Abstract—Hash tables (HTs) are poorly designed for multiple memory accesses during IP lookup and they will critically affect throughput in high-speed routers. Therefore a high capacity HT with predictable lookup throughput is desirable. Legacy HT (LHT) with linked list offers constant search time, but suffers from memory overhead due to use of pointers. Fast HT of [1] has drawbacks of high access time when load factor is greater than 0.2; low memory utilization for low load factor; and memory overheads.

In this paper, we propose a novel hash architecture using parallel Bloom Filters. The proposed scheme called hierarchically indexed hash table (HIHT) generates memory address in binary so that pointers overhead in linked list is removed. Secondly, using on-chip memory, HIHT shows approximately $\log_2 f(\log_2 n + 4)/(8 \log_2 f + 2 \log_2 n + 10)$ times better space efficiency with less memory access compared to FHT, where $f$ is a false positive of BF and $n$ is the number of hash entries. Our analyses on memory access time and memory efficiency show that for large size tables, HIHT performs the best among LHT and FHT.

1. Introduction

As the demand of high speed and large scale router has surged up, a class of fast packet processing like packet classification and IP lookup has been critical data path functions for many emerging networking applications. Those functions have wide application in networking devices to support firewall, access control list and quality of service in several network domains. They need various matching conditions like longest prefix matching (LPM) or exact matching. An interesting approach is a scheme of Ternary Content Addressable Memory (TCAM) to achieve deterministic, high-speed LPM for packet classification and IP lookup [2–6].

Unlike TCAM with high cost and power consumption, an approach with a Bloom Filter (BF) has been widely used in literature on networking even on wireless sensor networks due to memory efficiency [1, 7–12]. A BF is essentially a compact representation of a set for membership testing. However, standard exact representations of sets such as hash tables and binary trees require at least $L$ bits per element to be stored, where $L$ is the size of an element, and often require additional space for pointers. [7] introduced the first algorithm to parallelly employ BFs for LPM essential to IP routing lookup. The set of BFs which can be implemented in hardware of parallel access on bit-addressable memory [13], performs a fast search on their own range of IP address whose size is either 128-bit in IPv6 or 32-bit in IPv4.

Traditionally for fast search a hash table (HT) has been widely used for performing fast associative lookups, which requires $O(1)$ average memory access per lookup. A typical application of an HT is network packet processing and many applications include per-flow state management of IP lookup and packet classification in high speed network router device. [1] claimed that a FHT with help of a BF improves the performance over a legacy hash table (LHT) by reducing the number of memory accesses needed for the most time-consuming lookups. This benefit comes by combining hashed linked lists with $k$ hash functions so that only the shortest linked list is used in the search. However, overlapping $k$ linked lists cause a bunch of disadvantages as well as a generic limitation by linked lists implementation. First of all, due to merging $k$ linked list there is a chance that duplicate items are saved on-chip memory and the overall rate of duplicate items ranges from 1 to 3 in the experiment of [1]. Although searching an item is expedited by choosing the shortest linked list, insertion and deletion operation takes approximately $k$ times. These operations are suitable for a dynamically changing set because any change in the set by one item needs $2k$ times of off-chip memory access. Besides time complexities of insertion and deletion operations to get better performance over an LHT FHT needs a plethora of buckets intact to reduce collision and holds a large portion of buckets on on-chip memory wasted for all time.

Finally, due to the inherited drawback of a BF, delete operation was designed by introduction of a counter of empirically-set 4 bits in each bucket. Since [14] suggested a scheme of delete operation by putting a counter in each bucket of an array for BFs and [15] observed that when there are $m \ln 2$ total counter increments spread over $m$ counters, with high probability the maximum counter value is $O(\log m)$ and hence only $O(\log \log m)$ bits are necessary for each counter. Hence [1] claimed that the probability that a linked list has more than 8 elements is $1.4e-6$ minuscule enough that 3 or 4 bits of counters are suitable. However, in mission-critical applications with an HT such as management of nuclear power and an airplane, this minuscule probability could come true in the long run of it causing a dreadful catastrophe.

Beyond previous approaches like TCAM with high cost and power consumption as well as a FHT with duplicate copies of items and shared linked lists needing more memory accesses, we want to propose a basic fundamental hash architecture of novelty, so that many packet processing applications can get benefit from it. Our HIHT has the following contributions;
HIHT is implemented in flat and contiguous memory space so that items are saved in off-chip memory in the order they are inserted. Therefore, it does not need pointers in off-chip memory which were necessary in linked list approaches of an LHT and a FHT.

Unlike a FHT, no duplicate items are saved in off-chip memory.

HIHT implements insert operation taking $O(1)$ time as well as delete operation taking almost $O(0)$ time on average while a FHT approximately takes $O(k)$ for both, where $k$ is the number of hash functions.

HIHT does not need counting BF so that the overhead of bits of counting BF is resolved.

The probability for an unsuccessful search is incredibly smaller than an LHT and a FHT as the probability of a successful search is still smaller than them.

After the existing LHT and FHT recapitulated in Sec. III, in Sec. IV basic notion of HIHT and false index path of it incurred by false positives of a series of BF's will be mentioned and analyzed with plots in Sec. V. Then related works and conclusion follow in Sec. VI and Sec. VII.

II. Problem Statement

Searching is the most time-consuming part of many applications. Along with sequential or binary search via key comparison, it is impossible to complete a search of $n$ items in fewer than $\log n$ comparison on average. In contrast, hashing with appropriate data structures sets up a one-to-one correspondence between a key and an index of an HT.

Designing insert and delete operations according to a new hash table decides the efficiency of the hashing method in information retrieval, i.e., search. Under reasonable assumptions, the expected time to search for an element in an HT is $O(1)$. Hence our scheme also provides stable operation of that complexity. The paramount concern in hashing is to reduce collision among keys because even if the hash function is well designed it is impossible to hash universal elements without any collisions. For example, the famous “birthday paradox” asserts that if 23 or more people are present in a room, chances are good that two of them will have the same month and day of birth. In other words, if we select a random function, which maps 23 keys into a table of 365 buckets, the probability that no two keys map into the same location is only 0.4927[16]. Although literature has proposed the collision resolution with open addressing and chaining method, we propose a novel collision-free hash architecture.

Also in light of memory utilization, the number of buckets affects the collision rate because generally the hash function is based on the size of total buckets. Hence as the number of buckets get increased over the number of items to hash, the rate of collision is reduced. However, by-product most of the buckets are not used during operation. The load factor defined as the number of elements over the number of buckets is a useful metric to measure efficiency of memory usage as well as collision rate and the length of linked list. In high speed applications of networking, at the reasonable cost of memory to reduce collision rate is very important so that fast processing a bunch of network packets is easily achieved. Furthermore, collision of two keys on the same bucket introduces linked list structure and therefore memory overhead about pointers. On contrast, our scheme provides memory efficient hash architecture along with the collision-free feature.

