A Hash-based Scalable IP lookup using Bloom and Fingerprint Filters†

Heeyeol Yu
Computer Science and Engineering
University of California, Riverside
Email: hyyu@cs.ucr.edu

Rabi Mahapatra
Computer Science and Engineering
Texas A&M University
Email: rabi@cse.tamu.edu

Laxmi Bhuyan
Computer Science and Engineering
University of California, Riverside
Email: bhuyan@cs.ucr.edu

***Abstract*— Several challenges in the IP lookup architecture must be addressed for a high-speed forwarding in a large scale routing table: power, memory, and lookup complexity. Hash-based architectures have lookup schemes that are recognized for being both power and memory efficient due to their $O(1)$ lookup, in contrast to other contemporary architectures. In this paper, we propose a novel hash architecture to address these issues by using pipelined Bloom and fingerprint filters for a binary searching in keys. The proposed hash scheme encodes keys' indexes to an on-chip fingerprint table, approximately returns a few indexes in a key query without pointer overhead, and makes a perfect match in an off-chip key table. Due to a memory banking system in pipeline stages, we can achieve $O(1)$ pipelined throughput complexity of insertion, deletion, and query operations. For the IP lookup, a Lulea bitmap with our hash scheme supports a prefix lookup without inflating the numbers of prefixes and next-hops, so that our scalable hash-based scheme can achieve the worst case $O(1)$ IP lookup. The simulation with large scale routing tables shows that our IP lookup scheme offers 4.5 and 50.1 times memory and power efficiencies than other contemporary hash and TCAM schemes, respectively.**

I. Introduction

The demand for high-speed and large-scale routers continues to surge in networking fields. It has been reported that the traffic of the Internet is doubling every two years by Moore's law of data traffic [1] and the number of hosts is tripling every two years [2]. To date, the number of prefixes in a core router's routing table for is about 290K [3]. These rapid increases in traffic and hosts lead to two major IP lookup related problems in core routers. 1) Speed: a high-speed router needs to look up a routing table at the rate that corresponds to the router's bandwidth requirement. For example, at the rate of 160Gbps, 500M lookup requests must be processed in a second which implies that a packet of minimum 40 bytes must be forwarded to a next hop in 2ns in the worst case. 2) Scalability: a fast IP lookup must be made in searching the longest prefix match even with hundreds of thousands of prefixes.

Since a fast packet forwarding is a router's critical data path, literature on packet forwarding has developed schemes involving three major techniques: Ternary Content Addressable Memory (TCAM), trie-based, and hash-based schemes. Although a TCAM provides a deterministic and high-speed packet lookup [4, 5], due to its non-commodity nature and brute-force search method, its cost and power dissipation tend to become prohibitive for packets with a large number of prefixes and high line rates. Unlike TCAM, trie-based scheme uses a tree-like data structure to successively classify a packet a few bits at a time [6, 7], its drawback, however, lies in its inherent nature to space consumption when it holds pointers from nodes to their children and sequential memory accesses due to these pointers.

In contrast, since hash-based schemes do not perform brute-force lookups like a TCAM does, they can potentially receive an order-of-magnitude power saving. Furthermore, hash tables (HTs) employ a flat data-structure, unlike tries, to achieve potentially smaller memory sizes amenable to on-chip memory.

Traditionally, an HT is popularly used for a fast search due to its $O(1)$ average memory access per lookup under reasonable assumptions. Unlike a TCAM with its high hardware cost and power consumption, and a trie with imbalanced memory access and pointer overhead, hash-based, especially a Bloom filter (BF)-based, approaches have been widely documented in networking literature [8–13]. A BF is essentially a compact set representation for approximate membership testing. Although the BF testing is approximate, HTs and binary search trees which provide exact membership testing on a set are not preferable due to memory overhead like pointers.

For IP lookup, authors [9] introduce the first algorithm to employ BFs to work in *parallel*. Yet, their scheme does not provide a deterministic lookup due to a BF's *approximate* match. Thus, it suffers time loss in another *sequential* perfect match through a hash table in a slow off-chip memory. BF literature [11, 12] focuses on a perfect match lookup running at a very low collision rate, so that given a lookup only a few memory accesses to an off-chip hash table are made.

These recently-proposed schemes [11, 12], however, have the following design flaws not suitable for a high-speed and large-scale router: 1) Song *et al.* [11] claimed that a fast HT (FHT) with the help of a BF improves its perfect match performance over a legacy HT [14]. Their FHT scheme is to combine hashed linked lists with k hash functions to warrant that only the shortest linked list is used in the search. Beyond the generic linked-list implementation limitation, like pointer overhead and sequential accesses along a linked list, an FHT suffers from two drawbacks: 1.a) due to merging k linked lists, there is a possibility that duplicate keys are saved in off-chip memory with a depending factor on k. 1.b) Although a key search is expedited by choosing the shortest linked list, the *insert* and *delete* operations consume approximately k times.

†This research was supported in part by NSF grant CNS 0832108

978-1-4244-4634-6/09/$25.00 ©2009 IEEE

2) A Bloomier filter-based hash table (BFHT) [12] for IP lookup utilizes a Bloomier filter [15], which is capable of per-key information lookup, to ensure a collision-free lookup. In addition to the prefix collapsing that a BFHT contributes to, it also inherits two disadvantages of a Bloomier filter: 2.a) there is a setup failure in saving n keys' per-key information in a Bloomier filter, consequently another lookup mechanism is needed for the failed keys in the setup. 2.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 of other keys are performed seamlessly.

To address these flaws, like key duplicates in an FHT and the complicated setup/update in a BFHT, we propose a scalable hash-based IP lookup scheme using BFs and a fingerprint filter (FF) in *pipeline*. Our scheme, a pipelined indexing hash tree (PIHT) without pointers, encodes keys' indexes to a fingerprint (or key) table and approximately generates a few indexes in a key query through a binary search in a prefix tree which is built based on index bits of a key table. Pipelined BFs play a role in searching for a key's fingerprint in a b-ary tree, $b \in \{2, 4, ...\}$, while an FF which is the most memory-efficient set representation guarantees the less number of expensive false indexes to an off-chip key table in the worst lookup case. Since memory bank system in a pipeline stage supports multiple bank accesses in one clock, we can achieve $O(1)$ pipelined throughput of insertion, deletion, and query operations.

This paper has the following contributions:

- A PIHT provides indexes to a fingerprint (or key) table using pipelined BFs and an FF without pointer overhead.
- New algorithms on *insert*, *query*, *delete* operations are proposed for the PIHT and their pipelined throughputs per clock are $O(1)$.
- By using a Lulea bitmap (LB) [16], we reduce the next-hop duplicates as well as convert prefixes into a smaller number of collapsed prefixes of shorter bits.
- One path through both PIHT and LB phases ensures a perfect-match prefix lookup for IP lookup.
- In IP lookup simulation with scalable routing tables, an optimized PIHT in a 8-ary tree shows on average 4.5 and 50.1 times efficiencies of memory and power, respectively, over contemporary schemes.

The rest of the paper is organized as follows. Sec. II presents an overview on IP lookup schemes and their pros and cons. Sec. III discusses about basics of a BF and an FF. Then, Sec. IV shows the detailed PIHT build for a perfect IP lookup match and the false index path incurred by false positives of a series of BFs. Sec. V shows memory comparison of hash lookup mechanisms, and it also has memory and power comparisons of other IP lookup schemes for addressing a scalability issue. Then, a conclusion in Sec. VI follows the experiment in Sec. V.

