A Hierarchical Hashing Scheme to Accelerate Longest Prefix Matching

Hai Sun
Washington State University
Email: hsun1@eecs.wsu.edu

Yan Sun
Huawei Research Center
Email: ysun@eecs.wsu.edu

Victor C. Valgenti
Petabi, Inc.
Email: vvalgenti@petabi.com

Min Sik Kim
Petabi, Inc.
Email: msk@petabi.com

***Abstract*—Longest Prefix Matching in IP Address lookup remains a bottleneck for high-speed routers where large volumes of traffic at multi-gigabyte link speeds require extremely fast lookup time. By taking advantage of bitmap and hashing techniques effectively used in Tree Bitmap algorithm and Binary hash searching on prefix length algorithm we propose a hierarchical hashing scheme based on observations about prefix length distribution in real routing tables. Theoretical analysis and experiments using real routing tables show that our scheme significantly improve IP lookup efficiency by remarkably reducing the number of memory access, consuming less memory and enabling fast update.**

I. Introduction

The rapid growing of Internet traffic demands routers to perform high-speed packet forwarding. Drastic increases in volume of traffic result in large routing tables and thus require faster lookup. As a consequence Longest Prefix Matching (LPM) for IP Address lookup becomes a challenging task, often the bottleneck in routers and efficient schemes to perform LPM in high-speed routers are needed.

A routing table in a router stores variable-length IP address prefixes and corresponding outgoing ports (next-hop). A prefix is an IP destination address with mask, such as 2001:0cb0:c202::/48 in IPv6 or 202.104.1.0/24 in IPv4. The mask is also called the prefix length. Two prefixes may have the same IP address and distinct masks. When a packet arrives, a router gets the destination address in its header, finds the prefix by means of LPM in the routing table, and forwards the packet according to the corresponding next-hop. Table I displays a simple IPv4 routing table with 7 entries.

TABLE I
An IPv4 Routing Table

IP Address	Mask	Next-hop (Port)
167.19.0.0	16	2
202.104.1.0	24	1
202.113.8.0	24	3
202.104.1.0	25	4
202.104.1.0	27	2
202.104.1.16	28	3
202.104.1.25	32	1

Several metrics are essential to evaluate the performance of IP lookup algorithms. Lookup time or searching speed is the paramount metric, essentially determined by the number of memory access (worst and average) as memory access is the most time-consuming step in the lookup procedure [1]. Memory consumption is also important for routing tables can grow quite large and may contain as many as four hundred thousand prefix entries. In addition, update complexity, efficiency, scalability and ease of implementation are also important for actual deployment as well.

In this paper, we survey real routing tables and and note that the prefixes in IP routing tables are not uniformly distributed. Upon two observations we design a two-layer hierarchy, combining the advantages of bitmaps, as employed in the Tree BitMaP algorithm (TBMP) [3], and the hashing mechanism from the Binary hash Searching On prefix Length (BSOL) algorithm [12]. Our scheme outperforms TBMP and BSOL with:

- fewer worst-case and average memory accesses due to fewer hash access levels;
- less memory consumption due to efficient bitmap-based prefix encapsulation;
- faster update and preprocessing, and better scalability due to reasonable hash access model.

The remainder of this paper is organized as follows. In section II we introduce related work. In section III we present details of our design. In section IV we describe preprocessing, lookup and update procedures. In section V we conduct theoretical analysis and illustrate experiments. Finally in section VI we conclude and present future work.

II. Related Work

Many LPM approaches have been proposed using various algorithms and data structures such as hashing and tries to reduce computational complexity. Hardware algorithms, e.g. TCAM-based approaches such as [5], search contents in parallel to achieve constant time complexity. However, they suffer from limited memory, high power consumption and scalability to routing table size. In contrary software schemes behave well due to low cost and flexibility. Our work follows software track by combining bitmap and hashing techniques.

The one-bit trie algorithm organizes the prefixes with the bits of prefixes to direct the branching [3]. However, its worst-case memory access is so large that many techniques have been incorporated to reduce the height of the trie (like Patricia [6], LC trie [7], LPFST [13] etc). Unfortunately these techniques lead to complicated update, or to memory expansion. TBMP

[3] proposes a compressed trie-node structure with improved memory consumption, searching speed, and update efficiency using two bitmaps to encode prefixes in one node. Only one extra memory access is needed to retrieve a next-hop lazily when the searching is done. Although TBMP performs better than other trie-based approaches, it suffers heavily from performance degradation, particularly searching speed. Our scheme effectively avoids these drawbacks.

Hashing has the outstanding feature of $O(1)$ lookup time which has prompted many network processors to have built-in hash units. Dozens of hash approaches have been proposed to address LPM for IP lookup. Authors of [12] propose the BSOL scheme with complexity of O($\log W$) where W is the length of IP address. BSOL separates a binary trie and stores nodes in each trie level into a hash table. Binary search is performed on hash tables of each level (or prefix length). It relies on markers and BMP (best matching prefix) to guide the next searching direction. Although it guarantees a bounded worst memory access, BSOL is a static algorithm in which all prefix BMP fields need to be recomputed when updating one entry. Our scheme greatly improves the hash efficiency on basis of BSOL.

