A Space- and Time-Efficient Hash Table Hierarchically Indexed by Bloom Filters

Heeyeol Yu and Rabi Mahapatra
Texas A&M University
College Station, TX 77843
Email: {hyyu,rabi}@cs.tamu.edu

Abstract

Hash tables (HTs) are poorly designed for multiple memory accesses during IP lookup and this design flow critically affects their throughput in high-speed routers. Thus, a high capacity HT with a predictable lookup throughput is desirable. A recently proposed fast HT (FHT) [20] has drawbacks like low on-chip memory utilization for a high-speed router and substantial memory overheads due to off-chip duplicate keys and pointers. Similarly, a Bloomier filter-based HT (BFHT) [13], generating an index to a key table, suffers from setup failures and static membership testing for keys. In this paper, we propose a novel hash architecture which addresses these issues by using pipelined Bloom filters. The proposed scheme, a hierarchically indexed HT (HIHT), generates indexes to a key table for the given key, so that the on-chip memory size is reduced and the overhead of pointers in a linked list is removed. Secondly, an HIHT demonstrates approximately 5.1 and 2.3 times improvement in on-chip space efficiency with at most one off-chip memory access, compared to an FHT and a BFHT, respectively. In addition to our analyses on access time and memory space, our simulation for IP lookup with 6 BGP tables shows that an HIHT exhibits 4.5 and 2.0 times on-chip memory efficiencies for 160Gbps router than an FHT and a BFHT, respectively.

1. Introduction

The rapid increase in high-bandwidth usage applications from a huge number of hosts has resulted in a substantial demand for high-speed and large-scale routers. It is observed that a class of fast packet processing, such as packet classification and IP lookup, has become critical data path functions for these networking applications. These functions have enjoyed wide application in networking devices to support firewall, access control list, and quality of service in several network domains. They look up a packet in a table that fits its associated rule under various matching conditions, such as singleton, longest prefix match, or range match. One interesting lookup scheme uses Ternary Content Addressable Memory (TCAM) to achieve deterministic and high-speed packet processing [14, 22, 24].

Unlike TCAM with high cost and power consumption, approaches using a Bloom filter (BF) have been widely documented in literature on networking [8–11, 13, 18, 20]. A BF is a virtually generalized hash mechanism for an approximate membership testing with memory-efficient set representation. Dharmapurikar et al. in [9] introduced the first algorithm to employ BFs working in parallel for IP lookup. However, any packet processing that simply uses BFs like [9, 11, 18] can only provide an approximate match, so that it will suffer time loss from the later sequential perfect match.

Traditionally for a fast perfect match search, a hash table (HT) is widely used to perform fast associative lookups since the search requires \(O(1)\) average memory access per lookup. A typical HT application is a network packet processing in a high-speed network router device [13, 19, 20, 23]. In this domain, it is desirable to reduce memory cost and to improve lookup speed by using a small amount of fast on-chip memory that takes 1-5ns and cheap off-chip memory that takes 10ns. Beyond a need of having an \(O(1)\) complexity for IP lookup with on/off-chip memories, a collision rate for a given lookup must also satisfy the bandwidth requirement of a high-speed router. For example, a 160Gbps router requires a deterministic process of 500M packets of minimum size 40 bytes per second. Thus, on average, a chance of collision among 500M lookups does not satisfy the bandwidth requirement of the 160Gbps router. Literature [9, 13, 20] on a BF has focused on the deterministic lookup with a very low collision rate, so that for a given lookup a few accesses to off-chip memory are made.

These approaches [13, 20], however, have the following defects not suitable for a high-speed and large-scale router: 1) Song et al. [20] claimed that for a perfect match a fast HT (FHT) with help of a BF improves the performance over a legacy HT (LHT) by combining hashed linked lists with
A Bloomier filter-based HT (BFHT) [13] utilizes a Bloomier filter [6] with ability of per-key information lookup to provide collision-free IP lookup. Although a BFHT contributes to prefix collapsing as well, it also inherits two disadvantages of a Bloomier filter: a) there is a setup failure in saving \( n \) keys’ per-key informations in an HT, thus another lookup mechanism is required for the failed keys in this setup. b) The setup complexity of \( n \) keys is \( O(n \log n) \), implying that a copy of a BFHT works to update a new key in the rear while lookups for other keys in the BFHT is performed seamlessly.

We have seen the drawbacks of two hash-based schemes: duplicate copies of keys and pointer overhead in an FHT, and a setup failure and \( O(n \log n) \) complexity for update in a BFHT. To cope up with these drawbacks, we propose a basic fundamental hash architecture of novelty to benefit many packet processing applications that require a deterministic lookup with asymptotically tight bound to \( \Theta(1) \). Although a legacy BF is used for the approximate testing on keys’ membership, BFHTs in our scheme, a hierarchically indexed HT (HIHT), are used for approximate testing on index paths for keys in tree forms. Once BFHTs of the last pipelining stage complete their index addresses to entries in an on-chip key table, a perfect match is made by comparing saved keys in the indexed entries with a given key, so that at most one access to off-chip is made to know an associated rule with the given key. This paper has the following contributions:

- An HIHT scheme provides per-key information lookup so that the information per key is used as an index to an on-chip memory key table without pointer operation.

- New algorithms on \( \text{insert} \), \( \text{query} \), and \( \text{delete} \) operations of \( \Theta(1) \) complexity are proposed for the HIHT to reduce memory space and access time complexities.

- It has been shown that the HIHT performs significantly better in terms of space and time, compared to other contemporary techniques, such as an LHT, an FHT, and a BFHT.