II. Overview on IP Lookup Schemes

A. TCAM- and Trie-based IP Lookup Schemes

The TCAMs were developed with the ability to store an additional "Don't Care" state thereby enabling them to retain single-clock-cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes with the cost of storage density, access time, and power consumption. Although these disadvantages are resolved in trie-based IP lookup schemes, trie-based schemes still suffer from the worst case lookup time which is bounded to $O(W)$ where W is the IP address length. The fundamental issue with trie-based schemes is that its performance and scalability are mainly tied to address length. As Tables I and II depict feature differences among TCAM-, trie-, and hash-based IP lookup schemes, and they further demonstrate the urgency and the need to develop an efficient hash-based IP lookup scheme.

schemes	
Trie	$O(W)$ †
Hash	$O(1)$
TCAM	1

TABLE I
Lookup complexities

† W: # of IP address bits

	TCAM	SRAM(hash or trie)
clock†	266	400
Power ‡	≈15	≈0.1
Cell°	16	6

TABLE II
Hardware features from [17]

† MHz unit ‡ Watts unit
° # of transistors per bit

B. Hash-based IP Lookup Schemes

1) An approximate-match BF hashing:
Authors in [9] first propose an IP lookup scheme by using a set of counting Bloom filters to reduce off-chip memory accesses. This scheme associates each counting BF with a unique prefix length, and a set of off-chip hash tables are constructed for each distinct prefix length.

Although on-chip BF preprocessing minimizes the number of hash probes per lookup, there are two drawbacks: a BF's approximate match and its off-chip hash table. Since the BFs provide only approximate matches, the matches' confirmation is necessarily made through corresponding hash tables stored in slow off-chip memory. Thus, the lookup speed depends on the off-chip memory speed, and if an off-chip DRAM is used to reduce cost and power consumption, then the off-chip memory access time is further increased. Suppose an off-chip FHT [11] is used in place of a legacy hash table since an FHT surpasses the legacy hash table in search time by a factor of four or more. However, such an FHT inevitably suffers from a pointer overhead, key duplicates, and complicated key update as described in the following.

2) An fast hash table of a perfect match:
By birthday paradox [18], there is a possibility of a collision even in a perfect hash function. To resolve this collision among keys, a chaining method with a linked list has been widely used. An FHT couples a linked-list HT [14] with a memory efficient counting BF (CBF), so that a key searching is expedited by choosing the shortest linked list and a smaller number of memory accesses are achieved at the cost of duplicate keys [11].

However, the higher the number of hash functions is, the larger the number of duplicate keys is due to sharing linked lists. In 100 runs of an FHT build with synthetic 2^{15} keys, we found that the number of duplicate keys is proportional to k

and its fitting curve noticeably becomes a super-linear to k as illustrated in Fig. 1. Because a high-speed router needs a large k value to choose a shortest linked list among k linked lists, a 160Gbps router requires k=29 in order to achieve this. In k=29 an FHT has at most 6 times duplicate keys. This FHT is not scalable in terms of memory usage.

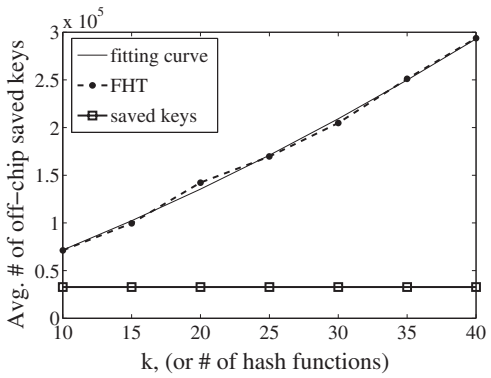

Fig. 1. Duplicate keys in off-chip according to k when n=2^{15}.

In addition to the duplicate key overhead, an FHT suffers from additional limitations that are described here. Firstly, the *insert* and *delete* operations take approximately k times to complete and these operations are not suitable for a dynamically changing set. Secondly, in order to catch up with reduced collisions at a high speed, an FHT needs a plethora of on-chip starting pointers to off-chip linked lists and it holds a large wasted portion of on-chip pointers. Also, the off-chip linked lists themselves incur additional pointer overhead while our scheme does not have pointer problem and key replication.

3) A Bloomier filter and other hash schemes: While a linked list needs sequential key accesses for off-chip matching in [9, 11], a BFHT generates an index to a key table and its corresponding next-hop table so that finding a key is completed with a collision-free and *perfect* $O(1)$ lookup. However, a BFHT inherits two disadvantages of a Bloomier filter: setup failure and the setup complexity $O(n \log n)$ of n keys. A Bloomier filter suffers from a dynamic membership change, because an index table stores a key's k hash values based on its neighborhood keys and this neighborhood is collected by avoiding collision with other keys' hash values. Thus, a BFHT using a Bloomier filter needs the same time complexity for updating prefixes. Also, although a BFHT introduces prefix collapse to reduce the number of prefixes, there is inflation of next-hops since a BFHT's monolithic bitmap does consider a tree relationship among prefixes.

There are other hash schemes for general packet processing [19, 20] to reduce the maximum load of a hash table. Two schemes use multi-leveled hash tables while a scheme in [20] uses the open addressing for collision and a scheme in [19] uses both the linked list chaining and open addressing. However, if the open addressing is adopted there is a chance of a *crisis* where a key cannot be inserted into any multi-leveled hash tables. Thus, these schemes' application to the IP lookup causes a setup failure as in a BFHT, which can cause a malfunction during the IP lookup operation. Even if a linked list chaining can be used to resolve the crisis, Sec. II-B.2 shows that a recently proposed FHT using the chaining suffers from several overheads. However, unlike a BFHT and schemes in [19, 20], our PIHT does not cause a setup failure. Also, our PIHT uses key and rule tables for a perfect-match $O(1)$ lookup through pipeline.

4) Prefix conversion for hash-based IP lookup schemes: Hash-based schemes, like a BFHT and an FHT, need a scheme to support wildcard bits in prefixes since hash provides a singleton match. A controlled prefix expansion (CPE) in [21] is to transform a set of prefixes by combining prefix expansion and prefix capture to reduce any set of arbitrary length prefixes into an expanded set of prefixes in optimized sequence of length. However, a CPE causes inflating the numbers of prefixes and next-hops. In contrast, a prefix collapsing (PC) converts a prefix of length x into a single prefix of shorter length x-l by replacing its l least significant bits with a wildcard [12]. Although a PC reduces the number of prefixes, there are still a next-hop's duplicates, depending on stride length l. Thus, we adopt a Lulea scheme [16] to remove the next-hop duplicates while retaining the benefit of the reduced number of prefixes.