III. Hierarchical Hashing

We outline core ideas behind our scheme that we call Hierarchical Hashing. The ideas originate from observations on real lookup tables and we design smart algorithm based on them.

3.1 Observations and Ideas

We survey real routing tables and make two observations: (1) prefix lengths are not uniformly distributed; (2) prefix lengths of multiples of 16 for IPv6 and 8 for IPv4 have much higher frequencies than others. The majority of IPv4 prefixes are a multiple of 8 bits and nearly half of IPv4 prefixes have a 24-bit length [11]. Likewise the most common IPv6 prefix lengths are 32-bit and 48-bit. The overall percentage of prefixes with lengths of 32, 48 or 64 bits are over 80% [4], [9], [10].

BSOL establishes one hash table regarding *every* prefix length. What about associating prefixes with *multiple* lengths to one hash table? This is the first idea obtained from $Observation1$ and BSOL. $Observation2$ incurs another idea about establishing as few hash tables as possible to facilitate preprocessing and lookup. $Observation2$ also suggests a reasonable empirical threshold to divide prefix ranges: 8-bit for IPv4 or 16-bit for IPv6. The first layer of our design comes from these two ideas. Moreover TBMP brings us the third idea for the second layer. The fundamental differences between our scheme and TBMP are the searching sequence (direction) and basic node structure shown in Figure 1.

Figure 1(a) left illustrates the searching direction in TBMP, a trie with k levels ($0 < k < M$, M is bottom level). The prefix length in each node level increases downwards. Consequently it takes more memory accesses to match a longer prefix than a shorter one if we reasonably assume a

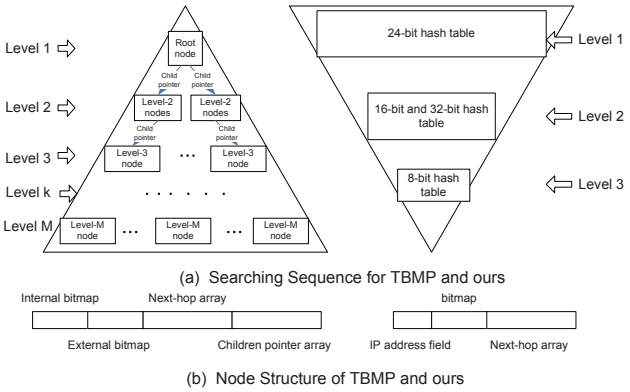


Fig. 1. Searching Direction

uniform matching probability for all prefixes. Meanwhile a great number of shorter prefixes have the same IP address fields as longer ones because of overlapping in real routing tables. This phenomenon accounts for the reasons why longer prefixes have higher possibility to be matched than shorter ones, and furthermore why the bitmap is efficient structure to aggregate multiple prefixes. Nevertheless from the perspective of the searching direction, TBMP is considered inefficient. Figure 1(a) right demonstrates the searching direction of our scheme. Our scheme outperforms TBMP in looking up longer prefixes with higher matching probability. We establish four IPv4 hash tables with three levels of hash access, covering prefixes with decreasing lengths downwards except special 32-bit hash table.

We compare trie node of TBMP and hash entry of our scheme in Figure 1(b). A TBMP node consists of two bitmaps (internal and external) and two arrays (children pointers and next-hops). In terms of the same internal bitmap technique in TBMP, our scheme aggregates prefixes with the same IP address and a variety of prefix lengths into one hash entry. We expect to avoid extra lookup and memory overhead using child pointer and external bitmap as they are necessary to guide searching in TBMP.

3.2 Scheme Details

Our Hierarchical Hashing scheme has a two-layer hierarchy. In the first layer are a limited number of hash tables. In the second layer each hash entry is a composite data structure.

To tackle the hash collision for IP lookup we note that the work in [8] allows prefix migration in multiple candidate hash buckets. We choose a chaining mechanism to resolve collision for convenience of implementation. Moreover a simple hash function such as CRC reduces collision to the minimum extent with negligible computation and lookup overhead. For the same reason we directly use string as key instead of a digest from hash function.

Suppose W denotes IP address width, 128-bit for IPv6 and 32-bit for IPv4. $HT_{i,j}$ for IPv4 or $HT^{v6}_{i,j}$ for IPv6 ($0 < i < j < W$) is a hash table with a W-bit IP address as key and encodes all prefixes in length range $[i, j]$. The size of length

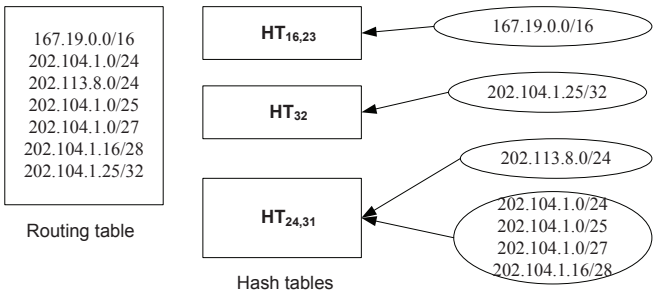


Fig. 2. A simple example

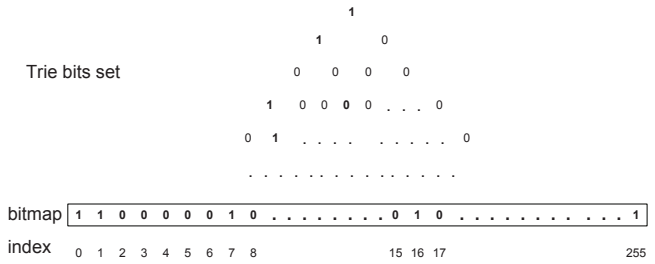


Fig. 3. The Bitmap for 202.104.1.0/24 Entry

range j-i+1 is defined as hs if it is a fixed stride for prefix range division. For example $HT^{v6}_{48,63}$ has 128-bit IP address as key and 16-bit hs, and encodes any prefixes with length in $[48, 63]$.

