But the change of the number of subtables may make the home decision hash function ($g(x)$) failed. To address this issue, we use linear hashing [30] as $g(x)$. Only items in a specific subtable (T_s) may be moved after adding a new subtable T_n. The moving procedure can take a long time, and it shouldn't block the query procedure. Thus, we use a flag to represent whether the moving procedure is over or not. If not, for an item x in T_s or T_n, SHT will consider both T_s and T_n are home subtables of x. This will cause additional bucket probes when query x. Once moving is done, the flag will be reset, and SHT will return to normal behavior.

V. ANALYSIS

In this section, we analyze the number of bucket probes of both positive queries and negative queries in SHT. Because of the false positives of the Bloom filter, the query results are not exact, which leads to additional bucket probes.

Let f denotes the false positive rate of the Bloom filter, X_i denotes the indicator variable that buckets in subtable T_i need to be accessed, and L denotes the number of buckets need to be probed. Although we get consecutive d bits during querying the Bloom filter, they can be regarded as independent queries in BF approximately. So the expectation of X_i is f ($E(X_i) = f$). For negative queries:

$$\begin{aligned} \overline{L_{neg}} &= E(L_{neg}) \\ &= E(\sum X_i + 1) \\ &= 1 + (d-1)f \end{aligned} \tag{1}$$

For positive queries, query for home items should probe all other abroad tables, and it is same as the negative queries. For abroad items, suppose the probability of an item in each subtable is equal, then on average abroad items may incur $\frac{(d-2)}{2}$ false positives. We define the abroad ratio β as the ratio between the number of abroad items and the number of insertions. Therefore:

$$\begin{aligned} \overline{L_{pos}} &= \beta E(L_{abroad}) + (1-\beta)E(L_{at-home}) \\ &= \beta(1 + \frac{(d-2)}{2}f) + (1-\beta)(1+(d-1)f) \\ &= 1 + ((1-\frac{\beta}{2})d - 1)f \end{aligned} \tag{2}$$

Suppose N is the number of insertions, there are βN items in the Bloom filter. Although the insertion procedure in our enhanced Bloom filter is different from that in the original one, the effect of these insertions can be regarded as independent because of items' independency. Thus, we apply the analysis of the original Bloom filter [31] to our one, i.e., given N, β and the length of the Bloom filter m, the optimum k is:

$$k = \frac{m}{n} * \ln 2 \tag{3}$$

The false positive rate is 2^{-k} under the optimum k.

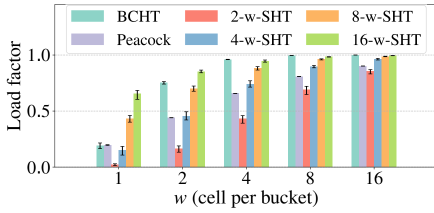

Fig. 2. Maximum load factor

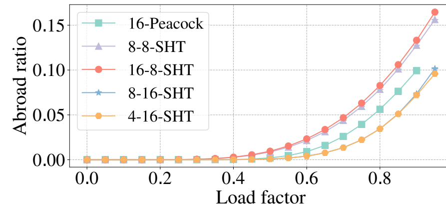

Fig. 3. Abroad ratio under different load factors

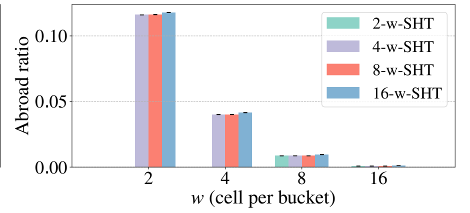

Fig. 4. Abroad ratio of SHT under different parameter settings (load factor = 0.5)

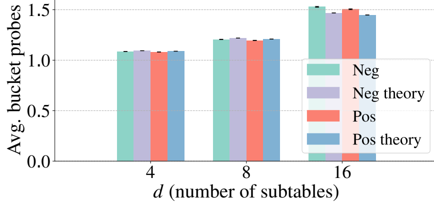

Fig. 5. Validations of our formulas of positive and negative queries

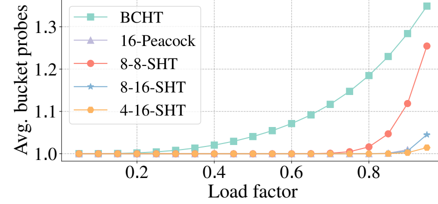

Fig. 6. Positive query performance under different load factors

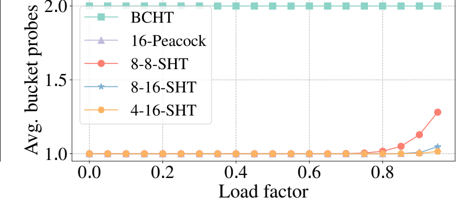

Fig. 7. Negative query performance under different load factors

VI. Evaluation

A. Experimental Setup

In the evaluation part, we explore the effect of parameters on performance in our SHT algorithm, and compare SHT with Peacock hashing and BCHT in terms of memory efficiency and query performance. We mainly use three metrics:

Load factor: We insert items into the hash table sequentially until the next item cannot be successfully inserted. The ratio of the number of the inserted items and the number of memory cells in the hash table is measured as the load factor which reflects the memory efficiency of slow memory. A higher load factor means less empty memory cells in the hash table.