- We simulate our HIHT for IP lookup with 6 BGP tables and show that our scheme has 4.5 and 2.0 times memory efficiencies in 160Gbps routers, compared to a corrected FHT and a BFHT, respectively.

Related works of various BFHTs and their applications to networking and IP lookup are mentioned in Sec. 2. In Sec. 3, the basic notion of a BF is presented and Sec. 4 introduces an HIHT for packet processing. More specifically, three operations of \( \text{insert} \), \( \text{delete} \), and \( \text{query} \) involved in an HIHT is described in Sec. 4. Also, an HIHT pipelining scheme for generating an index address as per-key information is illustrated in the same section. A false access to an on-chip key table incurred by a series of BFHTs’ false positives is analyzed in Sec. 5 as the memory efficiency ratios of an HIHT against both an FHT and a BFHT is. After an HIHT is applied to packet processing for IP lookup with 6 BGP tables in Sec. 6, the conclusion for this paper is made in Sec. 7.

2. Related Works

Since the burgeoning interest in a BF’s memory efficiency in late 1990’s, several types of BFHTs have been suggested in various application domains despite design disadvantages as a false positive and the lack of capability of deletion operation. Besides our main comparative literature [9, 13, 20] and other packet processing [1, 8, 11, 18], literature on several applications using BFHTs is discussed in this section.

2.1. Variants of BFHTs

A legacy BF is in an \( m \)-bit vector form and the BF does not support the deletion operation because a bit indexed by a hash function for a key can be overlapped by keys other than this key. To avoid this problem, Fan et al. [12] introduced the idea of a counting BF (CBF) in which each entry in a BF is not a single bit, but rather a small counter of 4 bits. However, our HIHT does not adopt this convention of using counters, causing 4 times larger memory size. Bonomi et al. [3] introduced Approximate Concurrent State Machines (ACSM). While it is similar in spirit to BFHTs, this 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. Although it is known that ACSM data structure takes much less space than a comparable counting BF, the fundamental problem in its approach is that for an \( f \)-positive there is no way to verify an ACSM result. In contrast, our HIHT provides a perfect match mechanism without pointers. Unlike the previous BF approaches in approximate membership testing, for the first time, Bloomier filter in [6] provides both the storage and retrieval of arbitrary per-key information. It guarantees a perfect-hashing with a constant lookup time in the worst case. However, its disadvantage lies in limitation on providing only the static support of membership while our HIHT provides the same arbitrary per-key information as well as a dynamic membership by rotating two trees used for the per-key information. Also, another BFHT disadvantage
is the setup failure probability depending on \( k \), the number of hash functions.

### 2.2. BF’s applications in networking

In an application of overlay networks, continuous reconfiguration of virtual topology by overlay management strives to establish paths with the most desirable end-to-end characteristics. The approximate reconciliation tree for overlay networks by Byers et al. [5] uses BFs on top of a tree structure to minimize the amount of data transmitted for verification. In the packet processing domain, BFs in [9, 18] were widely adapted for fast query with the known collision rate setback. Although a set of BFs designated for each prefix length returns a matched vector, the perfect match is verified in sequential order in off-chip memory until the perfect match is found or when the vector is exhausted, because BF provides an approximate match. However, our HIHT supplies a relational function between a key and its assigned index to an on-chip key table to achieve the perfect match.

For the perfect match in packet processing, Song et al. [20] claimed that an FHT with help of a BF improves the performance over an LHT by reducing the number of off-chip memory accesses. Although chaining in a linked list for resolving a collision is one solution, accessing a key in a linked list costs the same sequential memory accesses as the number of keys in the linked list because of pointer operations. Beyond the generic linked list implementation limitation, combining \( k \) linked lists in a FHT suffers from other limitations described here. First of all, due to merging \( k \) linked lists there exists a chance that duplicate keys are saved in off-chip memory with the depending factor on \( k \). In that \( k \) is reversely proportional to the collision rate, a need of a very low collision rate for a high-speed of 160Gbps makes a number of copies of a key that are proportional to \( k \). Although searching for a key is expedited by choosing the shortest linked list, the insert and delete operations take approximately \( k \) times. These operations are not suitable for a dynamically changing set because any change in the set by one key needs \( 2k \) times more off-chip memory accesses. Besides time complexities of insert and delete operations, to obtain better performance over an LHT in terms of a reduced collision, an FHT needs a plethora of starting pointers for off-chip linked lists and it holds a large wasted portion of on-chip buckets (or pointers). Also, since the perfect match is made in off-chip memory, every query needs at least one access to off-chip memory. Finally, due to the inherent drawback of a BF, the delete operation was designed with a counter of 4 bits in each BF element. Yet, an FHT does not consider the memory size of the counters, but just the number of pointers. However, we make a fair memory size comparison of an HIHT over an FHT. Also an HIHT does not suffer from pointer overhead and duplicate off-chip keys.

Although an FHT is suggested for packet processing’s general purposes, a BFHT [13] is more suitable for IP lookup. It utilizes Bloomier filter [6], which is capable of per-key information lookup to provide a collision-free IP lookup. This per-key information by Bloomier filter is considered as an indexing table for a given packet, so that a BFHT can perform perfect match to make a deterministic lookup. Although a BFHT contributes to prefix collapsing as well, it inherits two disadvantages of a Bloomier filter: first, there is a setup failure in saving \( n \) keys’ per-key information in an HT, so that another lookup mechanism is used for the failed keys in the setup. As a result the number of hash functions gets increased to reduce the setup failure rate and it leads to more memory need. Secondly, the setup complexity of \( n \) keys is \( O(n \log n) \), implying that a copy of a BFHT works for updating a new key in the rear as it performs seamless lookups for other keys. However, our HIHT does not suffer from two aforementioned disadvantages.