In section 5.3 we will discuss the pros and cons of a varying stride in our scheme if observations change. Now hs is fixed, 16-bit for IPv6 and 8-bit for IPv4. We construct hash tables of $HT_{i,j}$ or $HT^{v6}_{i,j}$ when i is times of hs. For example we establish IPv6 hash tables of $HT^{v6}_{16,31}$,$HT^{v6}_{32,47}$,...,$HT^{v6}_{112,127}$. The number of hash tables is calculated by $W \div hs$, 8 for IPv6 and 4 for IPv4. For W-bit prefixes we establish a special hash table, HT_W (HT_{32} for IPv4 or HT^{v6}_{128} for IPv6). Lookup on them is distinct from on other hash tables because of using exact match. We also establish another first-layer data structure, $Tnode_W$, to contain prefixes with length less than hs. For example, $Tnode_{16}$ contains all the IPv6 prefixes with length less than 16 bits.

The hash entry of each table except HT_W is a pair of $< key, BnodeList >$ when using chaining collision mechanism. The key is an IPv4 or IPv6 address. As shown in Figure 1(b) $Bnode$ denotes a composite class with three components: a hs-bit bitmap with 2^{hs}-1 bits, a next-hop array and an adr field. The adr field identifies the unique $Bnode$ in the list. Considering the scarce collision in actual implementation, the next pointer in a list can be neglected.

Two hash tables, $HT_{24,31}$ and $HT^{v6}_{112,127}$, are particular because $Bnode$ in them has a bitmap with 2^{j-i+1} bits, e.g. 256-bit for $HT_{24,31}$. The extra (least significant) bit indicates the existence of any relevant W-bit prefix in HT_W. For instance the 255^{th} bit set in bitmap (index from 0) of a $Bnode$ bn in $HT_{24,31}$ implies at least one 32-bit entry in HT_{32} has the same 24-bit masked address as the key of bn. In general for any IPv4 W-bit prefix in HT_W there must be at least one entry in HT_{W-hs} where its key is the IPv4 address masked by W-hs bits and the 255^{th} bit in bitmap of its $Bnode$ is set. It is also true for IPv6. Each entry in HT_W is a pair of $<$ key,next-hop$>$ where key is a W-bit address with W-bit mask and next-hop is a port value.

A $Tnode_{hs}$ has two fields, a hs-bit bitmap and a next-hop array, such that all the prefixes with length less than hs are encoded in it. A $Tnode_{hs}$ resembles a TBMP trie node except no external bitmap and children array.

Figure 2 illustrates how entries in Table I are roughly stored in $HT_{16,23}$, $HT_{24,31}$ and HT_{32}. 202.113.8.0/24 corresponds to one of the two entries in $HT_{24,31}$. The other entry encodes four prefixes with the same key of 202.104.1.0. Figure 3 demonstrates the bitmap in the $Bnode$ corresponding to the entry with key 202.104.1.0 illustrated in Figure 2. The bits set in index 0 (starting position), 1, 7 and 16 are mapped to prefixes of 202.104.1.0/24, 202.104.1.0/25, 202.104.1.0/27 and 202.104.1.16/28 as they are all encoded in the same $Bnode$ of $HT_{24,31}$. The bit in index 255 is set to indicate the prefix 202.104.1.25/32 in HT_{32}. Notice that 202.104.1.25 has the 24-bit masked address 202.104.1.0, exactly key of the entry in $HT_{24,31}$. The upper part of Figure 3 shows the tree-like structure with the same bit representation as the bitmap below. The root bit corresponds to 202.104.1.0/24.

IV. Preprocessing, Lookup and Update

4.1 Preprocessing Algorithm

The preprocessing store routing table prefixes into $HT_{i,j}$ (or $HT^{v6}_{i,j}$), HT_W or $Tnode_{hs}$ as illustrated in Algorithm 1 for IPv4 routing table. For simplicity the insertion of $nexthop$ in $Bnode$'s next-hop array is ignored. The algorithm for IPv6 is similar.

4.2 Lookup Algorithm

Given a packet lookup in IPv4 hash tables begins from $HT_{24,31}$ in two steps: fetch the $Bnode$ with the 24-bit key retrieved (masked) from the packet, and inspect its bitmap for the longest bit set. Under three conditions the lookup fails: (1) No such entry in $HT_{24,31}$; (2) No such $Bnode$ in the list; (3) No any bit set in the bitmap. If not found in $HT_{24,31}$, searching continues in $HT_{16,23}$, $HT_{8,15}$ and $Tnode_8$. Because the number of prefixes with length less than 8-bit is extremely scarce in real IPv4 routing tables, $HT_{8,15}$ is the actual end of searching. Hence the worst-case memory access is 4 for IPv4. The same bit-counting technique in TBMP is applied to find the longest bit set: remove the rightmost bit of the target IP address; check if the corresponding bit is set. If set the number of bits before this bit is counted. Otherwise remove the rightmost 2 bits of this address and check again until reaching the leftmost bit.