III. Preliminary of Memory-Efficient Hashing

A. A Bloom Filter

A BF for representing set S=$\{e_0, e_1, ..., e_{n-1}\}$ of n keys is described by an m-bit array and is initialized to 0. A BF uses k independent hash functions $h_0, h_1, ..., h_{k-1}$ which map the keys uniformly within the range of [*0:m-1*], and this k-parallel hashing implies that a BF memory module needs to support k randomly-addressed memory reads in parallel when performing an one-clock lookup. For each key $e_{j'} \in S$, the bits indexed by $h_{k'}(e_{j'})$ are encoded to 1 for $0 \le k' \le k$-1, $0 \le j' \le n$-1. To query that key y is in S, we check whether all bits in the BF indicated by $h_{k'}(y)$ are set to 1. If so, a BF returns 'yes' for the query. If not, then clearly y is not a member of S. Even if all of the indexes by $h_{k'}(y)$ are set to 1, there exists a probability that key y does not belong to set S due to the random gathering of k bits set to 1 for independent $e_{j'}$s.

The probability calculation is the following: After all n keys of S are hashed k times into the BF, the probability that a specific bit is still 0 is asymptotically p=$(1\text{-}1/m)^{kn} \approx e^{-kn/m}$, given our assumption that hash functions are perfectly random. Thus, the probability of an f-positive by randomly choosing k bits of 1 in an m-bit array is bounded as follows

$$f \ge \left(1\text{-}(1\text{-}1/m)^{kn}\right)^k \approx (1\text{-}p)^k \ge (1/2)^{m \ln 2/n} \quad (1)$$

according to the result of Broder and Mitzenmacher [22]. After some algebraic manipulation, they claim that $f \le \epsilon = 2^{-w}$, where w is termed the query precision, requires

$$m \ge (n(\log_2(1/\epsilon))/\ln 2 \approx 1.44n \log_2(1/\epsilon) = 1.44nw. \quad (2)$$

Also, k becomes w in an optimal configuration. Based on Eq. (2), we can conclude the linear property between m and n:

Lemma 1 (Linear Property) *Linear property between m and n exists in Eq. (2) because given f requires that variable n is linearly proportionate to variable m.*

We have linked the theoretical relationships between k, m, n for the required f-positive, ϵ, in a query. If a BF is to be used for the IP lookup despite producing an approximate query result, a lookup precision w should be at least 29 ($\approx -\log_2 1/500M$) for the 160Gbps routers because a collision in 500M lookups in a second is not tolerable when meeting bandwidth requirement satisfaction. The implementation of such a BF memory requires 29 read ports for the same number of hash functions, but this is not feasible since memory cost and power are superlinearly proportional to the number of read ports. To lessen these overheads, we use a segmented BF (SBF) with a memory banking. Using this scheme is far more practical with a commodity memory, such as IDT's product [23] of high-speed bank-switchable memory which is organized into a 64-bank memory array.

In an SBF, an m-bit vector is divided into k $m'(=m/k)$-bit subvectors, each put in an independent memory bank. k hash functions with the range [*0:m'-1*] are assigned to their corresponding subvectors, and an one-clock query in an SBF is based on k indexed values in k subvectors (or banks) together. Although a SBF's memory banking scheme removes the multiport overhead, the SBF's false positive probability, f', becomes the same BF's f as follows:

$$f = \left(1-(1-1/m)^{kn}\right)^k = \left(1-(1-1/km')^{kn}\right)^k$$
$$= \left(1-(1-k/km'+o(1))^n\right)^k \approx \left(1-(1-1/m')^n\right)^k = f' \qquad (3)$$

where a small o function is negligible at a large m' value. We use an SBF memory banking implementation scheme for a BF application to the IP lookup with the 'BF' notation.

B. A Memory- and Power-Efficient Fingerprint Filter

An FF is regarded the most memory-efficient set representation scheme [22]. Authors in the paper find that m needs to be essentially $n\log_2(1/\epsilon)$ for any data structure scheme with an f-positive rate bounded by $\epsilon=2^{-w}$. Thus, an FF can be considered as a table of n keys' fingerprints (FPs), each with w bits. Also, since an FF needs a read port of w-bit width compared to a BF with w read ports of 1-bit width, an FF is more memory-, power-, and die-area-efficient than a BF. However, although the FF memory size for required ϵ is 1.44 times smaller than a BF compared to Eq. (2), an FF does not have an efficient indexing mechanism to find a key's fingerprint as a BF does use k hash functions. Thus, if network applications use a power- and memory-efficient FF, designing an efficient indexing scheme with an FF is necessary. In the following section, we propose such an indexing scheme, i.e., an PIHT with BFs, for an FF in IP lookup application.

IV. Our IP Lookup Architecture with a PIHT

Hash-based IP lookup schemes have received favorable attention because of power and memory efficiency [11, 12]. Recall that the existing hash schemes themselves [11, 12, 19, 20] suffer from setup failures, more complicated update than $O(1)$, and pointer overhead as shown in Sec. II-B. To overcome these problems, we describe a hash-based scheme with a PIHT and a Lulea bitmap for a scalable IP lookup.

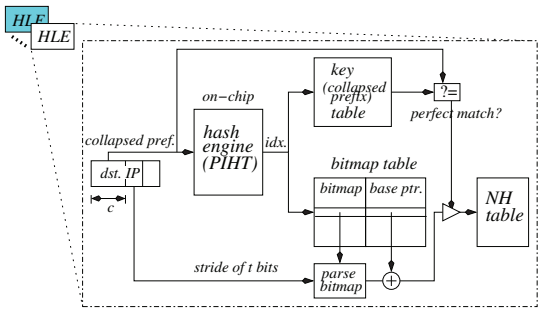

Fig. 2. IP lookup architecture with parallel Hash Lookup Engines (HLEs) for a wildcard support. Each HLE has different collapsed-prefix length c.

Fig. 2 illustrates our PIHT-based IP lookup architecture. We divide each prefix into a collapsed prefix and a stride. The strides under the same collapsed prefix are encoded in a bitmap. For example, 2-bit strides '0*' and '1*' from prefixes A('0100*') and B('0101*') in c=3 are expanded into a string 'AABB'. The brute-force bitmap would be '1111'. However, we know that a string with repetitions (e.g., AABB) can be compressed using a bitmap denoting repetition points (e.g., 1010) with a compressed sequence (i.e., AB). Among bitmap schemes [16, 24], for simplicity we choose the Lulea scheme [16] compressing repeated information without paying a high penalty in search time. After the prefix division and bitmap encoding as in Fig. 3, each HLE saves collapsed prefixes of the same length c in a key table for a perfect match and their corresponding bitmaps in a bitmap table in order to index into the next-hop table.

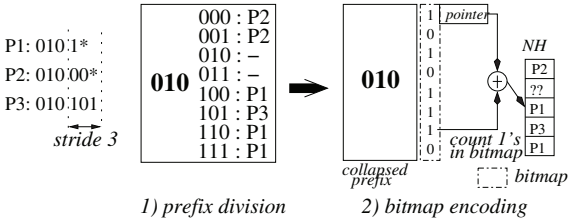

Fig. 3. The prefix division and bitmap encoding for prefixes P1, P2, and P3 in order to support a singleton hash match. c=3, t=3.

For an IP lookup operation, initially each HLE strips the first c bits and their following t bits from a destination IP, does hash based on c bits, and accesses a next-hop table by parsing a PIHT-indexed Lulea bitmap, if the hash is perfectly matched. A match with the longest collapsed prefix is the final match for a given IP lookup among all its perfect matches. The following sections show the details of our hash lookup engine.