III. The Existing Hashing Architecture

For comparison, we recapitulate an LHT [17] and a FHT [1] and compare the probability of the length of searched linked list in an LHT as well as a FHT. We consider hash algorithm in which collisions are resolved by chaining since it has better performance than open addressing schemes. Also, a FHT is considered as Shared-node Fast Hash Table of [1] for comparison.

A. Legacy Hash Table

An LHT consists of an array of $m$ buckets with each bucket pointing to a linked list of items hashed into it. and load factor of an LHT is considered as the average length of lists in [18]. Let $X_i$ be a random variable for the length of the searched bucket. Then the probability that the length of searched bucket is more than $j$, on the condition that the bucket has more than one item is

$$Pr[X_i = j | X_i > 0] = \frac{Pr[X_i = j, X_i > 0]}{Pr[X_i > 0]}$$

$$= \frac{n!}{j!(1/m)^j(1 - 1/m)^{n-j}}/(1 - 1/(1/m)^j).$$

B. Fast Hash Table

Originally BF was designed to be a simple space-efficient randomized data structure for representing a set of keys to support membership queries in short response time. The scheme of a FHT extends LHT with BF's scheme to choose the shortest linked list in on-chip memory for exact match, which needs verification of existence of an item in off-chip memory. By sharing the hashed linked lists for an item it showed how the accesses to the off-chip memory, due to either collision or unsuccessful searches, can be reduced significantly.

[1] calculated $Pr[X_2 = s]$ which is the probability of random variable $X_2$, the smallest counter value among $k$ hashed buckets is $s$ shown as:

$$Pr[X_2 = s] = \sum_{j=1}^{s} d(j, k) \times q(j, s, j)$$

where $q(r, s, j) = \sum_{h=1}^{r} p(h, j) \times (1 - \sum_{h=1}^{r} p(h, j))^{j-1}$, $p(i, j) = ((m^r)!(1/m)^j(1 - 1/m)^{r-j})/(r-j-1)$, and $d(j, k)$ is the probability that the first $r$ hashes of an item produce exactly $j$ distinct values.

Although its analysis showed that $Pr[X_2 = s]$ is smaller than $Pr[X_1 = j]$ as shown in Fig. 1(a), the analysis did not consider two memory constraints; the bit size of a bucket, i.e. a pointer, and the bit size of a counter in the bucket.
For fair comparison we need to adjust variables like the number of buckets \( m_1, m_2 \) and load factors \( \alpha_1, \alpha_2 \) for an LHT and a FHT, respectively. Given values of \( m_1 \) and \( \alpha_1 \) for an LHT, the number of elements to hash is \( m_1 \alpha_1 \) and the memory usage in bit for an LHT, \( m_1 \log_2 n \), should be equal to \( m_2 \left( \log_2 (m_1 \alpha_1) + 4 \right) \) for a FHT where 4 is bit size of a counter. Therefore \( m_2 = (m_1 \log_2 (m_1 \alpha_1))/ (\log_2 (m_1 \alpha_1) + 4) \) and \( \alpha_2 = \alpha_1 \left( \log_2 (m_1 \alpha_1) + 4 \right)/ \log_2 (m_1 \alpha_1) \). Fig. 1 (a) indicates that the probability of \( \Pr(X_2 = j) \) for our adjusted-FHT is bigger than that of an old FHT because given memory size in bits for an LHT \( m_2 \) for a FHT is reduced, hence the load factor of adjusted-FHT is increased. Another missing point of [1]'s analysis is that as the load factor is increased the \( \Pr(X_2 = 2) \) is bigger than \( \Pr(X_2 = 1) \). Fig. 1 (b) shows the case that if \( \alpha_1 \) of an LHT is 0.14 \( \Pr(X_2 = 2) \) is almost the same as \( \Pr(X_2 = 1) \) and furthermore, \( \Pr(X_2 = 2) \) with \( \alpha_2 = 0.25 \) is bigger than \( \Pr(X_1 = 2) \) with \( \alpha_1 = 0.2 \). Furthermore we observe that as the load factor is bigger than 0.14 \( \Pr(X_2 = j) \) grows rapidly while \( \Pr(X_1 = j) \) increases slowly, \( j \geq 2 \).

Besides adjusted memory constraints, a FHT has a shortcoming that as long as the load factor is quite smaller than 0.14 where a FHT is better than an LHT, most of the buckets in on-chip memory remain intact and useless. Moreover empirically-set bit size of counter can lead to disaster. In the following section, we will propose new hashing architecture to deal with these problems.

IV. HIERARCHICALLY INDEXED HASH TABLE

To figure out the fundamental relationship among memory size, capacity of members, we present the mathematics about a BF and a false positive, or \( f \)-positive, and after this we introduce the mechanism of insert, query, and delete operation. A legacy BF for representing set \( S = \{ e_0, e_1, ..., e_{m-1} \} \) of \( n \) elements is described by an array of \( m \) bits, initially all set to 0. A BF uses set \( H \) of \( k \) independent hash functions \( h_0, h_1, ..., h_{k-1} \) with range \( [0:m-1] \). For mathematical convenience, we make a natural assumption that these hash functions map each item in the universe to a random number uniform over the range. For each element \( e_y \in S \), the bits indexed by \( h_y(e_y) \) are set, i.e. encoded, to 1 for \( 0 \leq y \leq k-1 \). A bit in the array can be set to 1 multiple times, but only the first change has an effect. To verify that item \( y \) is in \( S \), we check whether all bits in BF indicated by \( h_y(y) \) are set to 1. If not, then clearly \( y \) is not a member of \( S \). Even if all of them by \( h_y(y) \) are set to 1, sometimes item \( y \) may not belong to set \( S \) with a probability because of random gathering of \( k \) bits set to 1 in the array for independent \( e_y \)'s. Hence, a BF may yield a \( f \)-positive suggesting that \( y \) is in \( S \) even though it is not.

The probability of a \( f \)-positive can be formulated in a straightforward way, given our assumption that hash functions are perfectly random. Among \( m \) bits the chance of becoming 1 by one of \( h_y \) is \( 1/m \). All the elements of \( S \) are hashed \( k \) times into the BF, i.e. totally \( k \cdot n \) times, the probability that a specific bit is still 0 is asymptotically \( p = (1 - 1/m)^k = e^{-k/m} \). Then, the probability of a \( f \)-positive by randomly choosing \( k \) elements in \( m \) bits is

\[
f \geq \left\{ 1 - \left(1 - \frac{1}{m}\right)^k \right\} = (1 - p)^k \geq (1/2)^{k \ln 2}
\]

because independently \( k \) bits with probability of becoming 0, or \( p \), could become 1 when a membership test is requested and this probability is bounded according to the results of [15, 19]. After some algebraic manipulation, [15] claimed that the requirement of \( f \leq \varepsilon \) suggests

\[
m \geq n \frac{\log_2 (1/\varepsilon)}{\ln 2} = n \log_2 e \cdot \log_2 \frac{1}{\varepsilon}
\]