Abroad ratio: The abroad ratio shows the number of items in the summary which reflects the memory efficiency of fast memory. A smaller abroad ratio means less fast memory is required to represent the summary. For peacock, the items in the backup tables are regarded as abroad items, because only these items need to be inserted into the summary. For BCHT, no summary is needed so we do not show this metric for it.

Average bucket probes in query: This regards to the query performance in the hash table. We measure the bucket probes in both positive queries and negative queries.

We generate 100 datasets, and each dataset contains 15M key value pairs. The keys and values are both 4-bytes integers and generated randomly from a uniform distribution. We guarantee the keys in each dataset are distinct. We use the first 10M items of each dataset as the insertion workload, and the rest 5M items as the negative query workload. The inserted items are used as the positive query workload. Each point in the figures shows the mean value of these 100 results, and we also plot the 5th and 95th percentile error bars in the figures.

We set the capacity (the number of memory cells) of each hash table to 10M, same as the size of insertion workload. For Peacock, we use the bucketized technique for collision resolving. The scaling factor r is set to 10 as recommended in its original paper [7], which means the common ratio of the size of two adjacent subtables is 10. We use 8 subtables for Peacock. The sizes of last two subtables are very small under this setting, so we assume the buckets in these two subtables can be stored in the fast memory, and the overhead of bucket probes of them can be ignored. For BCHT, we use only two hash functions, and 4 cells per bucket without specification. For convenience, we use **d-w-SHT** to denote SHT with d subtables and w cells per bucket, and use **w-Peacock** to denote peacock hashing with w cells per bucket.

B. Results and Analysis

As shown in Fig. 2, SHT achieves high load factors under different parameter settings. We also show the performance of peacock and BCHT under different w. With w increase, all these algorithms can achieve higher load factor. The growth speed of SHT is slower than BCHT but higher than peacock. For SHT, using more subtables can achieve higher load factor under the same w. Specifically, 8-8-SHT can achieve 96.17% load factor, and 4-16-SHT can achieve 96.50% load factor, while 16-Peacock can achieve 90.17% load factor.

Fig. 3 shows the abroad ratios trends of 16-Peacock and SHT during the insertion procedure. For convenience, we only show some of the lines. We observe that SHTs with the same w have almost the same abroad ratio during insertions, as shown in Fig. 4. Fig. 4 shows the abroad ratio of SHT when the load factor is 50%. (Some bars are missing because SHT cannot achieve a load factor of 50% under some parameters.) When d increases, the abroad ratio only slightly increases if we fix w, which means the number of cells per bucket is the major influencing factor to the abroad ratio. Back to Fig. 3, it shows that with $w = 16$, only 10% items will be inserted into the Bloom filter under 95% load factor, which is close to that of 16-Peacock when it achieving 90% load factor.

We validate the formulas in Fig. 5. In this experiment, we use SHT with the same w to avoid the influence of different abroad ratios. We fill up the tables to 80% load factor, and we estimate the abroad ratio is 8% according to Fig. 3. And we set the size and the number of hash functions of the Bloom filter to optimize the false positive rate. We can see that the empirical results is close to the theory results.

Finally, Fig. 6 and 7 shows the average bucket probes in positive queries and negative queries. We set the size of Bloom filter same as the number of memory cells, which means we need 1 bit per cell in the fast memory. We set the optimized k according to the size of the Bloom filter and the estimated abroad ratio. With the number of items increasing, all of these algorithms need more bucket probes in both positive and negative queries. Both SHT and peacock need much less bucket probes owing to filter function of the summary. Compared with peacock, 8-16-SHT and 4-16-SHT performs slightly better than 16-Peacock. Specifically, when the load factor is 90%, 8-16-SHT needs 1.0064 and 1.0068 bucket probes for positive queries and negative queries, respectively, while 16-Peacock needs 1.0096 and 1.0105 bucket probes for positive queries and negative queries, respectively. When the load factor is 95%, 8-16-SHT needs 1.045 and 1.048 bucket probes on average for positive queries and negative queries, respectively, while 16-Peacock cannot achieve this load factor.