Algorithm 2 introduces searching in $HT_{24,31}$. For $HT_{16,23}$ or $HT_{8,15}$ it is similar except no access to HT_{32}. Eventually $Tnode_8$ is probed if no match in $HT_{i,j}$. Lookup in IPv6 hash tables is slightly tricky. We begin from $HT^{v6}_{48,63}$ and search

Algorithm 1 IPv4 Preprocessing

Initialize $HT_{8,15}$, $HT_{16,23}$, $HT_{24,31}$, HT_{32} and $Tnode_8$;
for all $pfx \in$ routing table **do**
  $mask$:= $parseMask(pfx)$;
  key_{32} := $getKey32(pfx)$; key_{24} := $getKey24(pfx)$;
  if $mask$ = 32 **then**
    if $key_{32} \in HT_{32}$ **then**
      if $key_{24} \in HT_{24,31}$ **and** key_{24} = $bn.adr$ for a $Bnode$ bn in the list **then**
        setBit($bn.bmp$,255^{th});
      else
        $HT_{24,31}.insert(key_{24}$, new $Bnode$ $bnNew)$;
        setBit($bnNew.bmp$,255^{th});
    else
      $HT_{32}.insert(key_{32}, nexthop)$;
  else
    if $mask \geq 24$ **and** $mask \leq 31$ **then**
      if $key_{24} \in HT_{24,31}$ **and** key_{24} = $bn.adr$ of a $Bnode$ bn in the list **then**
        setBit($bn.bmp$,corresponding bit);
      else
        $HT_{24,31}.insert(key_{24}$, new $Bnode$ $bnNew)$;
        setBit($bnNew.bmp$,corresponding bit);
    else
      Continue for $HT_{8,15}$, $HT_{16,23}$, and $Tnode_8$;

in two directions. Downwards any failed lookup in $HT^{v6}_{48,63}$ turns to $HT^{v6}_{32,47}$, $HT^{v6}_{16,31}$ and $Tnode_{16}$. Upwards we perform binary search on the hash tables with prefix length longer than 48-bit. Consequently the worst-case memory accesses are 5, the sum of 4 hash accesses plus one extra memory access to next-hop array.

Algorithm 2 Lookup Algorithm

Given an IP Address adr for lookup;
nh := $\emptyset$; key_{24} := $getKey24(adr)$;
if $key_{24} \in HT_{24,31}$ **then**
  for all $Bnode$ bn in the list of this entry **do**
    if $bn.adr$ = adr **then**
      if $bn.bmp.255^{th}$ = 1 **and** $adr \in HT_{32}$ **then**
        nh := $nexthop$ in the HT_{32} entry;
      else
        if found the longest bit set in $bn.bmp$ **then**
          nh := corresponding $nexthop$ in $bn.nhArray$;
else
  continue searching in $HT_{16,23}$, $HT_{8,15}$ and $Tnode_8$;
return nh;

4.3 Update Algorithm

We consider three primitive operations: insertion, deletion and updating partial fields. Insertion is described in Algorithm 1. Updating partial fields manipulates IP address, mask or next-hop. Updating IP address or mask in a $Bnode$ is equivalent to removing the entry and inserting a new one.

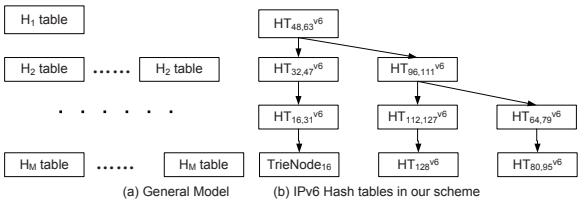


Fig. 4. Hash Table Access Model

Updating next-hop is carried out by locating a $Bnode$ and updating its next-hop array. In principle insertion and updating partial fields are simple. Deletion requires locating a $Bnode$ bn in one hash table with respect to IP address of the prefix. Next the bit in bn's bitmap is cleared. Meanwhile its next-hop is deleted. Even though the removed entry was the only prefix in bn, the empty $Bnode$ is kept after deletion. Because in some situations, e.g. prefix aggregation, multiple deletions and insertions occur in the same $Bnode$ and our strategy avoids the overhead of spawning a new $Bnode$ which is time-consuming.

Algorithm 3 illustrates deletion in HT_{32}, the most complicated operation involving two tables, HT_{32} and $HT_{24,31}$. In other hash tables deletion involves only one $Bnode$ in a single hash table.