### 3. Basics of a Bloom Filter

To figure out the fundamental relationship among memory size, capacity of keys, and the number of hash functions, we present the mathematics on a BF and its false positive, or \( f \)-positive. Then, we introduce the mechanism of insert, query, and delete operations for an HIHT in Sec. 4.

A BF for representing set \( S = \{e_0, e_1, ..., e_n\} \) of \( n \) keys is described as an array of \( m \) bits, initially all set to 0. A BF uses a set \( H \) of \( k \) independent hash functions \( h_0, h_1, ..., h_{k-1} \) with the range of \([0:m-1]\), implying that a hardware of a memory module for a BF needs to support \( k \) randomly-addressed memory reads in parallel for a deterministic lookup. For mathematical convenience, we make a natural assumption that these hash functions map each key in the universe to a random number uniform over the range. For each key \( e \in S \), the bits indexed by \( h_x(e) \) are encoded to 1 for \( 0 \leq x \leq k-1, 0 \leq j \leq m-1 \). A bit in the array can be set to 1 multiple times, but only the first change has an effect. To query that key \( y \) is \( S \), we check whether all bits, indicated by \( h_y(e) \), are set to 1 in the BF. If so, the BF returns ‘yes’ about the query. If not, then evidently \( y \) is not a member of \( S \). Even if all of indexes by \( h_y(e) \) are set to 1, there exists the probability that key \( y \) does not belong to set \( S \) due to random gathering of \( k \) bits set to 1 for independent \( e \). Hence, a BF may yield an \( f \)-positive suggesting that \( y \) is in \( S \) when in fact it is not.

The probability of \( f \)-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 \( h_x \) is \( 1/m \). After all \( n \) keys of \( S \) are hashed \( k \) times into a BF, i.e., totaling \( kn \) times, the probability that a specific bit is still 0 is asymptotically \( p = (1-1/m)^{kn} \approx e^{-kn/m} \). Then, the probability of an \( f \)-positive by randomly choosing \( k \) bits of 1 in an \( m \)-bit array is bounded.
as follows

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

(1)
given a query, according to the result of Broder and Mitzenmacher [4]. After some algebraic manipulation, they claim that

\[ m \geq (n(\log_2(1/e))) / \ln 2 \approx 1.44n \log_2(1/e) = 1.44n. \]

(2)

Furthermore, in an optimal configuration, \( k \) becomes \( w \) according to the following derivivation:

\[ k = \ln 2 \frac{m}{n} = \ln 2 \left( \frac{n \log_2(1/e) \ln 2}{n} \right) / n = w. \]

(3)

This \( k \), denoting the number of hash functions in a BF, turns into the number of read ports in \( mx1 \)-bit SRAM. The burden of a large number of \( k \) for a high query precision can be solved by dividing the SRAM into smaller-sized SRAMs with fewer number of read ports. This division does not affect the probability of an \( f \)-positive in Eq. (1) as noted in [9]. Although the minimum size of \( m \) for a BF is determined by Eq. (2) to guarantee \( \epsilon \), the BF intrinsically does not support perfect match lookup, unlike an FHT and a BFHT. To provide perfect match beyond approximate match by a BF, we build a hierarchical index path with a group of BFs to index an entry in a key table and the detailed implementation is in the following section.

4. A Hierarchically Indexed Hash Table

Unlike using a BF in an FHT, with a set of BFs we conceptually embed a hierarchical indexing tree (HIT) into an HT of less memory size than an FHT. That is, the memory area for an HT used for indexes to a key table is hierarchically partitioned to make indexes. An HIT for \( n \) keys in power of 2 is composed of \( \log_2 n \) layers (i.e. SRAM modules) and partitions the address space in a rectangle of \( n \times s \) 0/1 bits.

This means that a BF covers a column group of the same bits, either 0 or 1, in the address space to a key table. The build details of an HIT is as follows.

4.1. Building a conceptual on-chip HIT

As shown in Fig. 1(a) for HIT build, let \( B_i \) denote \( j \)-th BF in layer \( i \), hereinafter \( 0 \leq i \leq s-1 \), and let all \( n \) keys be filled in a on-chip key table sequentially from index \( 0_{i0}, ... 0_{i1} \) to index \( 1_{i0}, ... 1_{is} \). If key \( e \in S \) is to be inserted at index address \( A = a_0a_1...a_{s-1} \), where \( a_i \in [0, 1] \), \( 0 \leq i \leq s-1 \), a BF, denoted \( B_m = a_m \) at each layer \( i \), is involved to encode key \( e \) as a legacy BF does. In this hierarchical partitioning and encoding, \( B_i \) covers \( n_i = 2^s \) keys whose indexes in a key table range from \( j \cdot 2^{i-1} \) to \( (j+1) \cdot 2^{i-1} - 1 \). For instance, \( B_0, B_1, \) and \( B_2 \) take care of sets \( \{e_0, e_1, e_2\}, \{e_4, e_5\} \) and \( \{e_7\} \), respectively. Eq. (2) states that \( m \) is linearly proportional to \( n \) for a given \( f \). Thus, given \( f=2^{-m} \) for a BF on layer \( i \), where \( w_i \) is a query precision of a BF on layer \( i \), the total memory \( M_m \) for BFs on layer \( i \) is of size \( 2^{m-i}(1.44n)w_i < 1 \). Finally, an embedded HIT is comprised of an 0-tree and 1-tree covering half of \( n \) keys located in \( 0_{i0}, ... 0_{is} \) of a key table and the remaining half in \( 1_{i0}, ... 1_{is} \), where \( x, y \in [0, 1], 1 \leq i \leq s-1 \).