VII. Conclusion

Hash tables are widely used in many security applications. In this paper, we propose a novel hash table scheme, named Shifting Hash Table (SHT), achieving high load factor and high query speed in terms of the bucket probes at the same time. The novelty of SHT lies in two aspects: 1) SHT divides the items into two kinds: at home and abroad, and we create 3 rules to make the the number of abroad items small; 2) we modify the Bloom filter into an enhanced one to significantly reduce the query speed. Theoretical analysis and experimental results show that our proposed Shifting Hash Table significantly outperforms the state-of-the-art.

References

[1] W. Xie, L. Xie, C. Zhang, Q. Zhang, and C. Tang, “Cloud-based rfid authentication,” in *2013 IEEE International Conference on RFID (RFID)*. IEEE, 2013, pp. 168–175.

[2] G. Tsudik, “Ya-trap: Yet another trivial rfid authentication protocol,” in *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW’06)*. IEEE, 2006, pp. 4–pp.

[3] H. Sun, X. Wang, R. Buyya, and J. Su, “Cloudeyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things (iot) devices,” *Software: Practice and Experience*, vol. 47, no. 3, pp. 421–441, 2017.

[4] D. Oh, D. Kim, and W. Ro, “A malicious pattern detection engine for embedded security systems in the internet of things,” *Sensors*, vol. 14, no. 12, pp. 24 188–24 211, 2014.

[5] R. Pagh and F. F. Rodler, *Cuckoo hashing*. Springer, Aug. 2001.

[6] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, “Fast hash table lookup using extended bloom filter: an aid to network processing,” *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 181–192, 2005.

[7] S. Kumar, J. Turner, and P. Crowley, “Peacock hashing: Deterministic and updatable hashing for high performance networking,” *Proceedings - IEEE INFOCOM*, pp. 556–564, 2008.

[8] S. Kumar and P. Crowley, “Segmented hash: An efficient hash table implementation for high performance networking subsystems,” *2005 Symposium on Architectures for Networking and Communications Systems, ANCS 2005*, no. 1, pp. 91–103, 2005.

[9] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[10] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, “Optimizing Bloom filter: Challenges, solutions, and comparisons,” *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

[11] S. Z. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, “Bloom filter with a false positive free zone,” in *IEEE INFOCOM*, 2018.

[12] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, “Noisy bloom filters for multi-set membership testing,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1. ACM, 2016, pp. 139–151.

[13] H. Dai, M. Li, and A. Liu, “Finding persistent items in distributed datasets,” in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1403–1411.

[14] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, “Finding persistent items in data streams,” *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[15] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, “Space efficient hash tables with worst case constant access time,” in *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 2003, pp. 271–282.

[16] O. Polychroniou, A. Raghavan, and K. A. Ross, “Rethinking simd vectorization for in-memory databases,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1493–1508.

[17] M. Dietzfelbinger and C. Weidling, “Balanced allocation and dictionaries with tightly packed constant size bins,” *Theoretical Computer Science*, vol. 380, no. 1-2, pp. 47–68, 2007.

[18] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, “Horton tables: Fast hash tables for in-memory data-intensive computing.” in *USENIX Annual Technical Conference*, 2016, pp. 281–294.

[19] A. Kirsch, M. Mitzenmacher, and U. Wieder, “More robust hashing: Cuckoo hashing with a stash,” *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2009.

[20] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “Silt: A memory-efficient, high-performance key-value store,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 1–13.

[21] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo filter: Practically better than bloom,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 75–88.

[22] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, “Algorithmic improvements for fast concurrent cuckoo hashing,” in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 27.

[23] D. Li, J. Li, and Z. Du, “Deterministic and efficient hash table lookup using discriminated vectors,” in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–6.

[24] T. Yang, B. Yin, H. Li, M. Shahzad, S. Uhlig, B. Cm, and X. Li, “Rectangular hash table: Bloom filter and bitmap assisted hash table with high speed,” in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 837–846.

[25] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, “Smartcuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX{ Association}, 2017, pp. 553–565.

[26] M. Herlihy, N. Shavit, and M. Tzafrir, “Hopscotch hashing,” in *Distributed Computing*. Springer, Sept. 2008, pp. 350–364.

[27] P. Zuo and Y. Hua, “A write-friendly hashing scheme for non-volatile memory systems,” in *Proc. MSST*, 2017.

[28] T. Song, Y. Yang, and P. Crowley, “Rwhash: Rewritable hash table for fast network processing with dynamic membership updates,” in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2017, pp. 142–152.

[29] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[30] W. Litwin, “Linear hashing: a new tool for file and table addressing.” in *VLDB*, vol. 80, 1980, pp. 1–3.

[31] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.