A. Insert (Encode) in Hierarchical Indexing Tree

With help of parallel access on bit-addressable memory [13, 9] introduced another axis of the design space, multiple binary predicates that were used to forward packets
to corresponding interface. Beyond schemes of multiple binary predicates and a FHT, we propose new hashing architecture capable of indexing a off-chip memory space in binary used to search an entry of a table, so-called exact matching. Unlike one BF for shared linked lists in [1], we design a hierarchical indexing tree (HIT) with a set of normal BFs in two kinds. An HIT for set $S$ of $n$ elements in power of 2 is composed of $r = \log_2 n$ layers and partitions the address space in a rectangle of $n \times r$ bits as shown in Fig. 2. For layer $i$, hereinafter $0 \leq i \leq r - 1$, a BF, or s-BF, covers a column group of the same bits, either 0 or 1, in the same address space. On the other hand, for layer $i$, hereinafter $s \leq i \leq r - 1$, a BF covers a group of mixed 0 and 1 in dual and let us call it d-BF. Suppose data structure $BF_i^j$ denote j-th BF in layer i and let all n elements are filled in off-chip memory sequentially from $0_0...0_{i-1}$ to $1_0...1_{i-1}$. If element $e \in S$ is to be inserted at memory address $A = a_0a_1...a_{i-1}$ where $a_i \in \{0, 1\}$, 0 $\leq i \leq r - 1$, a s-BF denoted $BF_i^a = \{a\}$ at each layer $i$, is involved to encode element e as the normal BF is. In case of layer $i$, a d-BF, $BF_i^{a_0...a_{i-1}}$, is involved. In this hierarchical partitioning and encoding, $BF_i^j$ at each layer $i$ takes care of $n^j = n/2^{i+1}$ elements of set $S$ while for layer $i$ $n^j = n/2^i$ elements are assigned to a d-BF. According to (4), $m$ is linearly proportional to $n$ in given $g$ so that even if each layer $i$ is divided by $n/m$ BF's $m^j$ is large enough to maintain $f^j$ for each $BF_i^j$ in an HIT.

Finally an HIT is comprised of 0-tree and 1-tree covering half of n elements in $0_1...0_{i-1}$ and the rest half in $1_1...1_{i-1}$, where $x_1, x_i \in \{0, 1\}$, $1 \leq i \leq r - 1$. Suppose there is a virtual root on top of 0-tree and 1-tree. In either 0-tree or 1-tree, a sub-tree consisting of one child sub-tree is called a linear tree, or i-tree. Fig. 2 shows a basic structure of our HIT consisting of 4 layers of s-BFs as well as d-BFs for 16 elements. The left side of Fig. 2 shows the binary address space with a set of s-BFs and d-BFs partitioning the address space and the right side shows the transformed dual trees, 0-tree and 1-tree, where each node represents $BF_i^j$. In the sixth insertion of element $n_5$ located in memory address $0101$, $BF_0^s$ at layer 0, $BF_1^s$ at layer 1, $BF_2^s$ at layer 2, and $BF_3^s$ at layer 3 are involved.

The detailed procedure of insert is shown in the following. The first vertically-lined for loop in Procedure insert is executed in parallel at each layer. Also the second for loop is done in parallel as conventional BF does. Therefore, the time complexity in on-chip memory is $O(1)$ on the condition that hash functions return indexes and each layer conducts hashing in parallel. Moreover, the complexity of Procedure insert for memory access $M[A]$ where $M$ is off-chip memory and $A$ is given address is $O(1)$ because item $e$ is saved in designated address $A$ as shown in the last line. In contrast, a FHT claimed to take time complexity $O(nk^2/m + k)$.

**Procedure insert**

| Data: two parameters; item e and its given address $A = a_0a_1...a_{i-1}$ |
|**Result:** Encoded HIT about $e$ |
| 1 for layer $i = 0$ to $r - 1$ in HIHT do /\* On-chip Op. */ |
| 2 for $t = 0$ to $k - 1$ do |
| 3 $g = h_t(e)$; |
| 4 if $i \leq s$ then |
| 5 $j = a_0...a_{i-1}$; |
| 6 $BF_i^a[g] = 1$; |
| 7 else |
| 8 $j = a_0...a_{i-1}$; $x = a_i$; |
| 9 $BF_i^{x}[g] = 1$; |
| 10 end |
| 11 end |
| 12 end |
| 13 $M[A] = e$; /* Memory Access */ |

**B. Query Operation**

Once all elements are saved in off-chip memory in order and encoded in a set of BFs in on-chip memory, the remaining and ultimate goal of an HT is to search an item in it by query operation of short time. There are two kinds of search patterns: a successful search where an item is to be searched out in an HT but it takes time, and an unsuccessful search where an item is relentlessly searched although it is not in a HT.

In inserting an element in an array of BFs at each layer, the same procedure made in the conventional BF is conducted, that is, each relevant BF is hashed by $k$ hash functions and the $k$ bit elements is set to 1. Data structure $BF_i^{a_0...a_{i-1}}$ covers only either 0 or 0 bits as shown at lined boxes in Fig. 2. Alternatively let $BF_i^0$ and $BF_i^1$ denote as for 0 and 1 bits, respectively as well. Each covers $n_i$ elements with $n_i = n^{r_i} \log_2 e \log_2 (1/f^i)$ bits where $f^i$ is a predefined $f$-positive rate for each layer $i$. In other layer $i$, $BF_i^{a_0...a_{i-1}}$ is a dual mix of $BF_i^0$ and $BF_i^1$ which cover all 0 or 1 bits as shown in a dotted box of Fig. 2.

Suppose $B_{j,e}$ or $B_{x,e}$, $x \in \{0, 1\}$, denote a random variable of a result from corresponding data structure $BF_i^j$ shown in Fig. 3. In other way $B_{j,e}$ may be described in $(B_0^j, B_1^j)$.
for BF0'y and BF1'y covering 0-bits and 1-bits, respectively. Depending on the membership status of element e in layer i, B' is defined as following: 1) B' = 1 for e ∈ BF'y, 2) B' = 0 for e ∉ BF'y with 1 - f', 3) B' = 1 for e ∉ BF'y with f'. Similarly Bx'y', hereinafter x ∈ {0, 1} is defined as

\[
Bx'y' = \begin{cases} 
1 & \text{for } e \in BFx'y' \\
0 & \text{for } e \notin BFx'y' \text{ with } (1 - f') \\
1 & \text{for } e \notin BFx'y' \text{ with } f'
\end{cases}
\]

where a f-positive f' and f' is derived from Eq. (3) and (4) with proper n', m', b', and n'.

Before we step into two kinds of searches with possible false access further, let us introduce definitions of index path, index segment, false index path and false segment as well as important properties coming from an HIT; cumulative false positive by false segment and false positive by offspring.