4.2. Insert operation in an HIT

Fig. 1(a) shows a basic structure of our HIT consisting of 3 layers of BFs for 8 keys. The left side of Fig. 1(a) shows the binary address space with a set of BFs partitioning the address space of a key table, and the right side shows the transformed dual trees, 0-tree and 1-tree, where each node represents \( B_i \). For example, the insertion of key \( e_6 \) at index 100; means \( B_0 \) at layer 0, \( B_1 \) at layer 1, and \( B_2 \) at layer 2 are involved.

Procedure insert shows that a detailed insert operation is as simple as that for a BF. Albeit conceptually all BFs are
Procedure insert

Input: key $e$, rule $r$, and address $A=a_0 \cdots a_{s-1}$ in binary bits
Output: Encoded HIT about $e$

for layer $i = 0$ to $s-1$ do /* On-chip Op. */
    $m_i = 4A_i; w_i = j=a_0 \cdots a_i;$
    $idx = j \cdot m_i; /*$ Find right base index for $B_j'/*$
    for $r=0$ to $k-1$ do /* $M_{\text{now}}$ for BFs on layer $i/*/ 
        $M_{\text{now}}[idx+h_i(e)] = 1;$
end

$M_{\text{off}}[A] = r; M_{\text{off}}[A] = e;$

4.3. Delete operation in dual HITs

Delete operation is not as easy as the insert operation because a basic BF does not support deletion of a key which was inserted in the BF. However, we couple HITs, an l-HIT and a r-HIT, in on-chip memory to rotate a target HIT for insert operation and another target HIT for delete operation, as illustrated in Fig. 1(b). Once one HIT is full of previous $n$ keys, query operation stays with the HIT. But if set $S$ is dynamic and limited in size $n$, a new HIT takes care of a new key insertion by setting BFs in a new HIT as well as a bit in a valid bit array (VBA). An index for the new key is indicated by ‘next’ that is updated every time from a free address stack (FAS) in off-chip memory. Also, the old HIT handles delete operation simply by setting off a bit in a VBA that is coupled with the corresponding HIT. Updating ‘next’ and an FAS is not a burden task because whenever there are insert or delete operations, these operations need off-chip access, thus, ‘next’ and an FAS can be updated without additional cost. Checking $V_1$ and $V_2$ with indexes given by two HITs guarantees that an unnecessary off-chip access is blocked within on-chip for HITs. Also, when all $n$ keys are encoded in one HIT, i.e. the moment when a FAS is empty, the other HIT needs to be initialized to 0 for the next set of insert operations with initialized BFs.

Suppose, for example, 8 keys, $e'_0$, ... , and $e'_7$, are inserted in an l-HIT as shown in Fig. 1(b). Then, after the 8 keys, a target HIT for new insertion becomes a r-HIT. Now, the ensuing operations are deletions of $e'_2$, $e'_1$, and $e'_6$. After the deletions, suppose the next operation is insertion of $e'_5$ in a r-HIT. Then, after a bit for $e'_5$ in array $V_r$ is properly set as shown in Fig. 1(b), the key $e'_5$ is saved in an entry of index $6$ in a key table. Also, next and an FAS have 001 and 100, respectively. By rotating a target HIT for insertion among dual HITs and confirming an index returned by each HIT with a VBA, the operations of insert and delete are processed seamlessly. Also, by using two rotated HITs, we do not need counters in each BF, i.e. a CBF, which costs 4 times more memory size than a BF. Thus, we have proven using two HITs without CBFs eliminates 2 times the memory consumption. The detailed procedure and complexity of delete are provided in Sec. 4.4.

4.4. Query operation making indexes in HITs

Once all keys are saved in a key table and encoded in a set of on-chip BFs, the remaining and ultimate goal of an HIT is to search a key by query operation fast. There are two kinds of search patterns, an unsuccessful search (US) in which a key is relentlessly searched although it is not in an HIT, and a successful but time-consuming search (SS) in which a key is to be searched via an HIT. Before we discuss these two kinds of searches in details, let us introduce definitions of an index path, a false index path and a false segment.

Definition 1 (Index Path)
In an HIT, an index path, or $i$-path, for a key is defined as a series of $B_j's$ used in insert operation of the key and is hierarchically connected with each other from layer 0 to layer $s-1$, by making a sequence of index bits. The sequence of indexing bits in $B_j's$ is also matched with an arbitrary index for the key that is saved in a key table and the size of the sequence of bits from the series of $B_j's$ must be $s$.

As a corollary, we can conclude that in query for key $e$, previously encoded by insert for this key, an $i$-path for the key $e$ should show up as BFs return ‘yes’ for the true membership.

In an HIT, besides an $i$-path dedicated to a key, a false index to a key table is potentially possible due to $f$-positives generated from irrelevant BFs. For example, suppose key $e'_4$ is inserted with $i$-path $1_0; 0_0$ in Fig. 1(a) and then a query to $e'_4$ is requested. The result of the query may give an ambiguous $1_0; 2_0$, $x_6 \in \{0, 1\}$, due to an $f$-positive of $B_j$. Thus, this ambiguity needs two accesses to a key table. Given a query
for an i-path of size \( s \), there are totally \( 2^s - 1 \) false indexes because each \( B^i \) is independent and identically distributed, i.i.d. Besides the definition of an i-path, we define false index path in query operation which leads to a false indexing to a key table.