Algorithm 3 Remove Prefix

Given Prefix pfx to be removed;
$mask$:= $parseMask(pfx)$;
key_{32} := $getKey32(pfx)$; key_{24} := $getKey24(pfx)$;
if $mask$ = 32 **and** $key_{32} \in HT_{32}$ **then**
  $HT_{32}.remove$(entry of key_{32});
  if $key_{24} \in HT_{24,31}$ **then**
    clearBit(bitmap of $Bnode$ bn, 255^{th}) and remove that $nexthop$ from $bn.nhArray$;

V. Performance Evaluation

5.1 Theoretical Analysis

5.1.1 Memory Access: K denotes the number of hash accesses. H_K is the level of hash access. M is the maximum value of K. Figure 4(a) demonstrates the general hash access model for BSOL and our scheme where $K \geq 1$ and $K \leq M$. Figure 4(b) shows our IPv6 hash access model with four levels. $HT^{v6}_{48,63}$ lies uniquely in the top level when $K = 1$; $HT^{v6}_{32,47}$ and $HT^{v6}_{96,111}$ are in the second level with $K = 2$; and so on. Note that $Tnode_{16}$ is at the bottom level. The maximum hash access number of M in our scheme in Figure 4(a) is 4.

Although in $HT_{i,j}$ or $HT^{v6}_{i,j}$ a value list may contain more than one $Bnode$ and therefore more than one memory access is needed to traverse the list, in practice we can reasonably assume that the majority of lists contain only one $Bnode$ given sufficient memory as well as a simple hash function. An extremely limited number of lists containing more than one $Bnode$ incur a trivial performance penalty. As a consequence we can claim that the number of memory access for hash tables in K-level is $K + 1$, K hash accesses plus one extra

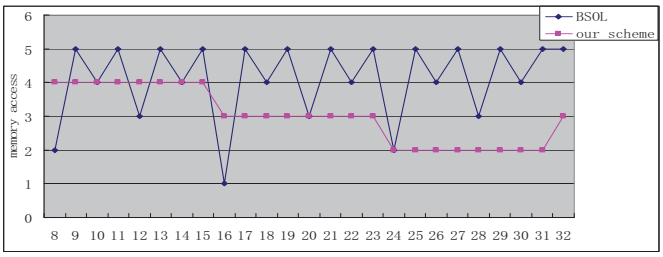
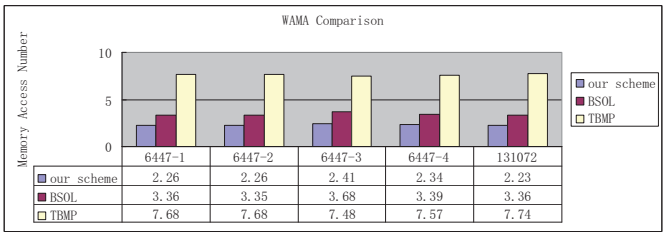


Fig. 5. Memory Access Comparison between BSOL and our scheme



	6447-1	6447-2	6447-3	6447-4	131072
our scheme	2.26	2.26	2.41	2.34	2.23
BSOL	3.36	3.35	3.68	3.39	3.36
TBMP	7.68	7.68	7.48	7.57	7.74

Fig. 6. WAMA comparison

memory access to next-hop array (the same lazy next-hop retrieval strategy as in TBMP).

The worst-case memory access occurs when $Observation1$ is not effective. In other words the distribution of prefix lengths is uniform. Our scheme still achieves $M+1$ memory accesses (4 for IPv4 and 5 for IPv6). The numbers for BSOL are 5 for IPv4 and 7 for IPv6 as BSOL conducts binary search on each hash table and increases hash access number. TBMP demands one extra memory access for each node for child pointer retrieval and hence has $2M$ (M is the maximum depth K) memory accesses, 8 for IPv4 and 16 for IPv6.

Figure 5 illustrates IPv4 memory access statistics for hash tables. There are no prefixes with less than 8-bit length in all routing tables. Our scheme is obviously better than BSOL in most lengths.

To evaluate average memory access we show the prefix distribution in our hash tables. We use five IPv4 and two IPv6 real routing tables from [9], [10], collected on May 29th 2013. They are 6447-1 (asppath), 6447-2 (aspath), 6447-3 (asorigin-max), 6447-4 (asorigin), 131072, 6447v6 and 131072v6. Their total prefix numbers are 285247, 281749, 193044, 192986, 351899, 12831 and 12503 respectively. We use Weighted Average Memory Access (WAMA) to effectively evaluate average memory access using our scheme. WAMA is defined by Equation 1. $K+1$ is the number of memory accesses for each hash table. $E_{i,j}$ is the number of entries in $HT_{i,j}$ or $HT^{v6}_{i,j}$. N is the total number of prefixes in a routing table. For example, K is 1 and $E_{24,31}$ is 114408 for $HT_{24,31}$, and N is 281749 for 6447-2 routing table. We consider the extra computation introduced by the 3^{rd} lookup failure condition depicted in section 4.2.

$$\frac{\sum (K+1) \times E_{i,j}}{N} \tag{1}$$

WAMA for BSOL is formulated in Equation 2. K is the number of memory access for each hash table in BSOL. E_K is the number of entries in each hash table. N is the total number of prefixes.

$$\frac{\sum K \times E_K}{N} \tag{2}$$

Likewise, WAMA for TBMP is formulated in Equation 3. Let hs-bit be the fixed TBMP stride. $2K$ is the number of memory access for each trie node. E_K is the trie node number in K level. N is the total number of prefixes. For example, K is 3 for any 24-bit prefix in the 3^{rd} level, E_3 is 114408 and N is 281749 for 6447-2 routing table.