**Definition 1 (Index Path vs. Index Segment)**

In a HIT an index path, or i-path is a series of B's hierarchically connected each other from layer 0 to layer r - 1 making a sequence of address bits. The sequence of bits is also matched with an arbitrary memory address for saving an element. If the size of sequence of bits in connection of the series of B's is less than r we call it index segment, or i-segment.

In an HIT, besides an i-path dedicated to an item, false access to off-chip memory due to f-positives from irrelevant BFs is possible. For example, suppose item e is inserted with i-path 010111 in Fig. 2 and then a query to e is made. The result of query gives an ambiguous 00010111 due to f-positives of BF1' and BF2'. Therefore, this ambiguity makes two off-chip memory accesses. In i-path of size r, there are totally 2^r - 1 cases of false access because each random variable B', which will be defined in the following, is independent and identically distributed, i.i.d. Besides the definition of an i-path, we define false index path by result of query query, leading to false access, after random variable A is defined as following.

Suppose A' in binary value be a random variable based on combination of a set of BFs at layer i for address bit a_i.

\[
A' = \begin{cases} 
1 & \text{for } (\cdots \bullet B'_1 \bullet \cdots) \bullet (\cdots \bullet B'_0 \bullet \cdots) \\
0 & \text{for } (\cdots \bullet B'_1 \bullet \cdots) \bullet (\cdots \bullet B'_0 \bullet \cdots) \text{ otherwise false index and double accesses}
\end{cases}
\]

where hereinafter • and ⋅ are AND and OR operation, respectively and j0 and j1 are index variables of BFs contributing to foaming address bit 0 and address bit 1 on the layer, respectively. The boxed line at the right of Fig. 3 show the combination logic about A'. Finally random variable A in size r is defined to be a binary sequence of random variables A', that is A = A'A'1...A'r'. This value of random variable A is used as an index to off-chip memory when an item is given to a HIT.

**Definition 2 (False Index Path vs. False Segment)**

In query operation, from hierarchically consecutive layers a group of B's not pertaining to an i-path can be formed in a series of size r. And to become a false index path, or f-path this series needs to be connected to a valid i-path or to be a completely different path of size r independent of the i-path in a HIT. Also we call the group attached to valid i-path false segment, or f-segment. This false segment is compatible with the length of searched linked list in [1].

After A' is set by a combination logic box in Fig. 3, it is possible that there is a series of consecutive f-positives in size range 1 to r due to a f-positive of each BF, given query operation. This series should be attached to the existing i-path or be a complete i-path; Otherwise the series needs to be removed because of definition of f-path, BFs in the series needs to be hierarchically connected in a HIT, and must be in size r. For example of previous 0101 for B0, even if B1 makes a f-positive right after query, there is no f-segment starting from the B1. Now the following theorems calculate the probabilities of f-path and the number of many f-paths in a HIT.

**Theorem 1 (Cumulative False Positives of False Segment)**

Suppose there is an i-path a_0...a_r-1 in an HIT after query. For f-segment L in size l to become an f-path, Bx'r-1, x ∈ {0, 1} on the f-segment should be a f-positive as well as all its ancestors B't, r - l ≤ t < r-1 be. Furthermore, the probability of the f-segment is the product of f-positives of all ancestors B't as well as Bx'r'. Let us call it cumulative false positive of the f-segment (See Appendix A for proof).

Fig. 4 shows the example of Theorem 1 with 1 i-path and 3 f-paths in an HIT. The dark lined and dashed boxes in a series of a0a1a2a3a4a5 is an i-path. The probability of the cumulative false positives of f-segment b2b3b4b5 becoming
f-path $a_0a_1b_3b_4b_5$ is $\Sigma_{i=2}^5 f_i$. Also, the rest f-paths, $c_0...c_5$ and $c_0...d_3d_4d_5$, have $\Sigma_{i=0}^5 f_i$ for cumulative false positives.

![Diagram](image)

**Fig. 4.** Example of Theorem 1

**Theorem 2 (Cumulative False Positives by Offspring)**

To have $n_p$ f-segments of size 1, binary tree $T$ rooted at $B^l$ with depth $l$ needs its left child tree $T_i$ to have $i$ f-segments and its right child tree $T_r$ to have $(n_p-i)$ f-segments, $0 \leq i \leq n_p$, while $B^l$ needs to be a f-positive. Therefore, the cumulative false positive by offspring for binary tree $T$ with its sub-trees, $F_T$ is the production of cumulative false positive by offspring of its left child and right child with itself, $f^i \cdot (F_{T_l} + F_{T_r})$ (See Appendix B for proof).

Fig. 5 shows the cases of Theorem 2; the right child tree $T_r$ that does not have any f-segment and both $T_l$ and $T_r$ that have any f-segment. By definition of f-segment to have any of it, $T_i$ must be a f-positive. Now that $B^l$ is a f-positive, f-segment in $T_l$ of size $l-1$ becomes a f-segment of size $l$. Therefore, automatically $T$ has the same number of f-segments from $T_l$ due to its f-positive, i.e. 2 f-segments. In addition two trees $T_l$ and $T_r$ in the right case of Fig. 5 contribute the summation of f-segments of $T$ in total 3 f-segments because $B^l$ is a f-positive and each of $T_l$ and $T_r$ has its own f-segments.

![Diagram](image)

**Fig. 5.** Example of Theorem 2

We have shown two basic theorems in a HIT. Each $B^l_i$ on the same layer $i$ has the same f-positive $f^l_i$ while it is different from that of other layers. Generally a f-positive is defined in range of less than 1 and even far less than $10^{-3}$ to maintain single off-chip memory access per lookup as much as possible as shown previous Fig. 1. Furthermore, cumulative false positives of f-segments in Theorem 1 and 2 is far less than single f-positive. To utilize this effect of decreasing cumulative false positives from Theorem 1 and 2 resulting in saving on-chip memory, we need to set the f-positive of $B^l_j$ of layer $j$, $1 \leq j \leq r-2$, to less than 1/2 while f-positives of the root and the $r$-s layer in the bottom of a HIT, $B^0_i$ and $B^r_j$, are to be set to desired false positive $f_d$.

1) **False Indexing in Successful Search:**

We have derived two cumulative false positives of f-segment and f-path in a HIT. Now we derive and calculate a probability of the number of false accesses in successful search in query. The next important step from Theorem 1 and 2 is how to recognize and cancel off a series of false positives randomly scattered in a HIT which does not belong to the definition of f-path so that the possibility of false access can be reduced. The basic fact in a HIT is that if a parent $BF_p$ returns 0, a series of f-positives starting downward from one of it children $BF_c$ should be cancelled off even if $BF_f$ has cumulative false positives by offspring. Let us call it fact 1. Reversely, so-called fact 2 even if a parent $BF_p$ returns 1, if both children $BF_c$s do not have cumulative false positives, any f-segment from a tree rooted at $BF_p$ upward should not be a part of false f-path because the tree can not foam f-segment. With two basic facts, the following validation method of index segment existing in a HIT shows a solution to remove an irrelevant series of f-positives.