A. Basic Principles of a Pipelined Indexing Hash Tree

A node in a binary search tree [14] has two *explicit* pointers to its children and a key to compare in a query. Basic operations on a n-key binary search take $O(\log n)$ time on average and $O(n)$ in the worst case. In a query of a key k, we compare a node key with the key k based on <, ==, and > operators and make a left sub-query, a match, and a right sub-query, respectively, after the key comparison.

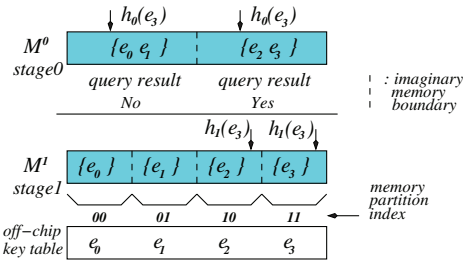

Fig. 4. Two partitioned memories in pipeline and indexing for key e_3

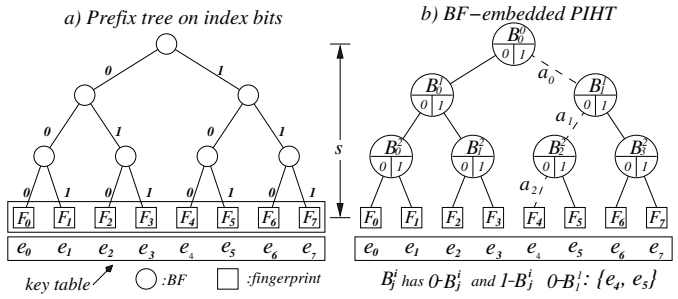

Fig. 5. A conceptual tree construction of a PIHT for 8 fingerprints (F_0,$\cdots$,F_7) and keys ($e_0, \cdots, e_7$)

In contrast to the comparison operators, we use BF query results in order to make such sub-query directions in a PIHT. Fig. 4 shows the basic principle of our PIHT scheme with 4 keys. Suppose that there are two bit-vector memories M^0 and M^1 with hash functions h_0 and h_1. Next, we virtually separate M^0 and M^1 into two and four memory partitions and four memory partitions, respectively. Let each partition have its corresponding BF-encoded subset of 4 keys as in Fig. 4. For a query of key e_3, we look up bits indexed by $h_0(e_3)$ in two memory partitions in stage 0. Suppose the second partition returns a true positive in a BF query while the first partition yields a negative. According to these query results in stage 0, we only look up the third and fourth memory partitions in stage 1. As long as the BF query result in the third memory partition is not a false positive, we can use an index 11_2 of the fourth memory partition in order to access a key table for the key's confirmation. Note that if there were no false positive, this query traversal would be similar to a binary search, and that there is no explicit pointer in the quasi binary search.

Specifically, in a PIHT we use a group of BFs to design an indexing scheme to index into a fingerprint table and consequently a key table. Such an index scheme works in the same way as a binary key search in pipeline. For the pipelining, we conceptually embed a binary prefix tree with a set of BFs into multiple memory modules. A PIHT node consists of two BFs, 0-BF and 1-BF, which cover different subsets and direct a key query into the left child and the right child, respectively. A key query in such a node returns 'x', $x\in\{0, 1\}$, in one clock cycle if an x-BF returns a query positive, either false or true. For instance, if a 0-BF in a node returns a query positive, the next key query proceeds to the left child of the node. A key query starts from a PIHT's root node, and this query proceeds to its corresponding children nodes in the next pipeline stage based on the query results in a current node. In the last pipeline stage, for a given key query the PIHT generates indexes to a fingerprint table with the query results of PIHT nodes.

B. Building a BF-embedded PIHT

A PIHT is a memory efficient hash mechanism to index a fingerprint table because it never uses pointers in implementing a tree for accessing a fingerprint table. In addition, if each stage's memory banking can process queries on nodes in one clock, then there is no memory stall in the pipeline; thereby we can achieve a one-clock lookup to a fingerprint table at the last stage. Since researchers have produced successful results on memory banking [25, 26] and pipeline [26, 27] for network memory, our PIHT scheme in memory banking becomes feasible. Also, our PIHT-based IP lookup fits in embedded SRAM since the current ASICs support can be as large as 10Mb SRAM. These facts are beneficial to a modern IP lookup scheme that demands high-speed lookups with a large number of prefixes. The details of how to build a PIHT is as follows.

In a macro view, a PIHT for n keys (i.e., collapsed prefixes) in power of 2 is composed of s=$\log_2 n$ layers (or pipeline memory stages interchangeably) and a key-table index space is partitioned rectangularly of $n\times s$ 0/1 bits to create a binary prefix tree. Fig. 5 shows the PIHT partition where 8 fingerprints (keys) are stored in an FF (a key table) consecutively and a binary prefix tree is built in the key-table index space.

In a micro view, a PIHT node is designed to act as a binary predicate for a key query on dual 0-BF and 1-BF. Let PIHT node B_j^i denote the j-th binary predicate in layer (or stage) i, hereinafter $0\le i\le s$-1. Then, if key $e\in S$ is to be inserted at index address A=$a_0 ... a_{s-1}$, where $a_t\in\{0, 1\}$, $0\le t\le s$-1, a BF, denoted as a_i-$B^i_{a_0\cdots a_i}$ at each layer i, is involved in encoding key e just like a legacy BF does. In this hierarchical partitioning and encoding, B_j^i covers n_i=$n/2^i$ keys whose indexes in a key table range from $j\cdot 2^{s-i}$ to $(j+1)\cdot 2^{s-i}$-1. For instance, B_1^1 covers set $\{e_4, \cdots, e_7\}$ while 1-B_0^1 considers set $\{e_2, e_3\}$ as in Fig. 5 b).

C. Node-to-memory Mapping

Although the nodes are conceptually partitioned in a layer according to their key sets, they concatenate each other in a memory bank module and are separated by their base addresses. That is, as Eq. (2) specifies the required BF memory size for a bounded f-positive, B_j^i has base address $j\cdot 1.44 n_i w_i$ in memory M^i_{on}, where w_i is a query precision of a BF on layer i. Also, as the equation states that m is linearly proportional to n for a given f, given f_i=2^{-w_i} for a BF on layer i, the total memory of M^i_{on} for BFs on layer i is of size $2^i(1.44 n_i w_i)$=$1.44 n w_i$. Since the B_j^i size specifies the next B^i_{j+1}'s base address on the same stage in order to maintain a tree structure, a PIHT does not use explicit pointers, but an *implicit* base index for each BF sub-block in M^i_{on}. That is, B_j^i's memory starts at $j\cdot 1.44 n_i w_i$ in M^i_{on}. Thus, a large memory size reserved for pointers is saved.

Fig. 6 illustrates our PIHT architecture for an HLE where a set of memory modules is in pipeline and key and bitmap tables are accessible by indexes generated from a PIHT

for a perfect lookup match. In each pipeline stage, a logic module records a set of nodes B^i_js to be probed by accessing their corresponding banks in memory module. If a node of partial index $a_0...a_{i\text{-}1}$ has a query positive in 0-B^i_j (1-B^i_j), an augmented partial index $a_0...a_{i\text{-}1}0$ ($a_0...a_{i\text{-}1}1$) for the left (right) child node of the node is saved for the next logic module. As long as the number of memory banks in a pipeline stage is larger than the number of nodes to be probed in the stage, there is no pipeline stall because the parallel access to memory banks in one clock cycle is designed in the memory bank system. Thus, the PIHT pipeline system with memory banking system is designed to provide one-clock throughput during insertion, query, and deletion operations. The details of these operations are the following.