$$\frac{\sum 2K \times E_K}{N} \tag{3}$$

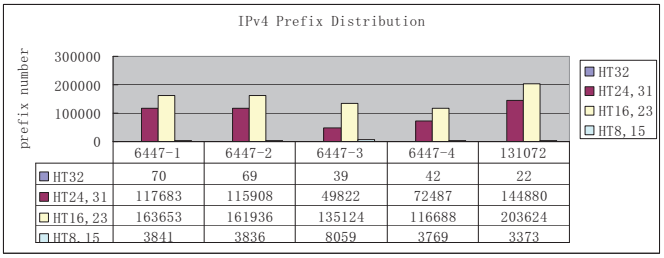
Figure 6 shows the comparison of memory access among these three schemes. Ours can save an average factor of 1.5 and 3 to BSOL and TBMP respectively. For IPv6 routing tables The WAMA comparison is even better as BSOL and TBMP both have worse memory accesses when W is 128-bit while our scheme scales well in IP address width.

5.1.2 Basic Storage Unit: To fully understand memory consumption, we examine the basic memory unit, hash entry for our scheme and BSOL, and trie node for TBMP. It is known that each entry of $HT_{i,j}$ or $HT^{v6}_{i,j}$ is a pair of key and a $Bnode$ list. Each $Bnode$ is made up of an adr field, an hs-bit bitmap with $2^{hs}-1$ bits and a next-hop array. The exception is $Bnode$ in $HT_{24,31}$ ($HT^{v6}_{112,127}$) with a bitmap of 2^{hs} bits. HT_W and $Tnode_hs$ contain trivial numbers of prefixes that we do not count.

Using a fixed hs a TBMP node consists of two hs-bit bitmaps, a next-hop array and a children pointer array. A BSOL entry consists of a key and a value of four fields, a marker flag, a mask, a next-hop and a BMP field. TBMP recommends a variable-stride (13-4-4-4-4-3) strategy to divide less node leaves with longer searching speed. Nevertheless memory consumption is still high. Our scheme consumes less memory than any TBMP implementation as no external bitmap and children array are used.

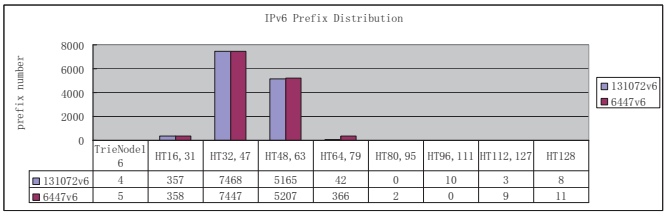
Each BSOL entry contains only one prefix. In our scheme each hash entry encodes as many as $2^{hs}-1$ prefixes using a bitmap. Therefore a bit sufficiently represents a prefix. More prefixes are represented in bitmap, better memory usage is obtained. Consequently our scheme scales much better than BSOL in routing table size. Moreover BSOL has extra memory overhead (at least 25% [12]) due to markers and BMP fields, essential to guide the lookup, with a tremendously heavy tradeoff in preprocessing and update. In terms of constant searching sequence our scheme avoids such overhead and allows fast preprocessing and update.

Therefore we claim that our scheme is proven to consume less basic memory than TBMP or BSOL. We will see the experimental result of memory consumption comparison using real routing tables which matches our analysis.



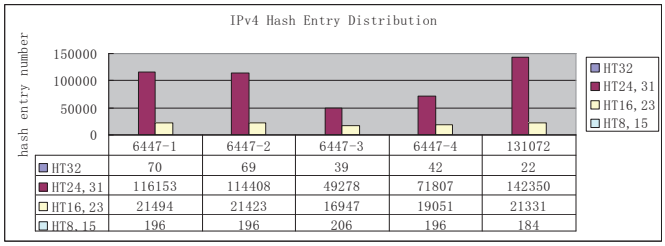
	6447-1	6447-2	6447-3	6447-4	131072
HT32	70	69	39	42	22
HT24, 31	117683	115908	49822	72487	144880
HT16, 23	163653	161936	135124	116688	203624
HT8, 15	3841	3836	8059	3769	3373

Fig. 7. Prefix Distribution for IPv4 routing tables



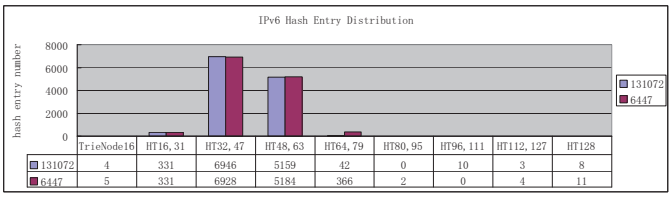
	TrieNode16	HT16, 31	HT32, 47	HT48, 63	HT64, 79	HT80, 95	HT96, 111	HT112, 127	HT128
131072v6	4	357	7468	5165	42	0	10	3	8
6447v6	5	358	7447	5207	366	2	0	9	11

Fig. 8. Prefix Distribution for IPv6 routing tables



	6447-1	6447-2	6447-3	6447-4	131072
HT32	70	69	39	42	22
HT24, 31	116153	114408	49278	71807	142350
HT16, 23	21494	21423	16947	19051	21331
HT8, 15	196	196	206	196	184