**Theorem 3 (Validation of False Segment in a Hit)**

In triangle-downward, or T-DW, in layer $i$ involving three BFs, one $B^l_i$ and its children $B^{l+1}_i$s, if $B^l_i$ is 0 cancel off the values of $B^{l+1}_i$s to value 0 even if they are 1, $0 \leq i \leq r-2$. Do T-DW on every triangle of three BFs, consecutively from layer 0 of a HIT to layer $r-2$. After triangle-downward step, in triangle-upward, in short t-upward on layer $i$ involving two $B^l_i$ and $B^{l+1}_i$, and their parent $B^{l+2}_i$, $1 \leq i \leq r-1$, if both $B^{l+1}_i$ and $B^{l+2}_i$ return 0, cancel off the returned value of $B^{l+3}_i$ even if it returns 1. Do t-upward consecutively from layer $r-1$ to layer 1. After these ordered T-DW and t-upward steps, any consecutive sequence of f-positives not foaming false segment is cancelled off so that only false index paths are left (See Appendix C for proof).

The same effect of Theorem 3 on linear-upward, or L-DW, and linear-downward, or L-DW, for linear tree, or l-tree configured with one parent and one child happens on each layer $i$. Fig. 2 shows l-tree starting from layer 2 and consisting of one s-BF and d-BFs, in this case only one d-BF. On l-tree the following modification of Theorem 2 is applied; a modification that in L-DW if parent $BF_p$ returns (00), the value of its child $BF_c$ is cancelled off even if the value is one of (01), (10), and (11). Likewise in L-UW if child $BF_c$ returns (00), the value of its parent $BF_p$ is cancelled off even if the value is one of (01), (10), and (11). Fig. 6 shows the instance of the effect of Theorem 3.
number of \( f \)-segments exist in \( d \)-trees and \( j \)-tree. In \( l_{s-1} \)-tree consisting of one \( s \)-BF on layer \( s \) and \( r \)-s \( d \)-BFs on the rest of layers, the probability that there is no false segment in the tree is the summation of two cases; 1) whether \( B_r^{-1} \) is a \( f \)-positive and the \( l_{s-1} \)-tree does not have false segment and 2) whether \( B_r^{-1} \) does not have a \( f \)-positive while \( l_{s-1} \)-tree has at most \( 2^{s-2} \) false segments, that is, \( P(s-1, P(0)+\sum_{i=0}^{s-2} (1-f_i^{r-1}) P(i) \) where probability \( P(n) \) that \( i \)-tree has \( n \) false segments is defined recursively as

\[
P(n) = \begin{cases} 
2f^{r-1}P(r-1)(n-1) + (f^r)^2 P(r-1)(n-r) & \text{if } s \leq i \leq r-2 \\
2f^{r-1}P(r-1)(n-r) + (f^r)^2 P(r-1)(n-r) & \text{if } i = r-1, \ldots, r-1 .
\end{cases}
\]

The base case of Eq. (7), \( n_r = 2^r, z \in N_0 , \) makes \( P(n_r) \) because all \( d \)-BFs have two false positives and the number of \( f \)-segments from a sequence of \( z \)-BFs is \( 2^z \). Also in \( d \)-tree comprised of a set of \( s \)-BFs on \( s \)-layers and \( r \)-s \( d \)-BFs on \( r \)-s layers, the probability that there is no false segment in the tree is the summation of two cases, 1) whether \( B_r \) is a \( f \)-positive and two \( d \)-trees do not have false segment and 2) whether \( B_r \) does not have a \( f \)-positive while two \( d \)-trees have at most \( 2^{r-2} \) false segments. Moreover, based on Theorem 2 probability \( P(n) \), \( 0 \leq i \leq s-2 \) that \( d \)-tree has \( n \) false segments is defined recursively as in case 1 \( \leq n_r \leq 2^{r-1} \).

\[
P(n) = \sum_{i=n_0}^{n_1} \sum_{t=0}^{2^{r-1}-2} \sum_{i=0}^{2^{r-1}-2} P(i) P(t) \]

and in case \( n_r = 0 \)

\[
P(n) = \sum_{i=n_0}^{n_1} \sum_{t=0}^{2^{r-1}-2} \sum_{i=0}^{2^{r-1}-2} P(i) P(t) \]

where the interconnection cases to \( l_{s-1} \)-tree are two cases of whether \( l_{s-1} \)-tree has 0 or \( n' \) \( f \)-segments. The corresponding probabilities are \( P(s-1, P(n') \) for \( (n', f^r) \) and \( P(s-1, P(0) = f^r-1 P(0) + (1 - f^r) \sum_{i=0}^{s-1} P(i) \) respectively.

Although \( d \)-trees, \( 0 \leq i \leq s-2 \), and one \( l_{s-1} \)-tree must have at least one false segment, if any exists, the probability of getting more than 1 false segments in the \( d \)-tree and the \( l_{s-1} \)-tree is minuscule compared to a desired false indexing probability because the desired false indexing probability is around \( 10^{-3} \) as set in [1] and cumulative false positives for false segment with more than 3 \( B \)-BFs is extremely small. The following plot shows this claim. Fig. 8 shows \( P(n) \) and \( P(n) \) of a 

Where the gaps among \( P(n) \) are bigger than those of \( P(n) \) because \( f^r \) is much smaller than \( f^r \). Also, a group of lines for \( P(n) \) have much smaller values than a group of \( P(n) \) because as long as a length of false segment is getting bigger, \( P(n) \) becomes much smaller based on Theorem 1 and 2. The noticeable gap between \( P(n) \) and the rest of \( P(n) \) happens because \( f_0 \) is set \( 2^{-10} \) while \( f^r \) is set \( 2^{-2} \).

Now we need to figure out the number of false accesses to off-chip memory in successful search. Suppose random variable \( X \), be the number of false paths from 0-tree and \( l \)-tree given valid \( i \)-path for an item in query operation. Then
One of needs of lower probability in unsuccessful search stems from anti-virus search in fingerprint [20].