**Definition 2 (False Index Path and False Segment)**

In query, from hierarchically consecutive layers, a group of \( B^i \)'s not pertaining to an i-path can be formed in a series of at most size \( s \), and to become a false index path, or f-path, this series needs to be either connected to an i-path or a completely different path of size \( s \), i.e. independent of an i-path in an HIT. Also, we call the group attached to an i-path a different path of size \( s \), i.e. independent of an i-path. Figure 2 shows an example of the probability calculation of an f-path.

Even if it is possible that there is a set of BFs with f-positives in query, BFs that are only hierarchically connected to each other and an i-path can be a part of an f-segment. Thus, f-positives from the rest BFs can be ignored. For example, given a query of the previous \( e_4 \) at index 100; in Fig. 1(a), even if \( B_3^1 \) and \( B_3^1 \) randomly make f-positives right after query, there is no f-segment starting from the \( B_3^1 \) and \( B_3^1 \). By the definition of an f-path, the probability of the f-path is cumulatively calculated as the product of f-positives from BFs along the f-path.

**Figure 2. Examples of an i-path, f-segments, and f-paths.**

Fig. 2 shows an example of the probability calculation of an f-path in an HIT with one i-path and three f-paths. A series of \( a_0a_1a_1a_1 \) in the dark boxes is an i-path. The probability of the f-segment \( b_3b_1 \) forming f-path \( a_0a_1b_3b_1 \) is \( \prod_{i=2}^{t} f_i \) where \( f_i \) is the f-positive of a BF on layer \( t \). Also, the probabilities of the remaining f-segments, \( c_0c_1c_2c_3 \) and \( c_0c_1c_2d_1 \), are the same as \( \prod_{i=2}^{t} f_i \) because the probabilities of f-positives of BFs on the same layer are the same with each other.

Once the probability of an individual f-path is known, we need to pay final attention to the probability that an HIT may have \( t \) f-paths, \( 0 < t < n \). Suppose a binary tree \( T \) of height \( l \) has sub-trees \( T_l \) and \( T_r \) of height \( l-1 \) which have \( n_l \) and \( n_r \) f-segments of size \( l-1 \). Also, let \( T_l \) and \( T_r \) have probabilities \( F_l \) and \( F_r \) for their f-segments. To have \( n_l + n_r \) f-segments of size \( l \), the binary tree \( T \) with height \( l \) needs to be an f-positive. Thus, the probability \( F_{n_l} \) of the binary tree \( T \) with its sub-trees having \( n_l + n_r \) f-segments is the product of three: the probability that \( T \) needs to be an f-positive, the probability that \( T_l \) has \( n_l \) f-segments, and the probability that \( T_r \) has \( n_r \) f-segments, i.e. \( f_l \cdot F_l \cdot F_r \). Based on this recursive way, the probability \( P_{P}(t) \) of \( t \) f-segments starting on layer \( i \) in an HIT of height \( s \) is calculated as the following:

\[
P_{P}(t) = \sum_{v=0}^{t} P(v) \cdot P_{T}(t-v) \cdot f_i \quad \text{if} \quad t \leq 2^{s+1},
\]

where base cases of \( t > 2^{s+1} \) and \( i = s \) are 0 and 1, respectively.

**False indexing to an on-chip key table for a US:** Besides the design issue of low probability of more than one key table access for an SS, it is equally important to note that the probability of f-indexes in a US is also lower. Unlike an SS, in a US there is no i-path for a given key, meaning that all BFs in query return ‘yes’ as f-positives. However, there is a chance that each of 0-tree and 1-tree may give plural f-paths. In contrast to an f-positive in an FHT leading to off-chip memory accesses, an f-path composed of a series of f-positives of hierarchically connected \( B^i \)'s in each layer \( i \) becomes one index access to a key table. Thus, we expect a far less probability due to the product of f-positive probabilities of BFs.

Suppose random variable \( X_v \) is the number of f-paths in a US on an HIHT. Then, the probability \( Pr[X_v = v], v > 0 \) can be easily derived based on Eq. (4) as the following

\[
Pr[X_v = v] = \sum_{r=0,t1,t2} P(t0) \cdot P(t1) \cdot P(t2) \cdot P(t3)
\]

because there are two HITs, \( l-HIT \) and \( r-HIT \), each having 0- and 1-trees. The sum in Eq. (5) accounts for the combination of deriving \( v \) among \( t0, t1, t2, \) and \( t3 \). That is, if \( v=1 \), there are four cases: \( 0+0+1+0, 0+1+0+0, 1+0+0+0, \) and \( 1+0+0+0 \).

Although we can choose a larger value for precision \( w_i \) on layer \( i \), the number of BFs on layer \( i \) in an HIT must be upper bounded, so that the total number of memory reads to \( M^u_{w_i} \) for the BFs must be sustainable in hardware implementation. The expected number of BFs to probe on layer \( i \) for a US becomes

\[
N_{w_i}^{u} = 2^{[2^i]} \cdot \left[ \sum_{i=0}^{t-1} f_i \times 2 \right] \times 2 \times [2^{i-1} \times 2^{-i} \times 2] = 2^{i-1},
\]

where \( f_i = 2^{w_i} = 2^{-2} \) except layer \( s \). On layer \( s \), \( f_s = 2^{w_s} \) is set to a collision rate as low as one for a high-speed router like \( 2^{-3} \) for 160Gbps. In convergence, \( \lim_{w \to \infty} N_{w_i}^{u} = 0 \), this means that as long as \( i \) for a layer (or SRAM module) index increases, the expected number of BFs on the layer \( i \) is minuscule enough that a simple memory hardware can support a small number of memory read requests.