Fig. 9. IPv4 Hash Entry Distribution



	TrieNode16	HT16, 31	HT32, 47	HT48, 63	HT64, 79	HT80, 95	HT96, 111	HT112, 127	HT128
131072	4	331	6946	5159	42	0	10	3	8
6447	5	331	6928	5184	366	2	0	4	11

Fig. 10. IPv6 Hash Entry Distribution

5.1.3 Update and Aggregation: It is complicated and time consuming to perform update spanning several prefix ranges, such as frequent prefix aggregation in BGP routing tables. In BSOL placing markers and precomputation of BMP fields usually involves multiple hash tables. By comparison our scheme accesses only one hash table in most situations, even a single $Bnode$. For variable-stride TBMP several nodes have to be involved, causing more delay and complexity.

An aggregation update, a set of pertinent update operations, refreshes a routing table by aggregating a few prefixes with longer lengths into a short one. We examine real IPv4 routing tables and observe that (1) all the aggregations take place in the range of $[16, 30]$, mostly in $[24, 30]$ or $[16, 21]$; (2) all the prefixes in an aggregation have the same IP address.

Accordingly an aggregation takes at most two $Bnode$ access (mostly only one in $HT_{24,31}$ or $HT_{16,23}$), i.e. at most three memory accesses (two $Bnode$s in $HT_{24,31}$ and $HT_{16,23}$ separately) using our scheme. For variable-stride TBMP the number of memory access is much larger and highly non-deterministic, depending on the stride division and the length range of the aggregation. BSOL requires $O(t)$ memory accesses, linearly proportional to the length range t.

5.2 Evaluation using Real Routing Tables

We also use real packet traces from [9], [10] for evaluation. Figure 7 and Figure 8 illustrate the prefix distribution in hash tables (and $Tnode_{16}$) using five IPv4 and two IPv6 routing tables respectively. The distribution reveals how many prefixes are encoded in $HT_{i,j}$ or $HT^{v6}_{i,j}$. For example, in Figure 7 $HT_{16,23}$ contains 161936 prefixes for routing table 6447-2. $Tnode_8$ is always empty as there are no prefixes are shorter than 8-bit in all IPv4 routing tables. It is evident that $HT_{24,31}$ and $HT_{16,23}$, $HT^{v6}_{48,63}$ and $HT^{v6}_{32,47}$ encapsulate the majority of prefixes in IPv4 and IPv6 routing tables respectively.

Figure 9 and Figure 10 reflect the hash entry distribution in hash tables (and $Tnode_{16}$). The distribution shows the number of hash entries in each hash table.

Average Prefix Density (APD) indicates the average ratio of the prefix number to the entry number in each hash table. For HT_{32} and HT^{v6}_{128} the hash entry number is exactly the prefix number and therefore their APD is 1. For other $HT_{i,j}$ the APD values are 1.01 for $HT_{24,31}$, 7.69 for $HT_{16,23}$ and 19.67 $HT_{8,15}$ on average. Obviously APD values increase when i decrease. It is not true for IPv6 routing tables because IPv6 addresses are far from full utilization currently.

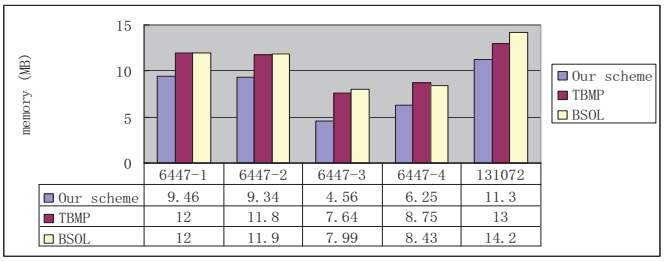
We implement our scheme as well as the basic forms of TBMP and BSOL in C++ and compare metrics of lookup time and memory consumption. The implementation environment includes g++ 4.6.3 compiler of the Ubuntu/Linaro, 3.0GHz Duo CPU with 32KB L1, 3072KB L2 caches and 4GB DRAM.

5.2.1 Lookup Time: We implement variable-stride TBMP, 13-4-4-4-4-3 suggested in [3], to reduce memory consumption with a compromise of lookup time.

TABLE II
SEARCH TIME PER IPV4 PACKET (μS)

Scheme	6447-1	6447-2	6447-3	6447-4	131072
Ours	3.64	3.62	3.73	3.68	3.50
TBMP	1,160	1,160	1,150	1,160	1,140
BSOL	8.17	8.16	9.01	8.24	8.30

Table II lists the lookup time using IPv4 routing table and packet traces. The tremendously long searching time in TBMP primarily attributes to much more node levels because of 6-level trie structure and an extra memory access for child pointer in each node. Our scheme outperforms BSOL with a factor of 2.5. Experiments using IPv6 data show similar comparison. Clearly the lookup experiments are consistent



	6447-1	6447-2	6447-3	6447-4	131072
Our scheme	9.46	9.34	4.56	6.25	11.3
TBMP	12	11.8	7.64	8.75	13
BSOL	12	11.9	7.99	8.43	14.2

Fig. 11. Memory Consumption

with the theoretical analysis of memory access in section 5.1.1.