Unlike successful search, in unsuccessful search there is no valid index path meaning that all BF's return 1 as a f-positive. But there is a chance that each of 0-tree and 1-tree can give plural f-paths. In contrast to one f-positive in [1] leading to off-chip memory access, one f-path by a series of f-positives of hierarchically-connected $B_i$ in each layer $i$ of a HIT becomes to one off-chip memory access. Therefore, we expect far less probability because of the product of each f-positive probability. Like $A_i$, suppose $A'_i$ as

$$A'_i = \begin{cases} 
1 & \text{for } (\cdots \ B'_i \cdots ) \cdots (\cdots \ B_0 \cdots ); \text{ f-positive} \\
0 & \text{for } (\cdots \ B'_i \cdots ) \cdots (\cdots \ B_0 \cdots ); \text{ f-positive} \\
X & \text{for } (\cdots \ B'_i \cdots ) \cdots (\cdots \ B_0 \cdots ); \text{ double acc.} \\
A & \text{for } (\cdots \ B'_i \cdots ) \cdots (\cdots \ B_0 \cdots ) 
\end{cases}$$

where $A$ means there is no f-positive from BF's on layer $i$. Compared to successful search, an interesting point in unsuccessful search is that any of $A'_i$ becomes $A$ the whole address $A$ is void because to be legal address each of $A'_i$ should be at least one of 0, 1, 0, 0, 0, 0. Therefore, the probability that random variable $X_i$ for the number of f-paths in unsuccessful search on a HIT is 0 becomes

$$Pr[X_i = 0] = 1 - \prod_{i=0}^{n-1} Pr[A'_i = A] = 1 - \prod_{i=0}^{n-1} (1 - Pr[A'_i = A])$$

Extra off-chip memory access happens when $X_i > 0$ where 0- and 1-tree together make $v$ f-paths or only one of the trees makes $v$ f-paths as described in the following

$$Pr[X_i = v] = \sum_{i=0}^{v} \prod_{i=1}^{n-1} Pr[A'_i = A]$$

where hereinafter $Pr_{d_0}^{d_1}(t)$ indicates $Pr(t)$ of 0-tree while $Pr_{l_1}^{d_1}(t)$ indicates $Pr(t)$ of 1-tree as defined in Eq. (9).

The detail of query operation is shown in Procedure query only considering on-chip operation. Although it looks complex, the time complexity of it is $O(1)$ on the condition that the operations of T-DW and T-UW bound so that making only false segments and false index path left in a HIT is performed in constant time like hashing. This is possible because operations of T-DW and T-UW need AND and OR gate operation between neighbor layers and it happens in series of layers, which would take $2(r-1) \cdot t_i$ where $t_i$ is gate delay of AND or OR gate. Therefore, simply path delay caused by gate delays in a HIT can be bound in constant time. As to memory access related to the size of returned $S_i$, the complexities are $O(E[X_i] + 1)$ or $O(E[X_i])$ on average for successful and unsuccessful search, respectively, where $E[X_i]$ and $E[X_a]$ can be derived from Eqs. (10), (11), (13) and (14) as

$$E[X_i] = \sum_{t=0}^{t_i} t \cdot Pr(X_i = t), \quad E[X_a] = \sum_{t=0}^{t_i} t \cdot Pr(X_a = t)$$

and they are considered $O(1)$ because $Pr(X_i = 1) \gg Pr(X_i = t)$ and $Pr(X_a = 1) \gg Pr(X_a = t), t > 1$.
Procedure query

Data: two parameters; HIT of 0-tree and 1-tree and item e
Result: Set of A = A₀...A⁻¹ including false index paths
1 for layer i = 0 to r - 1 in HIHTT do
2 | Bᵢ = H(e) ;
3 end
4 for i = 0 to r - 2 do /* T-DW */
5 if Bᵢ = 0 & i < s then Bᵢ⁺ = Bᵢ⁺ = 0 ;
6 if Bᵢ = (0)₂ & i ≥ s then Bᵢ⁺ = (0)₂ ;
7 end
8 for i = r - 2 to 0 do /* T-UV */
9 if Bᵢ = 0 & i < s then Bᵢ⁺ = 0 ;
10 if Bᵢ = (0)₂ & i ≥ s then Bᵢ⁺ = (0)₂ ;
11 end
12 for layer i = 0 to r - 1 in HIHTT do
13 if (-Bᵢ⁺) then A ⓘ = 1 ;
14 else if (-Bᵢ⁺) then A ⓘ = 0 ;
15 else if (-Bᵢ⁺) then A ⓘ = X ; D = D ∪ i ; /* false indexing */
16 A ⓘ = 0 ; return 0 ;
17 else A ⓘ = X ; D = D ∪ i ; /* false indexing */
18 end
19 foreach t in D do /* make an address set */
20 | Sᵦ = Sᵦ ∪ A₀...A⁻¹ ; Sᵦ = Sᵦ ∪ A₀...A⁻¹ ;
21 end
22 return Sᵦ ; /* No off-chip memory access */

C. Delete Operation

Delete operation is not as much easy as insert because basic BF does not support deletion of an item which was encoded in the BF. As shown in Fig. 10 there are dual HITs in on-chip memory to rotate a target HIT for insert operation and another target HIT for delete operation. Once one HIT is full for previous n elements, query operation stays with the HIT. But if set S is dynamic, but limited in size n, a new HIT needs to accommodate new elements for insert and provide delete operation of encoded items in the old HIT. Therefore, the new HIT takes care of new coming elements for insert by setting on a bit in valid bit array, with new memory address while the old HIT handles delete operation by simply setting off a bit in valid bit array indicated by the corresponding HIT. Checking valid bit array with index given by each HIT makes it sure that unnecessary access to off-chip memory is prevented within HITs. Also, once all n elements are encoded in one HIT, the other HIT needs to be flushed out for next insert operation.

Like query operation it is possible that there is f-path besides valid i-path about an item. Therefore, when random variable Z is denoted as the number of accesses to off-chip memory, the average memory access about delete operation on the condition of existence of a target item, i.e. in successful deletion, is

\[ E[Z] = \sum_{i=1}^{n} v \cdot Pr(Z = v) = \sum_{i=1}^{n} v \cdot \sum_{t=0}^{\infty} (v \cdot \mathbb{P}_l(t) \cdot \mathbb{P}_u(v - t)). \]  

The detail procedure of delete operation is shown in Procedure delete. The complexity of it in on-chip memory is \( O(1) \) based on the complexity of Query is \( O(1) \). The complexity of memory access is \( O(E[Z]) \) on average for successful deletion. It is to be constant as \( O(E[X]) \) is.
Moreover, from the observation of Fig. 8 $O(E^2 |Z|)$ can be considered as $O(0)$ on average.

**Procedure delete**

**Data**: three parameters; two HITs of $l$- and $r$-HIT and item $e$

**Result**: Updated $V_r$ or $V_l$

1. $S_l=\text{Query}(l\text{-HIT}, e)$; /* Only on-chip Op. */
2. $S_r=\text{Query}(r\text{-HIT}, e)$; /* Only on-chip Op. */
3. if $||S_l| \cup |S_r|| = 1$ then $V_r[A] = 0$, where $||S_l|| = 1$
   4. else if $||S_l| \cup |S_r|| > 1$ then
   5. for each $A \in S_l, t \in [l, r)$ do /* Off-chip Op. */
   6. if $V_r[A] = 1$ And $M[A] = e$ then $V_r[A] = 0$;

V. ANALYSIS AND SIMULATION

In this section, we will present analysis of average access time and memory efficiency about three schemes; an LHT, a FHT and a HIHT. Also, we will perform two simulation works for measuring the numbers of operated inserts and deletes which depend on $k$ in a FHT, but are constants in a HIHT, and for figuring out the off-chip memory usage. A class of universal hash functions are suitable for hardware implementation [21]. For any member item $e$ with $b$ bits representation in $e \leq x_1, x_2, \ldots, x_b$ the $l$-th hash function over $e$, $h_l(x)$ is calculated as

$$h_l(e) = (d_1 \cdot x_1) \oplus (d_2 \cdot x_2) \cdots \oplus (d_b \cdot x_b)$$ (17)

where $\oplus$ is a bitwise XOR gate. For simulations in Sec. V-B and V-D we conducted $5 \cdot 10^5$ runs with different seeds.