**False indexing to an on-chip key table for an SS:** We
have derived the probability of \( f \)-paths in a US. Now, we want to calculate the probability of the number of \( f \)-paths in an SS. The situation in an SS is very different from that of a US because there must be one \( i \)-path and several possible \( f \)-paths in an SS while there is no \( i \)-path in a US. Fig. 3(a) shows an example of 5 layers for 25 keys and along an \( i \)-path, but results in saving these layers’ memory. Besides decision rule of \( wi \) for layer \( i \), we need to calculate the number of partially found \( f \)-segments on each SS layer for hardware implementation. Two times of this number is considered as the number that BFs needed to probe on the next layer, like the expected number of BFs in Eq. (6) for a US. Note that in an SS, there is an \( i \)-path and at least two children BFs probe from a BF on the \( i \)-path due to a binary property. The expected number of BFs to probe on layer \( i>0 \), except layer \( s-1 \), for an SS becomes

\[
N_{fs} = 2 + f_{i-1} \cdot 2 + 2 \cdot f_{i-1} \cdot f_{i-2} \cdot 2^2 + \cdots + 2^{s-1} \cdot f_{i-1} \cdots f_0 \cdot 2^i
\]

where 2 accounts for two BFs on an \( i \)-path and \( f_i = 2^{-2} \). With Eq. (6), the maximum in convergence among the expected numbers of BFs to probe in an SS and a US becomes 3 as \( \lim_{s \to \infty} \max\{N_{fs}, N_{us}\} = 3 \). Thus, the total number of memory reads to \( M_{us} \) for layer \( i \), \( 0 \leq i < s-1 \), is \( k_i \times 3 \) because for each BF \( k_i \) hash functions are necessary. In conclusion, in a query, either unsuccessful or successful, the expected number of memory reads on \( M_{us} \) for layer \( i \), \( 0 \leq i < s-1 \), is upper bounded to \( 3k_i \), where \( k_i \) is set 2. On layer \( s-1 \), \( k_{s-1} = w_{s-1} \) is set to the log of reverse collision rate, i.e., 29 for 160Gbps, so that the bandwidth requirement is secured by our deterministic \( \Theta(1) \) lookup. The \( k_{s-1} \) memory reads of random locations by \( k_{s-1} \) hash functions can be supported by a simple switching circuitry in a cycle. Even if a commodity of SRAM has a limitation in the number of supported memory reads, a large-sized SRAM can be divided into a smaller SRAM with a fewer number of memory reads without worsening an \( f \)-positive as suggested in [20].

Based on the observation from Fig. 3(b), we need to calculate the number of \( f \)-paths for false indexes to an on-chip key table in an SS. Let random variable \( X_i \) be the number of false paths from an HIHT, given an \( i \)-path for a key in query operation. Then, \( X_i+1 \) is the total number of SS indexes is equal to the searched-linked-list length in an FHIT [20]. The detailed probability of \( X_i \) for an SS without \( f \)-paths is defined as following

\[
Pr[X_i = 0] = P_0(0) \cdot P_1(0) \cdot P_2(0) \cdots P_{i-1}(0),
\]

(8)

because each \( d \)-tree along an \( i \)-path is independent to each another. In general, the \( Pr[X_i = v] \) is calculated based on the independent property of each \( d \)-tree along an \( i \)-path as the
Note that following (9) and (5) as that 2 average memory access on the condition of an existing target noted as the number of accesses to an on-chip key table, the Fig. 3(b). 

The detailed procedure is shown in Procedure delete. Note that query operations in line 1 take place in on-chip.

The complexity of delete in on-chip memory is \( \Theta(1) \), which is calculated based on the fact that the query complexity is \( \Theta(1) \). The complexity of indexes to access a key table is \( \Theta(E[Z]) \) on average for a successful deletion and it is to be constant as \( E[X_i] \) becomes \( \Theta(1) \).

5. Analysis of Time and Space among HTs

In this section, we present an analysis on an average access time and memory efficiency for four schemes: an LHT, an FHT, a BFHT, and an HIHT. Also, we later provide a brief overview on the complexities of on-chip and off-chip memory for these four schemes. As to a perfect hash function, Mitzenmacher and Vadhan [17] claimed that simple hash function can provide a truly random hash function. A class of universal hash functions are suitable for hardware implementation and thus we chose a scheme from [16].

Table 1 summarizes the complexities of operations in line 1 take place in on-chip.

The detailed procedure is shown in Procedure delete.
Table 2 lists the difference of four schemes in their complexities for on-chip memory accesses. A BFHT suffers from harder complexities of insert and delete due to the lack of dynamic membership change than the rest, including an HIHT.

5.1. Average access time

Let us define the average access time to be the number of accesses to a key table for a given query. For an LHT and an FHT, the average access time is the number of keys accessed in an off-chip linked list while it is the number of an i-path and f-paths calculated from query-i for the perfect match in an HIHT. Note that the number of f-paths plus an i-path is comparable with the length of a chosen linked list in an LHT and an FHT. Also, regardless of an SS and a US, the final result of indexes from query-i on layer s-1 is the only candidate set of indexes used to find a key in a key table based on Definitions 1 and 2.