5.2.2 Memory Consumption: Figure 11 compares the memory consumption comparison using IPv4 routing tables. In the actual implementation, we use appropriate data structures for basic memory units of three schemes. The experimental results endorse our analysis in section 5.1.2. Moreover we observe that small APD values (e.g. 1.01 averagely for $HT_{24,31}$) result from many consecutive zeros in bitmap of $Bnode$. Naturally we can achieve much better memory usage by means of any bit compression techniques such as one in [1].

5.3 Discussion

Our scheme is based upon two crucial observations. What if the observations change? An IP prefix assigned to an organization uncovers inherent network-related properties such as scale of network, sub networks, geographic distribution, etc. Our observations honestly reflect the underlying connection between routing tables and these network properties. Thus our observations, particularly $Observation1$, always hold. Nevertheless $Observation2$ may change. For example the total percentage of prefixes with lengths of 32, 48 or 64 bits in IPv6 routing tables is not the majority in the future as IPv6 addresses and prefixes are large enough to be assigned more irregularly. Furthermore, the majority of prefixes in some IPv4 routing tables in edge routers are not multiple of 8 bits in length.

We recall the analysis in section 5.1.1 which guarantees the worst-case memory access despite that the prefix length distribution is uniform. Our hash-bitmap hierarchical structure is apparently resilient to changes of observations and assures of few memory accesses even if $Observation2$ varies. In addition we use a fixed hs (8-bit for IPv4 and 16-bit for IPv6) to divide length ranges for hash tables which encode prefixes in $[i, j]$ where i is times of hs and j=i+hs-1. The rigid stride arrangement restricts the adaptivity of our scheme when $Observation2$ varies. To address the problem, we may use varying strides. A pre-survey for a given routing table can be performed before preprocessing in order to attain actual prefix length distribution. Like variable-stride TBMP, this upgraded scheme can configure varying length ranges for hash tables flexibly in term of the database-dependent result from pre-survey so that each $Bnode$ in various hash tables may have different bitmap size, consistent with its length range. Clearly the variable-stride strategy benefits from better memory usage and adaptability to routing tables though it trades away lookup time and update efficiency due to potentially more hash access levels.

Scalability in routing table size and IP address length does matter [2]. As the worst-case memory access sharply increases in TBMP, our scheme guarantees small WAMA values due to the balanced trade-off between throughput and storage. In addition, TBMP suffers from the varying length of children array. BSOL consumes much more memory and performs update tremendously complicated when routing tables expand or IP address length rises. Our scheme scales well in both scalability aspects.

VI. Conclusion and Future Work

We propose a hierarchical hashing scheme to handle LPM for IP lookup. We survey real routing tables and make two crucial observations upon the distribution of prefix lengths. Based on the observations we combine bitmap and hashing mechanisms to build up a two-layer data structure to efficiently contain and encode prefixes. Through theoretical analysis and implementation evaluation on real IPv4/IPv6 routing tables, our scheme outperforms TBMP and BSOL in every performance metric. We can improve current work by saving memory consumption and using better collision resolution policy. As we indicate in section V, use of any bit compression skill definitely reduces memory consumption of bitmap in $Bnode$, particularly for $HT_{24,31}$ (or $HT^{v6}_{48,63}$) with extremely low APD values. Use of better collision resolution can further reduce the average length of $Bnode$ list when hash tables have high load factor.

References

[1] M.i Bando and H. J. Chao. Flashtrie: hash-based prefix-compressed trie for ip route lookup beyond 100gbps. In *IEEE Infocom*, pages 821 – 829, 2010.

[2] Y. K. Chang and Y. C. Lin. A fast and memory efficient dynamic ip lookup algorithm based on b-tree. In *IEEE AINA*, pages 278 – 284, 2009.

[3] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: Hardware/software ip lookups with incremental updates. *ACM Sigcomm Computer Communication Review*, 34:97–122, 2004.

[4] Y. M. Hsiao, Y. S. Chu, J. F. Lee, and J. S. Wang. A high-throughput and high-capacity ipv6 routing lookup system. *Computer Networks*, 57:782–794, 2013.

[5] S. K. Maurya and L. T. Clark. A dynamic longest prefix matching content addressable memory for ip routing. *IEEE/ACM Transactions on VLSI Systems*, 19:963–972, 2011.

[6] Donald R. Morrison. Patricia:practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[7] S. Nilsson and G. Karlsson. Ip-address lookup using lctries. *IEEE Journal on Selected Areas in Communications*, 17:1083–1092, 1999.

[8] F. Pong and N. F. Tzeng. Concise lookup tables for ipv4 and ipv6 longest prefix matching in scalable routers. *IEEE/ACM ToN*, 20:729–741, 2012.

[9] Potaroo project. bgp.potaroo.net. In *routing table and packet trace*, 2013.

[10] Routeview project. www.routeviews.org. In *routing table and packet trace*, 2013.

[11] Y. Sun and M. S. Kim. A hybrid approach to cam-based longest prefix matching for ip route lookup. In *IEEE Globecom*, pages 1 – 5, 2010.

[12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *IEEE Sigcomm*, pages 25 – 36, 1997.

[13] L. C. Wuu, T. J. Liu, and K. M. Chen. A longest prefix first search tree for ip lookup. *Computer Networks*, 51:3354–3367, 2007.