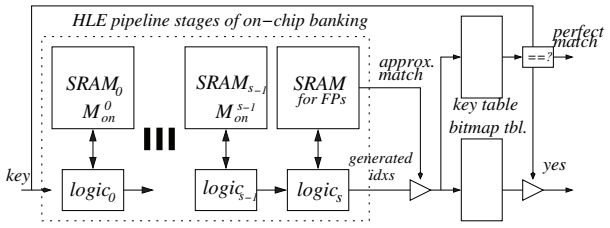

Fig. 6. A pipelined PIHT architecture with s+1 stages for an HLE

D. `Insert` Operation in a PIHT

When we insert a key with an arbitrary designated address onto a PIHT, there is a corresponding node to the address on each PIHT layer. Thus, while traversing the nodes from a PIHT root based on the address, we do a basic BF key insertion onto each node. The insertion of key e_4 at index 100_2, for example, means that 1-B^0_0 of layer 0, 0-B^1_1 of layer 1, and 0-B^2_2 of layer 2 are involved in BF insertions.

Algorithm 1: `insert-i()`

Input: key e, and partial index $A'=a_0a_1\cdots a_i$ in binary bits
Output: Encoded PIHT node about e on stage i

1 m_i=1.44n_i; $j=a_0\cdots a_i$;
2 **for** t=0 **to** k-1 **do**
  // $h_t(e)\in\{0,\cdots,m_i\text{-}1\}$, M^i_{on} of $2^{i+1}m_i\times 1$ bits
3 $idx=j\cdot m_i\cdot t$; // Find bank's base index in B^i_j
4 $M^i_{on}[idx+h_t(e)]$=1; // M^i_{on}: BFs on layer i
5 **end**

Algorithm `insert-i` shows the detailed insert operation on stage i as simple as that for a BF, except a lined **for** loop is for parallel memory banks. After the last stage s-1, key e, its fingerprint (FP), and its associated bitmap are saved in $M_{key}[A]$, $M_{FP}[A]$, and $M_{bitmap}[A]$, where A is the designated address for key e. Thus, the per-clock throughput of a pipelined insertion operation to key and bitmap tables is 1 despite the fact that the memory access complexity during the pipelining is $O(\log_2 n)$. In contrast, schemes in [11, 12] claim throughput and memory access complexities of $O(nk^2/m+k)$ and $O(n\log n)$, respectively.

E. `Query` Operation Making Indexes in a PIHT

Once all keys are saved in a key table and the keys' indexes are encoded in a set of the corresponding BF memory modules, the ultimate remaining PIHT goal is to search a key's index by performing a fast key query operation. Once we find the key's index, we can easily access to an on-chip fingerprint table with the index for an approximate match and then to an off-chip key table with the same index for a perfect match. There are two kinds of search patterns, an unsuccessful search (US) in which a key is relentlessly searched even though it does not exist in a PIHT, and a successful but time-consuming search (SS) in which a key is to be searched in a PIHT. Before we discuss these two kinds of searches, let us introduce definitions of an index path and a false index path here.

DEFINITION 1 (INDEX PATH)
In a PIHT, an index path, or ***i-path****, is defined as a series of x-B^i_js, $x\in\{0,1\}$, used in* `insert` *operation and is hierarchically connected to each other from layer* 0 *to layer* s-1 *in order to produce a sequence of index bits. The sequence of index bits from x-B^i_js, or a series of x-bits, is also matched with an assigned index of a key saved in a fingerprint (key) table and the bit sequence size of the series is to be s.*

As a corollary, we can conclude that in `query` for key e which is previously encoded by `insert`, an i-path for the key e should show up as BFs return 'yes' for their true membership, i.e. no false-negative.

Suppose there is no f-positive in a key query on a BF. Then, there is no other index path in a PIHT for this key query. This assumption ensures to locate the exact corresponding i-path to a key table. However, a false index to a fingerprint table, other than a dedicated i-path to a key, is made possible due to the f-positives derived from irrelevant x-BFs in a PIHT. For example, suppose key e_4 is inserted with i-path 100_2 in Fig. 5 and a query of e_4 has progressed along true positives of 1-B^0_0 and 0-B^1_1. Then, a query on B^2_2 may return two positives from 0-B^2_2 and 1-B^2_2. Since we do not know which positive is true, this query result with these positives give ambiguous indexes 100_2 and 101_2. Thus, this ambiguity needs to be resolved by two accesses to a fingerprint table, and a memory banking system can process such multiple bank accesses in one clock if the ambiguous indexes head for different banks. Given a query for an s-bit i-path, there are totally 2^s-*1* false index paths because each B^i_j is independent and identically distributed.

Even if it is possible that there is a set of BFs (or nodes) giving f-positives in a query, nodes that are only hierarchically and mutually connected to their parents and children nodes of positives, true or false, can be a part of an f-path. Thus, f-positives from the rest BFs can be ignored. In the previous example of e_4 with i-path 100_2 in Fig. 5, although 1-B^1_1 can randomly make an f-positive in query, this f-positive can not be a part of an f-path if there is no f-positive in B^2_3. By the definition of an f-path, the probability of an f-path is cumulatively calculated as the product of f-positives from all the nodes along the f-path. Thus, the number of f-paths is expected to be extremely small so that each stage has a few

nodes to be probed and the number of f-paths in the last stage is bounded to a small value. We formalize the bounded number of nodes on stage and f-paths for US and SS cases in the following.

1) The false indexing to a key table in a US:
Besides the design issue of producing a low probability of f-paths to a fingerprint table in an SS, it is equally important that the probability of f-paths in a US is also lower. Unlike an SS, there is no i-path for a given key in a US, meaning that all BFs in a query return positives as f-positives. However, there is a chance that nodes, those that are connected to each another in a hierarchical path, make one or more f-paths. In contrast to an f-positive in an FHT [11] leading to expensive off-chip memory accesses, an f-path caused by a series of f-positives with *hierarchically* connected B^i_js in each layer i requires one index access to an on-chip fingerprint table.

There are 2^i nodes on PIHT layer i and it is possible that any of them become a part of an f-path. To become such a node, an i-length path from a root to the node should be made by i f-positives from x-B^i_js. Thus, the average number of stage i' nodes to be probed is the following:

$$N^i_U = \sum_{j=1}^{2^i} f^i \cdot 1 = f^i \cdot 2^i. \tag{4}$$

Eq. (4) has maximum value, 1, at stage 0, and the larger i is, the smaller the value of Eq. (4) is. Finally, on the last stage this value is small enough to converge to zero as follows:

$$\lim_{i\to\infty} N^i_U = \lim_{i\to\infty} f^i \cdot 2^i = \lim_{i\to\infty} 2^{-w_i i} \cdot 2^i = \lim_{i\to\infty} 2^{(1-w_i)i} = 0, \tag{5}$$

where f=2^{-w_i}, w_i≥2, for a BF on layer i.