Eq. (10) is used to get an HIHT average access time as the following

\[ T = p_s E[X_s] + (1 - p_s) E[X_n], \]  

(12)

where \( p_s \) denotes the frequency of successful searches for keys. The average access time of other schemes can be calculated as easily as Eq. (12) of an HIHT. Based on [7] and [20], those of an LHT and an FHT are related with the load factor defined as the number of keys over the number of buckets (or starting pointers). Note that an FHT does not consider the number of buckets in bits. For a BFHT, the per-key information is saved as an additional memory overhead and the probability of a false query in a US is determined by \( k \) and \( w \), defined as an entry’s bit width in an index table.

Fig. 4 shows the probabilities of the number of indexes (or accesses) in an SS and a US, and the average access times of Eq. (12) and others schemes under the same query precision requirement, \( w \). Note that we put a corrected FTH, marked as c-FHT considering counters and the existing FHT, marked as FHT not considering counters as memory. Also, note that a limited number of indexes can be resolved in one HIHT cycle, while \( t \) access times in a linked list need exactly \( t \) cycles in an LHT and an FHT; this is due to sequential access along the linked list. Fig. 4 (a) shows that \( Pr[X_s] \) of an HIHT is always less than those of an LHT and an FHT. In Fig. 4 (b), the probability \( Pr[X_n] \) of an HIHT is significantly less than that of an FHT. Especially, the result in Fig. 4 (c) indicates that the lower the successful search rate, the better our HIHT performance is than those of an LHT, an FHT, and a BFHT.

Although we set the collision rate (i.e. a rate of \( j \)-positive for an FHT and a \( j \)-path for an HIHT) to \( 2^{-10} \), our HIHT has the lowest access time among all comparative schemes, implying that our HIHT offers the most time-efficient hashing scheme.

5.2. On-chip memory usage

To maximize memory efficiency of an HIHT, we set the precision of layer \( t \) to 2, \( \log_2 t+1 \) and that of layer \( s-1 \) to \( w \) as the same as precisions of an FHT and a BFHT. The precision value 2 is chosen based on the hardware consideration for pipelining and memory read ports as stated in Sec. 4.4. Also, for a fair comparison, the on-chip memory tables were considered in a BFTH and an HIHT. According to Eq. (2) for the requirement of \( f=2^{n} \), the total HIHT memory usage, \( M_s \), counts two copies of HITs plus two VBAs and a key table and is calculated as the following

\[ M_s = 2(2^f \beta_n \omega_0 + \cdots + 2^f \beta_n \omega_{s-1} \omega_{s-1} + n) + n \log_2 n \]  

(13)

where \( \beta = 1.44 \). In contrast, the FHT memory usage is \( M_f = \beta n \omega n + 4 \beta n \omega n = \beta n \omega (s+4) \) by considering 4-bit counters while an FHT in [20] did not consider the counters for a fair memory comparison. The memory size \( M_b \) for a BFHT is 2 times \( k \omega (n \log_2 n + \log_2 k + w) \) due to the seamless update in \( O(n \log n) \) complexity. Thus, considering \( n \log_2 n \) for an on-chip key table, memory efficiency ratios \( R_{HF} \) and \( R_{HB} \) of \( M_s \) over \( M_f \) and \( M_b \) become

\[ R_{HF} = \frac{\beta n \omega (s + 4)}{2^f n \sum_{i=0}^{s-1} \frac{w_i + 2n \log_2 n}{2^n} + n \log_2 n} \]  

(14)

\[ R_{HB} = \frac{2 \kappa n \log_2 n + \log_2 k + w + n \log_2 n}{2^f n \sum_{i=0}^{s-1} \frac{w_i + 2n \log_2 n}{2^n} + n \log_2 n}. \]  

(15)

Fig. 5(a) shows two ratios, \( R_{HF} \) and \( R_{HB} \), calculated from Eqs. (15) and (14) in the range [10:30] for \( w \) with the required precision of each scheme and in the range [10:30] for \( s \). Note that the LHT memory size against \( M_h \) is not considered nor drawn in the figure, because with a given high precision \( w \) of 29, the LHT memory size is more than a hundred times larger than that of an HIHT. The change in \( w \) provides a greater benefit in \( R_{HF} \) than the change in \( s \) does, implying that as long as the collision rate stays lower for a high bandwidth, our HIHT maintains multi-integer-fold, not fractional-fold, efficiency gain. For instance, 160Gbps needs to set \( w \) to be at least 29 (= \( \log_2 500 \)) M. However, \( R_{HB} \) shows the minor change according to \( w \) and \( s \), and it gives around two times memory efficiency over all ranges. These results on \( R_{HF} \) and \( R_{HB} \) support our claim that our HIT offers a better space-efficient as well as a better time-efficient hashing architecture shown in the previous section.

5.3. Off-chip memory usage

It is very challenging to figure out the characteristics of the number of duplicated keys in a real hash function implementation as well as the keys dynamic order of insert and delete operations. However, although Song [20] showed that
there exists up to 3 times in memory overhead for duplicates in \( k=10 \) scenario, our experiments with various values of \( k \) shows that the number of duplicated keys depends on \( k \) to decide a precision of query for low collision rate. As shown in Fig. 5(b), with synthetic keys on various \( k \) in a hundred runs, we derive that the average number of saved duplicate keys is proportional to \( k \), which now is proportional to \( \log_2(1/f) \) according to Eqs. (1) and (2). This means that for a given high precision search requirement for a high-speed router, the smaller value of \( f \) is, the larger \( \log_2(1/f) \) is, and the larger \( k \) leads to more duplicate keys on off-chip memory. Note that the numbers of saved keys in a BFHT and an HIHT are the same number as an LHT as shown in the three bottom lines of the figure.