A. Average Access Time

For an LHT, the load factor $\alpha_1$ can be given as $n_1/m_1$ where $n_1$ is the number of items and $m_1$ is the number of buckets used to point an address of off-chip memory after hashing. Let $T_l^i$ and $T_l^f$ denote the time for an average successful and unsuccessful search, respectively. Then they are defined as the following

$$T_l^i = 1 + \alpha_1/2 - 1/2m_1,$$

$$T_l^f = \alpha_1.$$

To evaluate the average search time, we introduce another parameter $p_2$ which denotes the frequency of searches which are finally successful in finding an item in off-chip memory. Also, $p_0 = 1 - p_2$ denotes the frequency of unsuccessful searches. With these notations, the average search time $T_1$ can be expressed as

$$T_1 = p_1T_l^i + p_2T_l^f = p_1\left(1 + \frac{n_1}{2m_1}\right) + (1 - p_2)\frac{n_1}{m_1};$$ (18)

For a FHT, let $E_f$ be the expected length of linked list in the FHT for an item and $E_f$ be the expected length of linked list in the FHT for a false positive match. $E_f$ can be derived from the average number of items for which all buckets length $> j$, or $n \cdot B(n-1) \cdot k, 1/m, (j-1)^k$ where $B(n, 1/m, > j) = \sum_{i=0}^{n} (\binom{n}{i} (1/m)^i (1-1/m)^{n-i})$. Also $E_f$ can be derived from Eq. (2). So the average search time $T_2$ is

$$T_2 = p_1E_f + p_2fE_f = p_1E_f + (1 - p_2)(1/2)^{(m/n)n^2} E_f$$ (19)

Now for HIHT, Eq. (10) and (11) for $E_p$ and Eq. (13) and (14) for $E_f$ are involved to get an average search time $T_3$ as the following

$$T_3 = p_1E_f + p_2fE_f = p_1E_f + (1 - p_2)E_f$$

$$= p_1 \sum_{l} \frac{E_f}{l} \text{Pr}[S_l = l - 1] + (1 - p_2) \sum_{l} \frac{E_f}{l} \text{Pr}[S_l = l]$$ (20)

B. Complexity of Operations in Off-chip Memory

Table I summarizes the complexities of off-chip memory access about insert, query, delete operations in each of an LHT, a FHT and a HIHT. The big difference comes in a FHT, which are the complexities of insert and delete operations depending on variables $n$, $k$, and $m$ while others are constant. Fig. 11 shows the cumulative number of

<table>
<thead>
<tr>
<th>Operation</th>
<th>insert</th>
<th>query</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHT</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>FHT</td>
<td>$O(nk^2/m + k)$</td>
<td>$O(1)$</td>
<td>$O(nk^2/m + k)$</td>
</tr>
<tr>
<td>HIHT</td>
<td>$1$</td>
<td>$O(1)$</td>
<td>$O(0)$</td>
</tr>
</tbody>
</table>

**TABLE I**

Comparisons of each operation of LHT, FHT, and HIHT

1. In optimal configuration, $O(1)$.
2. In detail, $O(p_l(1 + E_f(S_l) + p_2E_f(S_l)))$.
3. Approximately $O(0)$ due to $O(E^2 |Z|)$.

![Fig. 11. Measuring the number of off-chip memory accesses after a HT is filled. l=insert, q=query, d=delete. m = 128K, f_2 = 2^{-10} and k = 10](image-url)
C. Memory Usage in On-Chip

Denote $f_{d} = 2^{-w}$ desired false positive probability of BF as set in [1] and let $f_{d}^{i}$ of layer 0 be $f_{d}$, and $f_{d}^{i+1}$ of layer $i+1$ be $f_{d}^{i}/2$ while the rest layers have false positive $1/2$. Then according to Eq. 4 for the requirement of $f_{d}$ the total memory usage for HIHT $M_{t}$ counts two copies of HITs in the summation of $l$-HIT and $r$-HIT with layer 0 and up to layer $r-1$ plus two arrays of valid bits and $B_{j}$ residing in $l$-HIT and $r$-HIT.

$$
M_{t} = 2 \times (2\beta n^{w} \cdot w + 4\beta n^{1} + \cdots + 2^{i}\beta n^{w-1}) + 2^{i+1}\beta n^{w}(w+3) + \cdots + 2^{i} \beta n^{w-1}(w+3) + 2n
$$

$$
= 2\beta ((r-s+1)w + 3r + s - 4)n + 4n.
$$

(21)

where $\beta = 1.44$. In contrary, the memory usage of a FHT is $M_{2} = \beta wn + 4\beta n = \beta n(r+4)n$ considering 4 bits of counts. Therefore, memory efficiency ratio $R_{2,3}$ of $M_{2}$ to $M_{t}$ for $s=r-3$ becomes

$$
R_{2,3} = \frac{\beta n(r+4)}{8w(2w+10) + 4n} = \frac{w(r+4)}{8w+2r+10}
$$

(22)

on the condition that $s \geq 10$, $r \geq 10$ and $4n$ in Eq. (21) is negligible to $M_{2}$. Similarly memory efficiency ratio $R_{1,3}$ of $M_{1}$ to $M_{t}$ for $s=3$ is

$$
R_{1,3} = \frac{\beta n w r}{8w(2w+10) + 4n}.
$$

(23)

Fig. 12 shows that two ratios $R_{2,3}$ and $R_{1,3}$ calculated from Eq. (22) and (23) in range [10 : 20] for $w$ and range [10 : 30] for $r$. The change of $r$ makes bigger benefit in both $R_{2,3}$ and $R_{1,3}$ than the change of $w$ does. Also, this fact supports your claim that our HIT gives space- and time-efficient hashing architecture because as long as the depth of HIT is bigger $R_{2,3}$ is increased while $Pr(X_{i})$ and $Pr(X_{o})$ are getting minuscule. In all ranges of $r$ and $w$, $R_{2,3}$ is bigger than $R_{1,3}$ because a FHT needs counters for all $m$ buckets.