2) The false indexing to a key table in an SS:
The situation in an SS is very different from that of a US, because there must exist one i-path with a number of possible f-paths in a highly low probability while there is no i-path in a US. Fig. 7 shows an example of 8 keys where along an i-path $a_0a_1a_2$ of key e_4 there are 3 dangling trees contributing to f-paths, if any. All dangling trees are attached to the e_4's i-path and they can contribute to a number of f-paths and all are with different probabilities related to Eq. (4). Thus, the average number of stage i's nodes to probe is one for an i-path plus the summation of $N^j_U/2$, $1 \le j \le i$ for f-paths as follows:

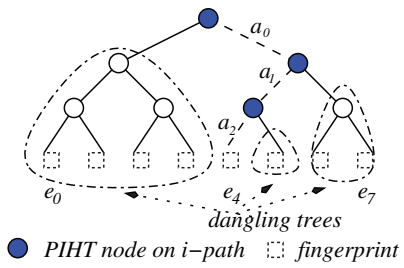

Fig. 7. A PIHT of 8 keys with i-path $a_0a_1a_2$ and 3 dangling trees.

$$N^i_S = 1 + \sum_{j=1}^{i} N^j_U/2, \qquad i > 0. \tag{6}$$

In i=0, there is only a starting root node. Since the Eq. (4) value exponentially gets decreased as i, on the last stage N^i_S is small enough to converge to 2 as follows

$$\lim_{i\to\infty} N^i_S = \lim_{i\to\infty} 1 + \sum_{j=1}^{i} f^j \cdot 2^{j-1} = \lim_{i\to\infty} 1 + (1-2^{-i})/2 < 2 \tag{7}$$

where f=2^{-w_i} and w_i≥2 is a BF lookup precision on layer i.

Note that each PIHT node has two BFs 0-B^i_j and 1-B^i_j, each with w_i memory banks. Based on Eqs. (7) and (5), we can say that if each stage's memory module has $2\times N^i_S \times \log_2 1/f$, i.e. $2\times2\times w_i$, memory banks, the whole number of nodes to probe can be processed in one clock cycle without pipeline stall in a stage. Thus, as long as access time in the memory bank system satisfies the worst case lookup limit, like 2ns for 160Gbps, we can say that a PIHT can process the worst-case number of lookups on average.

3) The detailed algorithm for the `query` *operation:*
A complete `query` operation consists of `query-i` shown in **Algorithm** `query-i` working for layer i≤s-1. Note that given the number of the partial indexes in L, the number of BFs to be probed doubles since each binary tree node has two children. Also, this alrogrithm's throughput per clock is 1 under the condition that $2w_i|L|$ is bounded by the number of memory banks that are supported by the memory hardware without any overhead. Now, in terms of a key query, the time complexity to finish the key query from a PIHT root is $O(\log_2 n)$. However, the insertion or query operations can be back-to-back each another in pipeline. For instance, the `query-i` operation is not affected by either of the insertion or query operation on the previous stage i-1 or the next stage i+1 since all these operations are independent memory access and do not constitute computer architectural hazard. Thus, we can claim that a pipelined query operation throughput per clock is 1.

Algorithm 2: `query-i`()

Input: M^i_{on} for layer i≤s-1, list L of partial indexes (or nodes) found up to layer i-1, including i-path, and key e
Output: A set of partial $A = a_0 \cdots a_i$ of i+1 bits, including segments of f-paths

```
   // S : Set of partial paths. L={A_0, ··· A_{|L|-1}}
1  S = ∅;   m_i=1.44 · n_i · 1;   A_t = L[t];
2  for t = 0 to | L |-1 do
3      cnt_0=cnt_1=0;
4      for t=0 to k_i-1 do // Note that k_i=w_i
           // idx_0 (idx_1) is base idx. of 0-B^i_j (1-B^i_j)
5          idx_0=2A_t · m_i · t;   idx_1=(2A_t + 1) · m_i · t;
6          if M^i_on[idx_0+h_t(e)]==1 then cnt_0++;
7          if M^i_on[idx_1+h_t(e)]==1 then cnt_1++;
8      end
       // concatenate 0 or 1 bit at the end of A_t
9      if cnt_0==k_i then S=S∪ A_t · 0;
10     else if cnt_1==k_i then S=S∪ A_t · 1;
11 end
12 return S;
```

F. The `Delete` Operation in Dual PIHTs to Update Prefixes

`Delete` operation is not as easy as the `insert` because a basic BF does not support the deletion of a key that is inserted

in the BF. However, we couple PIHTs, an *l*-PIHT and a *r*-PIHT, to rotate a target PIHT for the `insert` operation and another target PIHT for the `delete` operation. Once one PIHT is full of previous n keys, the `query` operation stays with the PIHT. But if set S is dynamic but limited in size n, a new PIHT takes care of new key insertions by working on BFs in the new PIHT as well as a bit in a *valid bit array* (VBA). A new key's index is given from the head of a *free address queue* (FAQ) where every `delete` puts the deleted-key's index at the *FAQ*. Also, the old PIHT handles `delete` operation by simply setting off a bit in a corresponding VBA of the PIHT. Checking two VBAs with the indexes given by two PIHTs ensures that an unnecessary fingerprint-table access is blocked. Also, when all n keys are encoded in one PIHT, i.e. the moment that an FAQ is empty, the other PIHT needs to be initialized to 0 for the next set of `insert` operations with the BFs initialized in the PIHT. By using two rotated PIHTs, we do not need counting BFs, which otherwise would cost 4 times additional in memory size. Thus, using two PIHTs without CBFs saves 2 times the memory space. **Algorithm** `delete` shows the detailed algorithm.

Algorithm 3: `delete()`

Input: l-PIHT, r-PIHT, key e, and its fingerprint F_e
Output: Updated *valid bit array* V_l or V_r for l- and r-PIHT

```
1 S_l=query-s-1(l-PIHT,e);      S_r=query-s-1(r-PIHT,e);
2 if ||S_l ∪ S_r|| ≥ 1 then
3     foreach A ∈ S_t, t ∈ x{l, r} do // FP == fingerprint
4         if V_t[A]==1 and M_FP[A]==F_e and M_key[A]==e then
5             V_t[A] = 0; M_FP[A]=0; M_key[A]=0;
6             Save A in FAQ;
7         end
8     end
9 end
```

G. The optimization of a PIHT Memory in a b-ary Tree

In general, the height of a binary prefix tree on the index space of n keys is s=$\log_2 n$ which is the PIHT height. However, when a b-ary prefix tree, b∈$\{4, 8, \cdots\}$, is adopted in a PIHT, the new PIHT's height becomes $\log_b n$ while the index notation is based on the b-base number system. Thus, using a b-ary prefix tree shortens a PIHT height, thereby the memory reduction. However, this change does not create any index addressing disturbance. That is, index 0100_2 for e_4 in 2-base is simply transformed to 10_4 in 4-base with no key-table change.