6. Simulation for IP Lookup

As an application of an HIHT, we calculate the on/off-chip memory size for IP lookup in this section. While other packet processing, like IP lookup and packet classification, need prefix and range match, hash-based approaches via an FHT and a Bloomier filter support singleton match where a given key lookup is matched with a stored key. However, like a BFHT [13], we provide a deterministic lookup needing at most one off-chip memory access, by splitting an IP address of 32-bit string into \( 32/s' \) substrings for prefixes by stride \( s' \), i.e. prefix collapsing. Fig. 6 shows the detailed hardware configuration with HIHTs. In parallel, each on-chip HIHT covers its own prefix length. As done via prefix collapsing of [13], a priority encoder determines the final HIHT of the longest length among the matching HIHTs with the given an IP packet. Later, the result in an index is added with a bit vector of the index, so the off-chip next-hop table is accessed by the summed index only to test if a query is an SS. Every bit vector with a prefix in on-chip table is calculated by prefix

<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>BFHT</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>HIHT</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>

Table 1. Complexities of operations to off-chip in an LHT, an FHT, a BFHT, and an HIHT.

<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(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>BFHT</td>
<td>( O(n \log n) )</td>
<td>( \Theta(1) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>HIHT</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>

Table 2. Complexities of operations to on-chip in an LHT, an FHT, a BFHT, and an HIHT.

Figure 4. Probabilities of memory access in SS/US and the average access time for an LHT, an FHT, a BFHT, and an HIHT with \( n=10^{210} \). \( k=10 \) for an LHT, an FHT, and an HIHT. \( k=4, w=10 \) for a BFHT.

Figure 6. IP lookup configuration with \( L/s' \) HIHTs in on-chip and tables in off-chip.
collapsing to efficiently support the wildcard bits. A second dotted box in Fig. 6 shows a query of prefix 01101 has an i-path and an f-index returned by HITs, and only an i-path is matched in a key table. This kind of hardware configuration makes a deterministic \( \Theta(1) \) query, unlike an FHT, because the HIHT match decision is made in on-chip.

6.1. On-chip memory size

![Graph showing on-chip memory size with various hash schemes for 6 BGP tables.](image)

Fig. 7 shows the simulation results of hash schemes by NePSim [15] for IP lookup with precision \( w=29 \). 6 BGP tables of IPv4 prefixes are obtained from [2, 21]. The number of prefixes for BGP table AS65000, AS6447, AS1221, AS12654, AS5459, and AS3303 are 233451, 235307, 181295, 170459, 78133, and 68739, respectively. Each BGP table is indexed beginning with the letter 'B'. Fig. 7 shows the on-chip memory size of each BGP table with different collision rates for 160Gbps and 40Gbps routers whose required collision rates are \( 2^{-29} \) and \( 2^{-27} \), respectively. Fig. 7 does not show the size of an LHT in a colossal difference due to the very high collision rate. It is shown that under all schemes, an HIHT has the smallest memory size. In 160Gbps, it has 4.5 and 2.0 times efficiency over a c-FHT and a BFHT, respectively, while in 40Gbps it has 4.2 and 2.0 times. The reason for different efficiencies seen in an c-FHT is that the higher bandwidth requires a lower collision rate, i.e. \( f \).

7. Conclusion and Future Work

We have proposed a novel hash architecture with two HITs generating indexes to a key table with a set of BFs. The BFs in two HITs work systematically, or in pipelining and hierarchically, so that the number of the generated indexes is minimized and in the worst case only one off-chip memory access is guaranteed while the on-chip memory efficiency is achieved. As to insert, an i-path is assigned to a key, and one BF along the i-path on each layer \( i \) is involved in encoding the key in one of HITs. In query, one on-chip memory module for each layer is probed for candidate BFs by having their base memory indexes derived from Eq. (2). After the last probing in layer \( s-1 \), the returned indexes are used for perfect match in an on-chip key table, so that a deterministic \( \Theta(1) \) lookup is guaranteed. For delete, by rotating two HITs, seamless update of keys is provided without counters costing four times the memory, so that a total of two times memory is saved.

In our analysis with rotating two HITs in Sec. 5, we calculated the average numbers of indexes of a BFHT and an HIHT, and compared them with the numbers of off-chip memory accesses of an LHT and an FHT. The generated
HIHT indexes can be resolved in an on-chip key table in parallel, so that at most one off-chip access to a rule table in one cycle is sufficient. However both an LHT and an FHT need the same accesses as the number of keys in a linked list due to sequential accesses via pointers. In HIHT memory efficiency ratios compared against the rest schemes, our HIHT is proven as the most efficient scheme than comparative schemes in high precision $w$ for 160Gbps routers. In terms of off-chip memory, due to duplicate keys, an FHT is affected by precision $w$ while our HIHT is not. As an application to IP lookup, an HIHT used at most 4.5 times less on-chip memory for six BGP tables, compared to the rest schemes.

We have shown many benefits of an HIHT, such as arbitrary per-key information without using a pointer, a deterministic $Θ(1)$ lookup, a dynamic membership change, and on-chip memory efficiency. Although we applied our HIHT to IP lookup, other packet processing, like packet classification, are not considered due to their complicated problem domains. As future work, we will consider this multidimensional packet processing with an HIHT to provide space- and time-efficient lookup.

Acknowledgment

The author would like to thank the reviewers for helpful feedback and Dr. Rabi Mahapatra for significant help with comments.

References

[21] University of Oregon Route Views Project.