D. Memory Usage in Off-Chip

It is very hard to figure out the characteristics of the number of duplicated items because of real hash function implementation as well as dynamic order of insert and delete on items. However, although [1] showed that up to 3 times of memory overhead for duplicate happened in an experimental result, we illustrate one of the worst cases that coming items are duplicated more than 3 times with a probability. To derive the function of the number of duplicated items in one case as shown in Fig. 13 among the other worst cases, suppose item $x$ and $y$ put in buckets in order with one hash index of item $x$ overlapped another hash index of item $y$. The probability of this situation is $Pr(O_{2}) = \left(\frac{m}{w}ight)^{l} m^{a-1}$ because given one fixed bucket position, $2(k-1)$ indexes of hash functions from both $x$ and $y$ are distinctively distributed among remaining $m-1$ buckets. Let us call it event $O_{2}$. The counters indicated by hash functions of item $x$ do not consider item $y$. Therefore, whenever a new item is hashed and assigned to this duplicating-zone, or $d$-zone, the item should be prepended, not appended, meaning that the item is duplicated in off-chip memory for that hashing. Now the probability of duplicating a new item after event $O_{2}$ is the binomial distribution $B(k, (k-1)/m)$ because $k$ hash functions randomly assign the item either inside of $d$-zone or outside of it.

Now we need to figure out how many duplicates new inserts can cause. Suppose random variable $D_{2}$ be the number of hashing assigned into $d$-zone for one new item after event $O_{2}$ happens. Then

$$
Pr(D_{2} \geq j | O_{2}) = \sum_{t=j}^{k} \binom{k-1}{t} \left(\frac{k-1}{m}\right)^{t} \left(1 - \frac{k-1}{m}\right)^{k-t}.
$$

(24)

Now the ensuing $n-3$ items left over follows the same distribution, that is binomial distribution $B(n-2, Pr(D_{2} \geq j | O_{2}))$. Therefore, the probability that the remaining $n-2$ items have more than 3 can be derived from the Probability $B(n-2, Pr(D_{2} \geq 4 | O_{2}))$. Similarly the case of $O_{3}$ at the right side of Fig. 13 gives more chance of duplicating due to large area of $d$-zone. Therefore, with a probability related to binomial distribution $B(n-t, Pr(D_{2} \geq 4 | O_{2}))$, $k > 3$, there is a chance that the number of overall duplicated items is more than 3 times although the probability is minuscule.

Fig. 14 indicates memory usage for three schemes of LHT, FHT and HIHT on the condition that each of an item and a pointer is considered to be saved in one unit in off-chip memory. For instance any item for insert with 2 duplicates in a FHT needs 7 units of off-chip memory. This
kind of 2 times overhead will be worsen if a FHT is required to save an associate data about the item or a point to the associate data because a FHT only saves keys for exact matching while a general hash table is associated with a pair of a key and its associate data. However, a LHT needs the exactly two times of units for one item due to pointers. Fig. 14 shows that a FHT needs nearly 7 times of \( n \) units in off-chip memory while a HIHT only use exactly \( n \) units.

VI. RELATED WORKS
Since the burgeoning interest to BF in late 90’s, several types of BFs have been suggested in various application domains even with the disadvantage that a false positive and incapable operation of deletion besides our main comparative literature [1, 7, 9] and packet processing [8, 10–12].

If the element is to be deleted and set the corresponding bits to 0, setting a location to 0 disturbs the same location that is hashed to by some other element in the set. To avoid this problem, [14] introduced the idea of a counting BF where each entry in the BF is not a single bit but rather a small counter in a couple of bits. In Compressed Bloom filter [19], a server is sending a BF to several other servers over a network. Compressed BF improves performance when the filter is passed as a message and its transmission size is a limiting fact. [22] have introduced Approximate Concurrent State Machines. While similar in spirit to BFs, its scheme is based on a combination of hashing and fingerprints, using \( d \)-left hashing to obtain a near-perfect hash function in a dynamic setting. Surprisingly, it is found out that its data structure takes much less space than a comparable counting BF. [23] introduces Spectral Bloom Filter (SBF), an extension of the original BF to multi-sets, allowing the filtering of elements whose multiplicities are below a threshold given at query time. Using memory only slightly larger than that of the original BF, SBF supports queries on the multiplicities of individual keys with a guaranteed, small error probability. [24] provides duplicate detection in data streams of World Wide Web is utilized in various applications including fraud detection. In an application of overlay networks, continuous reconciliation of virtual topology by overlay management strives to establish paths with the most desirable end-to-end characteristics. [25]'s approximate reconciliation tree uses BFs on top of a tree structure to minimize the amount of data transmitted for verification.

In architecture domain, besides [7, 8, 11] for packet processing, to solve the synonym problem where the same physical cache line can be present at multiple locations in the cache due to their distinct virtual addresses, leading to potential data consistency issues, [26] tracked the address stream using BF and saved synonym lookup energy.

VII. CONCLUSION AND FUTURE WORK
We have proposed HIHT generating an off-chip memory address with a set of BFs on \( r \) layers. BFs in two HITs work systematically, or in parallel, so that a series of false positives scattered in HITs, but not attached to \( r \)-path for a given item is removed by Validity of False Segment procedure. After \( T-DW \) and \( T-UW \), each \( t \) in address bit is set by AND and OR logic in each layer. We derived probabilities \( P_t(X_i) \) and \( P_t(X_n) \) for successful and unsuccessful search, respectively and showed that the complexities of off-chip memory access on average, \( O(E[X_i]) \) and \( O(E[X_n]) \) are \( O(1) \), as those for insert and delete are while a FHT needs \( O(nk^2/m + k) \). Surprisingly, due to cowork with each layer in HITs unsuccessful search, important fact in fingerprint of security, showed near 0 probability on contrast to that of a FHT. Although we showed a colossal variety of benefits from HIHT it was not applied to real applications. As future work we consider a real application of a HIHT on IP router for core networks handling with large scale traffic.

APPENDIX

A. Proof of Theorem 1

**Proof:** By induction method, as basis suppose there is a given \( i \)-path \( a_0a_1...a_{r-1} \) and \( l \) be 1, meaning that except the first \( r-1 \) address bits, \( a_0...a_{r-2} \), the last address bit indicates a reverse bit, i.e. \( a_{r-1} \). Therefore, a false index path by the \( f \)-segment has address \( a_0...a_{r-2}a_{r-1} \) and the cumulative false positive for \( f \)-segment, or the false index path, is \( f^{r-1} \). As inductive step, suppose \( f \)-segment \( L' \) with a series of \( B' \)'s in size \( l' > 1 \) have cumulative false positive \( f^{r-1} \) and make address bits \( a_{r-1-r}...a_{r} \). If its parent \( B^{r-1} \) becomes a false positive about address bit \( a_{r-1} \), then new series including \( B^{r-1} \) in front becomes \( f \)-segment with size \( l' + 1 \) and the cumulative false positive does \( f^{r-1} \) \( f^{r-1} \) because a false-positive of \( B^{r-1} \) is independent to \( f \)-segment \( L' \). Now new \( f \)-segment \( L' \) has new false index path \( a_0...a_{r-1}a_{r-1-r}...a_{r} \) where \( a_i \in \{0, 1\} \), \( r' - l' \leq t \leq r - 1 \).

B. Proof of Theorem 2

**Proof:** By induction method, as basis step suppose \( l \) of binary tree \( T \) be 2. Then cumulative false positive by