In a b-ary tree adoption, a node B^i_j covers n_i=n/b^i keys while one of b BFs in the B^i_j covers a b-th of n_i. Due to Lemma 1 of *Linear Property* between m and n, it is the number of layers, s=$\log_b n$, not the size of M^i_{on}, that is affected by the new b-ary tree adoption. In using the b-ary prefix tree, the average number of f-positives, bf, among b BFs in a PIHT node is increased due to the BFs' binomial distribution. Once this factor is considered by increasing lookup precision w_i, the PIHT memory size in b-base number becomes

$$M_b = 1.44n \sum_{i=0}^{\log_b n-1} (w_i + \log_2 b\text{-}1) + nw, \qquad (8)$$

where w is the worst case lookup precision for a router's bandwidth requirement. Fig. 8 shows the reduced memory size in using a b-ary tree rather than a binary tree in a PIHT for 2^{22} keys. As b is larger in a b-ary tree of b-base, the reduction becomes significant so that the PIHT of w_i=6 in 32-base saves 2.4 times memory than that in 2-base. Also, the number of pipeline stages is reduced.

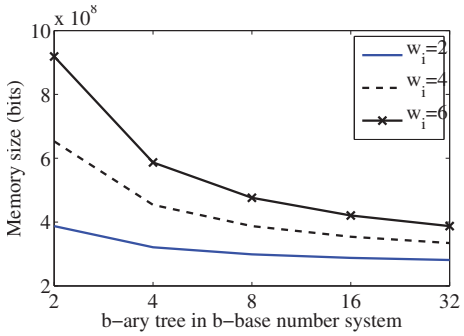

Fig. 8. Total PIHT memory in a b-ary tree. n=2^{22}. w=29.

V. The Experimental Results for IP Lookup Schemes

In this section, we evaluate a PIHT scheme among the latest hash schemes: an FHT, a linked-list Peacock, and a BFHT in terms of memory and operation complexity. Also, key update comparison of these hash schemes is delivered. After these comparisons for hash-based schemes are discussed, we compare a PIHT-based IP lookup scheme against the representative schemes from the trie and TCAM-based IP lookup schemes, with regards to power, memory, and scalability.

To evaluate the implementation cost, power, and memory for these IP lookup schemes, we utilize CACTI [17] and a tool [28] for SRAM and TCAM measurements, respectively, and a parameter of 90*nm* technology process is used in these tools. In the evaluation, each scheme is considered with different prefix distributions for addressing a scalability issue. Among the four distributions, only one is a real recent routing table [29] while the rest three distributions of the larger size are synthesized according to the same prefix distribution pattern as the real one. Also, we set the stride size s to 4 as in [12].

A. The Comparison among Hash-based schemes

Linked-list hash schemes reduce the probability of collisions at the cost of a larger number of pointers to linked lists, and the average length of a linked list determines the number of memory access due to sequential link accesses. One memory access is required to take 2ns(=40 bytes*8/160G) for the worst case lookup in a 160Gbps router. In such a lookup case, the collision probability of larger than a 1-in-500M($\approx 2^{-29}$) is not tolerable in a linked-list-based hash since traveling the linked list needs the same number of memory accesses as the linked list length [12]. In contrast, the number of PIHT nodes to be probed, which is concerned by our PIHT-based scheme, only needs to be bounded within the number of memory banks in each pipeline stage.

1) The memory comparison:
Since each component in a hash mechanism uses a different memory size, we separately measured all components' memory for a lookup precision w=29. Fig. 9 shows the memory

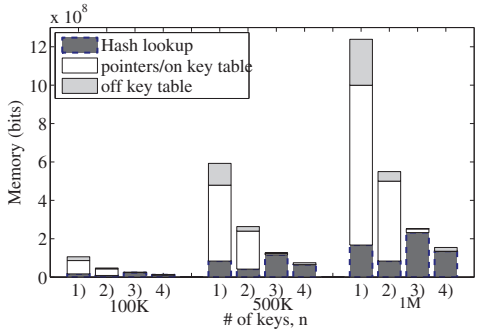

Fig. 9. On- and off-chip memory size of four schemes: 1) CBF [9] +FHT [11], 2) CBF+Peacock [19], 3) BFHT [12], and 4) PIHT of b=8

sizes of four schemes (CBF+FHT, CBF+Peacock, a BFHT, and a PIHT) at various n. Schemes 1) and 2) use CBFs marked as 'hash lookup' and an on-chip list of pointer buckets marked as 'pointers' while they put off-chip linked lists with keys marked as 'off key table'. Similarly, a PIHT uses an on-chip hash engine and a key table marked as 'hash lookup' and 'off key table', respectively. From the figure, it is apparent that our PIHT scheme records the least memory size at various n and linked-list hash schemes 1) and 2) suffer from significant on- and off-chip pointer overheads. It is observed that a PIHT scheme saves 2.0 and 7.3 times on-chip memory on average than the BFHT and CBF+FHT schemes, respectively.

2) The operation complexities:
It is identified that only a negligibly small fraction of prefix updates actually require a new key to be inserted into a hash scheme [12]. However, the update throughput can affect the lookup throughput if it depends on several scalable variables. The time complexities of insertion, deletion, and query operations in each hash scheme are different as shown in Table III. If each operation with a different time complexity cannot be processed in one clock, the operation throughput is less than 1 over the operation's time complexity. The operations in an FHT and a BFHT schemes are not designed for pipelining while those of our PIHT are. That is, each pipeline stage processes one of the operations independently and the last stage completes an operation at every clock cycle. Thus, we can assert that PIHT operations achieve a constant throughput independent of the variables shown in Table III's last row.

Operation	`insert`	`query`	`delete`
FHT	$^{\circ}O(nk^2/m+k)$	$^{\circ}O(1)$	$^{\circ}O(nk^2/m+k)$
[15]	$^{\circ}O(n\log n)$	$^{\circ}O(1)$	$^{\circ}O(n\log n)$
PIHT	$^{\circ}O(\log_2 n)/^{\dagger}O(1)$	$^{\circ}O(\log_2 n)/^{\dagger}O(1)$	$^{\circ}O(\log_2 n)/^{\dagger}O(1)$

TABLE III
OPERATION COMPLEXITIES OF IN AN FHT, [15] (A BLOOMIER FILTER), AND A PIHT.

$^{\circ}$ Time complexity † Pipelined throughput complexity

B. Memory Comparison for All IP Lookup Schemes

For comparison among all IP lookup schemes, we need to differentiate their number of transistors since a TCAM cell uses 16 transistors while a SRAM cell requires 6 transistors as shown in Table II. This consideration is illustrated in Fig.

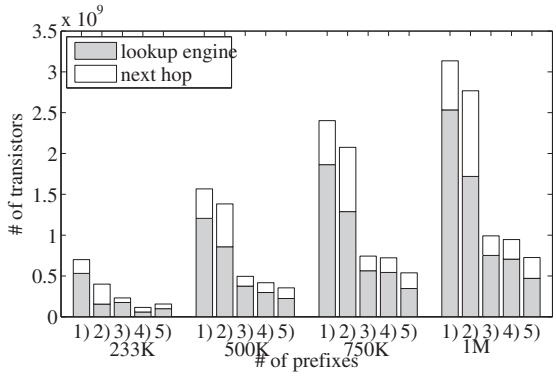

Fig. 10. Memory comparison for 5 schemes: 1) CBF+Peacock+Tree bitmap, 2) BFHT+PC, 3) TCAM, 4) Tree trie [6], and 5) PIHT+LB of b=8

10 by showing the number of transistors used in these five IP lookup schemes at various n. Although the inflated number of the next-hops in a BFHT scheme is 1.5 times larger than that of scheme 1) on average, the overall memory size of scheme 1) is 1.8 times larger than that of a BFHT scheme due to the pointer overhead. Additionally, the memory used by our PIHT-based IP lookup scheme scales well above other schemes, and it saves 4.5 times smaller than others' memory, at most.

C. The Power Comparison for All IP Lookup Schemes

This section details power measurement of all IP lookup schemes. We measured the power consumption of two parts: the longest prefix lookup marked as 'lookup engine' and a next-hop lookup for the matched marked as 'next hop' as in Sec. V-B. We understand that power is linearly proportional to load capacitance and the wire is the major source of capacitance in VLSI system based on the submicron technology. Thus, as the memory size is larger, the power consumed for memory access is larger proportionally. Since both BFHT bitmap and linked-list Peacock cause a next-hop inflation, consequently the next-hop lookups for these schemes incur 3 and 7 times more power consumption in comparison to other next-hop lookup schemes, respectively.

However, on average the overall power consumption of these schemes 1) and 2) is 2.5 times smaller than that of a TCAM. The main drawback of TCAMs is their prohibitive power consumption. In addition, because of the larger number of transistors required per bit TCAMs cannot be fabricated with the same density as SRAMs. Thus, TCAM schemes consume power 13.2 times more, on average, compared to a PIHT scheme. For only 'lookup engine' comparison, TCAMs use 50.1 times more power than a PIHT at n=1M. Also, Tree trie scheme [6] suffers from power consumption shown in Fig. 11, which is attributed by its need for sequential memory accesses along a trie. Thus, Tree trie scheme uses 2.2 times more power than a PIHT scheme on average.

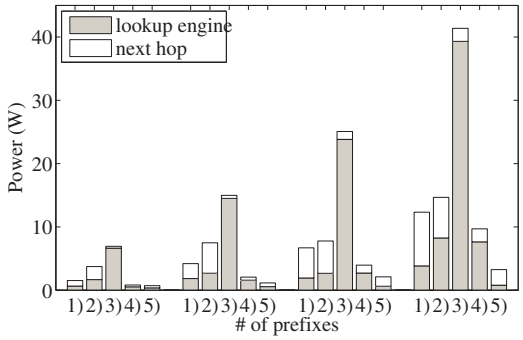

Fig. 11. Power comparison for 5 schemes: 1) CBF+Peacock+Tree bitmap, 2) BFHT+PC, 3) TCAM, 4) Tree trie [6], and 5) PIHT+LB of b=8

VI. Conclusion

The hash-based schemes have more attractive features such as lower power consumption, smaller memory and latency, compared to TCAM and trie-based schemes. However, there are three key obstacles that hinder their practical deployment. 1) The HTs incur collisions and use the chaining method to resolve them, and consequently this creates unpredictable lookup rates and introduces memory wastes [11, 19]. 2) A setup failure happens in an open-addressing collision resolution [19, 20] and a Bloomier filter [15], and this failure is not tolerable in IP lookup. 3) A hash cannot provide a prefix lookup and the current solutions generate an inflation of either prefixes or next-hops [12, 21].

We proposed a novel hash architecture to address these problems for IP lookup. The proposed scalable hash-based IP lookup scheme uses both a PIHT to do a b-nary key search, $b \in \{2, 4, \cdots\}$, in pipeline and a Lulea bitmap to make a smaller number of collapsed prefixes and next-hops. A PIHT in a prefix tree encodes keys' indexes and returns an i-index plus f-indexes, if any, in a key query as BFs' query results are used for a b-nary key search. In hardware implementation, a stage in a pipelined PIHT utilizes a memory banking system in order to remove multiport memory overhead and to provide an $O(1)$ pipelined throughput complexity.

We have successfully demonstrated that the IP lookup combination of existing hash schemes (a CBF, an FHT, a linked-list Peacock, and a Bloomier filter) and prefix conversions (PC and Tree bitmaps) are not scalable in terms of a routing table size and a forwarding speed. In contrast to other hash schemes, a PIHT reduces on average 7.3 and 9.8 times memory and die area, respectively. For IP lookup, a PIHT-based IP lookup scheme saves up to 4.5 and 50.1 times memory and power, respectively. Since we chose b=8 in a PIHT, we could get better efficiencies if b=16.

As a future plan, a PIHT can be used a basic building block for packet classification, intrusion detection as well as for generic content searches.

References

[1] K. G. Coffman and A. M. Odlyzko, *Internet growth: Is there a "Moore's Law" for data traffic?, Handbook of Massive Data Sets*. New York, New York: Kluwer, 2002.

[2] Mathew Gray, Internet Groth Summary. [Online]. Available: http://www.mit.edu/people/mkgray/net/internet-growth-summary.html

[3] BGP Routing Tables Analysis Report. [Online]. Available: http://bgp.potaroo.net

[4] V. Ravikumar and R. Mahapatra, "TCAM Architecture for IP Lookup using Prefix Properties," *MICRO, IEEE*, vol. 24, no. 2, pp. 60–69, 2004.

[5] V. C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, "EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 521–533, 2005.

[6] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[7] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines," *IEEE/ACM Trans. Netw.*, vol. 13, 2005.

[8] A. C. Snoeren, "Hash-based IP Traceback," in *SIGCOMM '01*.

[9] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching using Bloom Filters," in *SIGCOMM '03*.

[10] T. S. Sarang Dharmapurikar, Praveen Krishnamurthy and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," in *MICRO 37*.

[11] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup using Extended Bloom Filter: An Aid to Network Processing," in *SIGCOMM '05*.

[12] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A Storage-Efficient, Collision-free Hash-based Network Processing Architecture," in *ISCA '06*.

[13] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," in *SIGCOMM '06*.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

[15] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: an Efficient Data Structure for Static Support Lookup Tables," in *SODA '04*.

[16] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 3–14, 1997.

[17] CACTI. [Online]. Available: http://www.hpl.hp.co.uk/personal/ Norman_Jouppi/cacti5.html

[18] D. E. Knuth, *The Art of Computer Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978.

[19] J. T. Sailesh Kumar and P. Crowley, "Peacock Hashing: Deterministic and Updatable Hashing for High Performance Networking," in *INFOCOM '08*, 2008.

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

[21] V. Srinivasan and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion," *ACM Trans. Comput. Syst.*, vol. 17, no. 1, 1999.

[22] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2002.

[23] Integrated Device Technology Inc. [Online]. Available: http://www.idt.com/products

[24] H. Song, J. Turner, and S. Dharmapurikar, "Packet Classification Using Coarse-grained Tuple Spaces," in *ANCS '06*.

[25] B. Agrawal and T. Sherwood, "Virtually Pipelined Network Memory," in *MICRO 39*, 2006.

[26] T. Sherwood, G. Varghese, and B. Calder, "A Pipelined Memory Architecture for High Throughput Network Processors," in *ISCA '03*.

[27] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A Tree Based Router Search Engine Architecture with Single Port Memories," in *ISCA '05*.

[28] B. Agrawal and T. Sherwood, "Modeling TCAM Power for Next Generation Network Devices," in *ISPASS '06*, 2006.

[29] RIPE Network Coordination Centre. [Online]. Available: http://www.ripe.net/projects/ris/rawdata.html